# Component Based Software Development for the Internet

Kris Luyten
kris.luyten@student.luc.ac.be

October 2, 2000

# Component Based Software Development for the Internet

Kris Luyten

Thesis voorgedragen tot het behalen
van de graad van doctorandus in de kennistechnolgie,
afstudeervariant informatica/multimedia



*promotor* prof. dr. Karin Coninx
*co-promotor* Benny Daems
*supervision* Jan Van den Bergh
*academiejaar* 1999-2000
*School voor Kennistechnologie*
*Limburgs Universitair Centrum - Universiteit Maastricht*

# Contents

# List of Figures

# List of Tables

# Acknowledgements

I would like to thank the following persons, who all have made a difference in making this thesis;
*Prof. dr. Karin Coninx*, my promoter, for her suggestions and correcting and helping me with the thesis,
*Benny Daems*, my co-promoter for giving useful hints and tips and for feedback on my early drafts,
*Jan Van den Bergh*, for correcting and giving feedback on my text and guiding me through the tons of available data,
*Steven Fransens*, for correcting a great deal of my English spelling,
*Kristel Clijsters*, for her daily support, love, motivation and wonderful discussions,
*my parents and my sister*, without who I would not have had the chance to put all my time in studying,
and all the friends I have had the chance meeting during the past four years, who have made it all time well spent.

# Abstract (Dutch)

Software development kende een grote groei sinds de enorme bloei die het internet gemaakt heeft. De vraag naar software "bouw blokken", klaar voor gebruik, heeft geleid naar component gebaseerde software ontwikkeling. Dit maakt het gemakkelijk om nieuwe software te bouwen met de reeds bestaande blokken en ze op de markt te brengen voor gebruik. Zelfs zonder het internet zou component gebaseerde software ontwikkeling furore gemaakt hebben als de gedoodverfde opvolger van het object geörienteerd programmeren. De opkomst van netwerken in het algemeen en niet-gespecialiseerd gebruik heeft bijgedragen tot sommige krachtige eigenschappen die software componenten de dag van vandaag bepalen. Enkele voorbeelden zijn transparantie, beveiliging en overdraagbaarheid. Deze thesis concentreert zich op transparantie en beveiliging van componenten die geschikt zijn voor het internet. Er wordt bekeken hoe de verschillende component modellen zich tot elkaar verhouden met betrekking tot deze eigenschappen en hoe ze zich verhouden ten opzichte van een algemeen component model. Alle relevante eigenschappen van componenten, geschikt voor het internet, beschouwen zou de studie te uitgebreid en te oppervlakkig maken. Deze thesis heeft niet de pretentie een volledige beschrijving noch een volledige vergelijking te maken over beveiligings aspecten voor het internet of internet componenten.

# Abstract

Software development has made a major growth since the worldwide internet boom. The demand for ready to use software building blocks leaded towards a component based software development approach which makes it easier and faster to build new software, ready to deploy. Even without the internet object-oriented programming would have evolved into CBSD, but the introduction of networking into software development added some powerful properties to software components available today, like transparency, portability, security,...

This thesis will focus on transparency and security as properties of components, because of their importance for the internet. Considering all properties would be an impossible task, and by taking two of these properties there is more room for a detailed approach. The security properties of components will be discussed more in-depth, while the transparency properties will be discussed less. This thesis has not the intention to be a full investigation of internet security or a full comparison of all security possibilities of components. It is impossible to give a full description of all security services available today, so the discussion is limited to security services in particular important for CBSD for the internet. Making a comparison between several component models, like JavaBeans, COM and CORBA, will be done where it is relevant and possible, founded on a general component model introduced in the first chapter.

# Chapter 1

# Internet Components defined

What exactly are Internet Components? This section will try to give some answers to this question. First the component concept is roughly explained from a developers point of view. Next the use of these components for Internet services will be explained. In these discussion we will be mostly interested in the security aspects from these components.

## 1.1 A software component

It is not easy to exactly define the properties a piece of software must have in order to be called a component. In the following sections I will try to give the constraints which are typical for software components (if they can be called 'constraints'). In the literature we do not find always uniform definitions for the concept; some are more flexible than others. Some definitions of components summed together in [Jan99] are:

- "Components are pre-developed pieces of application code that can be assembled into working application systems." [Ann98b]
- "A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties." [C. 97, Cle97]
- "component-oriented programming = polymorphism + (really) late binding + (real, enforced) encapsulation + interface inheritance + binary reuse."[Don98]

A more general definition including parts of the second and third previous definitions is: in general, components have the following challenges to target:

- being platform neutral;
- offering an interface representing the possibilities and restrictions (a contractual specification);
- independent from the programming language being used;
- supporting distributed use with a specification for network communication;
- supporting reusability without re-compilation across applications (easy deployable);

The best approach is to start with an approximation of the most tight definitions because the languages and development tools which support component oriented software development are not perfect and will weaken the definitions stated. This way it is easier to see where the differences between them are situated. Before we start, one thing is certain: components would not be available today were it not for object oriented programming. The concept of object oriented software development is the origin of components and the reader should have a good knowledge of object oriented properties and concepts to be able to follow the discussion.

## 1.2 Guidelines and properties for components

A software component only has its interface visible to the user, a programmer in this context. Components are almost always linked with their host application at runtime. A comparison that has been used a lot to represent this idea is a *blackbox*. This is not to be confused with blackbox-testing, which is a technique used for testing software systems (mostly separate objects in the implementation), though it represents a parallel line of thoughts. The user knows how the outside looks like and what the blackbox can accomplish, but does not know how the blackbox works internally. So, components can be reused without recompiling if the user knows their interface. An interface can be defined as: "A group of semantically related functions, or methods. Taken together, the methods in an interface define a logical group of services that a component object can provide to the system." according to [Pau95]. This principle is emphasized by the use of interface definition languages, where only the specification of a component can be described, independent from any implementation. All interface functionality should be defined with pre- and postconditions and invariants which are 'legal' throughout its use. If so, a programmer can use components as building blocks for new applications. Important in this thesis is that components need *not* to be just the way they are if they were local on the users machine. They can reside in other or the same applications, local or non-local (location transparency). The user does not have to know where the component resides for instantiating it and does not mind component migration through the (networked) system. Of course, there are means to force this to be true. But certainly not all implementations have reached this degree of transparency. The following summation of properties of components are thoroughly discussed in [KK98].

### 1.2.1 Object oriented inheritage

As is said before, object oriented programming is the origin of components. It is clear that most of the concepts we encounter in this programming method must also be part of component oriented programming. The following paragraphs identify which of these properties a component must have to be classified as a real component. This list is discussed in more detail in [KK98] and [Ham97] for JavaBeans in specific. The reader is assumed to know what objects, classes, interfaces, instantiation (construction) and finalisation (destruction) are. These concepts are basic knowledge for the OO-programmer, and they apply to component models as well as they apply to regular OOP.

Assuming the reader knows what object orientation means, it is trivial to see that this is the main foundation for components. There is one major difference though; the interface must be separable from the object. Otherwise the *blackbox* principle is violated. It is possible to violate this principle and still have a piece of software that acts as a component (like an abstract data type delivered in a dynamic link library). If we are talking about classes it is naturally to mention object creation and finalisation. This comes into mind when we think of components

as objects exporting their public functions. Objects need to be created, initialized and finalized when residing in a running application, and if the component is integrated into an object oriented system it must have notion of these concepts. Inheritance is not a concept that was well supported in the component world (notice this an improving concept. Nowadays the CORBA-specification provides inheritance through its IDL [Obj99] section 3.7.2). Many programmers saw the *blackbox* principle as hard to use in inheritance. I do not think this is a fair opinion: when defining carefully the post- an preconditions and invariants the user should be able to know quite detailed which effects can be reached and which not. It is true that data encapsulation does not allow the user see the true insight, but as stated; a good design can overcome this problem. In a good design of a component the interface should provide sufficient information to use it like a regular object.

This is a good moment to reflect about contractual and defensive programming. Defensive programming is the old-style programming where methods do all the checking and most of the times, in case of an error, the error-code is returned instead of the value expected to return when the method encountered no errors. There are a lot of programmers still programming this way (C-programmers did it, C++-programmers also used error-indication by return value, and the use is still reflected in the `HRESULT` COM methods return). The other and preferred way is contractual programming and a division between inspectors (inspect (part of) the state of object) and mutators (change (part of) the state of an object) . The programmer specifies each method with the following specification features i.e. specifies a **software contract** for the class:

**pre-conditions** describes which conditions and constraints must be fulfilled when invoking the particular method, this could be describing possible values of arguments passed to the method;

**post-conditions** describes the effect the method invocation had on the object in case the method is a mutator;

**class invariants** describes which conditions and constraints are always valid during the object's lifetime;

**exceptions** describes which exceptions could be thrown, and which can be the cause of the particular exception;

**return value** describes the return value in case the method is an inspector.

Unfortunately, on this moment most programming languages do not really support all of these conditions, but only partially. Extensive research is already done in this area and a lot of programming languages like Java, Smalltalk, Eiffel and C++ do support this contractual approach to certain extents [Gre92, Eri98, Dou92, Pet98]. Nowadays, most of the time a combination of defensive and contractual programming is used, depending on the purposes of the object. E.g. is it meant to work in a real-time environment, contractual is preferred because there is less checking involved at the core classes. An object working in a life-critical system could be a defensive one, ensuring correct execution instead of a possible stateless system if a method fails because the contract was not respected by the user of the object.

Polymorphism, often said to be one of the foundations of OOP, cannot be omitted in this discussion. It is polymorphism that contributes a great deal to reusability of the components (e.g. one can treat different components with the same goals in a similar way). A drawback is

the need for exception handling in runtime type handling. This is independent whether the used programming language is statically typed or not; a component should be enabled to be linked at runtime. Exception handling is not only important for error prone type casting, there are more uses one can imagine such as runtime array bounds checking, indicating illegal resource access,.... It is clear a good component model must provide an exception handling mechanism. Fortunately most of them do. It is no wonder why the CORBA IDL and the Microsoft IDL (for COM) are providing exception handling. The Java language gives a good example: the exceptions that may occur in execution are part of the specification of the particular method. I have stated before an interface should define pre- and postconditions and invariants. In case a clear exception model (or another error handling mechanism) is available this must be part of the specification too. The user of a component will appreciate this because there is less chance of meeting unexpected errors during execution which were not found at compile-time. Exception handling may seem strange for novice user, but becomes a more simple approach of handling *unexpected* situations while executing a piece of code. When a programmer becomes used to the idea of exception handling through the use of try-catch(-finally) block statements it just becomes a natural extension of their programming methods. It also helps writing cleaner and more comprehensible code.

The items discussed next are more related to what one would consider as properties of components. Composition, introspection and persistence are keywords that are used a lot when discussing a component model. Without composition it is not possible to build an application with different components. Introspection is something completely different, most of the component models do not support introspection at all! Users of the Bean Development Kit (BDK) are familiar to this concept: they can import a JavaBean and immediately see the available interface without running it first. Introspection allows a component to reveal itself for the user, it shows itself. There is no uniform way defined for introspection, several methods are used to allow a component to present itself. This will be discussed later. Some components even allow to be adapted at design-time, rather than at run-time. For example, when a bean is imported in the BDK, a list of available properties, methods, events and exceptions is displayed. The user can choose a property ("this bean has blue text") and change it ("this bean has yellow text now"), this will not only be reflected at runtime, but also while working in the BDK.

A mechanism used to allow components to store their state is called persistence. Like introspection, persistence is not represented very well in most component models. Persistence allows a component to save its state to a persistent storage medium and to reload its state. A component model can handle this in two different ways: let the component implement the storage procedure itself or implement a centralized mechanism for storing it. This implies that a program can stop running and resume its execution with the same state it had stopped executing before. [KK98] states three base rules for persistence (Persistence independence, Persistence data type orthogonality and orthogonal persistence management, [Sø97] discusses this further).

[Ham97] also discusses introspection, persistence and composition in further technical detail particular for JavaBeans. It is clear that persistence has its limits when using composition as viewpoint; saving pointers (in memory or to a storage medium, like untyped void pointers in C++) to other components is a bad idea because it is not sure they will be residing in the same address space next time. A certain amount of location transparency for our components is required. [Ham97] also provides a thoroughly discussion of introspection, one of the key properties of the JavaBeans model. The interface of such a JavaBean is defined using *Design*

*Patterns*, which defines the syntax to use. This is not to be confused with object oriented software design patterns also called design patterns in [Eri95].

### 1.2.2 Resource, version and data management

While developing software, it is natural to have more than one version of your builds. Most of the time components also oblige to this law of incremental development. Say you have developed a component and put in on the web for free use, calling the component `MyFirstComponent`. One month later somebody reports two or three major bugs and you correct them. You put the new version on the web and calling this component `MySecondComponent`. If you do so, you will have to support the same interface used in `MyFirstComponent`. In evolving over time and developing sequential versions of the same component the developer has to make sure the interface of the previous versions stay supported. Lets take this one step further: reviewing all properties of a nearly perfect component it should be possible for the component to replace itself with a newer (and probably better) version without the users knowledge. This requires a good versioning system, and even if it cannot replace itself there should be a global system service taking care of the component versions and replacing them appropriately. There should be no need for rebuilding, recompiling or notifying the application from the change; dynamic replacement is a minimum demand. The JavaBeans component model is one model that lacks this functionality, there is no support for versioning at all. ActiveX components on the other hand have a pretty good versioning system.[1] This is because ActiveX resides on COM and it is this technology that is responsible for providing the versioning system. COM defines binary interfaces for the component in order to become language independent on several levels.

Specification is also an important concept in versioning. Backward compatibility is only possible if the newer components support the interface (post- and preconditions, invariants and exceptions) of their predecessors. This means an implementation of a method can be changed as long as it does not change the specification. If we want to be very strict about this we could say that blackbox tests are forced to have the same results with the new version as the results obtained with the predecessor. We cannot state this for whitebox tests because they do not test the interface isolated from the implementation but consider all possible paths in the implementation of the tested piece of software. It is advisable to take care whitebox tests also satisfy the conditions valid for the blackbox test and give the same results for newer versions of a component. Using this line of thought it is more assumable a component will react the same way using the interface as was described in a predecessor in extreme situations.

The next item in this discussion is how transaction management should be supported in a component model. This is a topic closely related to the topics in the next section which deals with distributed concepts. Actually it should appear in both this and the following sections. I choose to put it here, but that does not mean it is not related to distributed concepts, not at all. Since a component is expected to have access to shared resources the component model should include the necessary guidelines for dealing with this. A transaction is defined as:

> A sequence of server operations that is guaranteed to be atomic in the presence of multiple clients and server failures.

[1] Many programmers do not think of ActiveX as components but as a technology that allows other technologies to work together. They are right in one extent, that is: ActiveX *can not* be defined as a single technology, it is more like a glue; glueing different technologies together in one working component. Do not confuse ActiveX Controls with ActiveX because they are not equal, see section 3.2.1.

Consider two components, both writing the same file. This a situation well-known in the databases; there is a risk the data is not consistent if the actions the components has executed are not serial equivalent. There are several methods for dealing with this problem such as using locks on the data, optimistic concurrency control or timestamp ordering [Geo94]. There is a lot of literature available for distributed transactions (and nested transactions) and the commit protocols used in this situations. Because these do not fall in the scope of this thesis and will not be further discussed into detail. Further reading can be found in [Geo94] which gives a good introduction focusing on these topics with distributed systems in mind. The reason guidelines for transaction management are included in a component model is not intuitive, nevertheless very necessary. Components are not restricted to a single existence on a single machine, but can be migrating from one machine to another and sharing a lot of resources along the way (network-connections, files, databases, screens,...). In practice most serious component models indeed have implemented transaction management (e.g. the Microsoft Transaction Server, Enterprise JavaBeans with the Java Transaction Architecture, the CORBA Transaction Services Specification, IBM Customer Interface Control System (CICS),...).

### 1.2.3 Distributed concepts

Until the previous paragraph, every concept discussed here deals with components for use on a single host. One can remark version management is also subject to networked computers. This is partially true because version management is something to be considered for a particular user, but the "component-dealer" is probably another host on a computer network. Thinking of component models as only applicable for a single host approach is obsolete, nowadays interconnections must be taken into account. This viewpoint forces us to have guidelines for components on distributed systems (or normal computer networks) as well.

A successful component cannot be restricted to a platform, programming language or location. This means a component, at its very best, must be fully *portable*. This portability has its consequences for the instantiation of components. Consider a component, ready for use, but residing on another host machine then the target machine. Should it be

- transported from the remote machine to the local machine and be instantiated and executed on the local machine (remote-local) or
- kept on the remote host and be instantiated and executed there (remote-remote)?

There are still two other possibilities: it is obvious that the case local-local (a component resides and is executed and activated on the same machine) is trivial for understanding. Local-remote (a component resides on the client (local) and is transported to the remote host and instantiated and executed there) is the fourth possibility and is almost never used. Most commercial component models already support the first kind of migration of components. More particular JavaBeans and ActiveX are transported to the client machine and executed there, which is, as we will see, not always the best or most elegant way of doing things. DCOM as one of the successors of COM provides functionality especially for distributed concepts. The demand for migration possibility does imply other guidelines to be taken care of, like location transparency.

The ANSA Reference Manual [Cas89] and the International Standards Organization's Reference Model for Open Distributed Processing [Org92] define eight forms of transparency for distributed systems (for entities/information objects, but redefined for components here). When

the word client is used it is considered to be an entity—user of a component (whereby entities can be defined as persons or applications)

**Access transparency** Accessing locate or remote components should not make any difference in the used operations.

**Location transparency** Clients are not aware if the component(s) they are using are local or remote.

**Concurrency transparency** Using shared data components should not have any affect on the consistence of the data neither interfere with each other.

**Replication transparency** Multiple instances of components can upgrade reliability and sometimes performance. The client should not be aware of the multiple instances residing over the system.

**Failure transparency** This item is not applicable to software components. It says that users should be able to complete their tasks in case of software or hardware failure.

**Migration transparency** Components are allowed to move within a system without the clients needing to be aware of their movements and not causing errors due to moving.

**Performance transparency** clients must be able to reconfigure their use of components for the purpose of improving performance.

**Scaling transparency** Like Failure transparency, this item is not directly related to transparency defined for components. It means the system is allowed to expand without change to system structure and implemented algorithms. But it is closely related to load balancing which is directly applicable to components. This indicates an indirect link to replication transparency.

Despite the fact that failure and scaling transparency do not influence component architecture directly, they have implicit consequences in designing the architecture. For example, when the software fails it is important for some components that they do not keep on running with inconsistent data. Scaling transparency is actually already taken care of: version management can be a tool to overcome changes in system structure and replication transparency can overcome difficulties in case of increasing system scale. The two most important and most widely used kinds of transparency are *access transparency* and *location transparency* together also referred as network transparency. It is most likely that network transparency will have a major growth in the upcoming few years, proportional with the growth of importance of distributed systems. A distributed systems relies on different components working together over a network, and a connection between components in single host software and distributed systems is quickly made.

Finally, one of the most important issues in distributed system development, and the components that come with it, is *security*. Because this is an important topic of this thesis, only a short introduction is given here and a whole chapter is dedicated to this subject later on (see chapter 4). When we are talking about components and security we take into account the different levels of security issues (as well vertical as horizontal). Vertically we go from language or technical dependent security on the lowest level to language independent authentication logic and cryptography on the highest levels. This means we consider the security related to components in being on a different level as security related to the network itself. The border between levels is

rather vague. Horizontally we consider the range of technologies and algorithms available for each vertical level. It is certainly not easy, probably impossible, to define security as a whole. It is more like different components which can be added together to make a system more safe rather than one set of rules making a system safe, especially when we are considering distributed applications or applications using network connections. Unfortunately we can never say a system is totally safe in the real world(though theoretically speaking a formal prove of a "secure" system is possible). Is best to focus on making this a *more secure* instead of a *secure* system. there have to be taken into account a lot of threats when investigating security issues for components. Some questions that one can ask about how secure a system is are:

- Can the client be trusted?
- Can the remote server be trusted?
- Are there possible tampering or store-and-forward threats?
- Which cryptographic algorithm is most suited for the tackled problems?
- How secure is the programming language in use?

### 1.2.4 Other aspects of components

Besides all aspects considered in the previous discussion, there are less global properties of components. I would like to emphasize the fact that the following properties are not to be considered as rules or guidelines for components models. They are rather optional functionality for making the use of components more attractive and probably a lot easier.

Graphical user interfaces are becoming more and more important in software development. Designers and programmers are continuously trying different methods and approaches for making more comfortable to use software. It is no wonder user interfacing marks some component models as well. Building a user interface by choosing out of a set available components is becoming the major theme in recent GUI- API's. OLE Controls, ActiveX Controls and JavaBeans take user interface building one step further providing customisability and flexibility which are required in modern software development. GUI-components can be seen as prebuilt OOP software pieces ready to use, without the complexity we encountered e.g. by the old C winapi for developing Microsoft Windows applications. There are a lot of GUI toolkits available on the market nowadays offering prebuilt UI elements and components to the user. MFC, JFC, Qt, OWL and a lot of other libraries are available for reasonable prices. Why do it the hard way when you can use these, unless you could do it better then the professionals? Not only it is much easier to build a GUI, it is also a lot easier to model and analyze it because of the information already known before creating the GUI.

Customization is another aspect of components, most of the time GUI related. To enhance tools for building applications with component models, it can define another way to access its properties than by code. Just think of the Bean Development Kit, JBuilder or Visual Studio. Components can be imported and enhanced accessing properties with a visual interface. One model is far ahead of the rest at the moment of writing: the JavaBeans model. By their introspection facility it can be used together with a visual interface for accessing its properties at design-time. This is a promising development for the future, because programmers do not

have to concentrate on the UI-handling any more, but can concentrate on the underlying data-model. The view on the data and the independence of UI- and data-layers can be improved using these techniques. Because a full description of GUI possibilities falls outside the scope of this thesis, it will not be discussed further.

### 1.2.5 Summary

Because of the large quantity of component model properties discussed in this section, here is a small summary of all properties related to the reference model we have discussed so far here. These can be considered as general guidelines for evaluating existing component models.

- object oriented concepts (classes, interfaces, construction, destruction, objects,...)
- blackbox principle
- polymorphism
- exception handling
- composition
- introspection
- persistence
- versioning
- transaction management
- dynamic replacement
- portability
- transparency
- security
- GUI
- customizability

## 1.3 An internet component

### 1.3.1 What are internet components?

Now that we have defined a general model for components we can extend it toward components for the internet. Actually, few changes must be made to the concept of component model already discussed to be internet components. However, there are some properties more stressed for internet components then for components targeted towards single host software. For example security, location transparency and dynamic replacement are important properties for components likely to operate on several hosts connected to a network. An internet component can be defined as a component enabled to have its interface used over a network connection (more in particular a TCP/IP connection) while the implementation is situated on a remote host. In plain English: the place of input and output can be separated from the place of the actual processing of the data.

### 1.3.2 What is the use?

Most present components support use in a network environments. The software market, focusing on components specialized in internet services, is growing fast. One of the most well-known example of an integrated and internet-able component architecture is the Microsoft Windows 98 Operating system. Their desktop is composed out of components also used in their internet browser. The internet browser is more then just an internet browser. It is part of the desktop and able to read different types of documents. The HTML-browser is just a component that can be replaced at runtime. It can be replaced by another component for reading the portable document format from Adobe, reading Word documents, listening to music, viewing pictures and much more. Because the Microsoft browser is a component itself it can be instantiated by another application providing also the ability to read hypertext documents using the component.

Internet is a very fast evolving medium and used by a growing number of users. Without going into ethical questions, it is the number one source to find data of all kinds. Internet has evolved from a static medium to a fully interactive tool. Watching television, listening to the radio, playing games, banking, telephoning, learning,... are some of the possibilities of internet today. Some leaders of important computer companies like Larry Ellison (CEO at Oracle) even say there is no more future for desktop applications. They say if somebody needs an application, he or she can connect to the software providers server and load the application using the internet connection; *The world is your hard-disk!* Docucentered design is one of the factors influencing this evolution. But all this is still wishful thinking and recent developments and network connection speed are not really suited for these sort of applications *yet*.

The biggest users of internet components available today are probably the financial services and online shopping services (better known as e-commerce). E-commerce is a slowly growing business, which still needs improvement. Increasing security possibilities and ready-to-use software building blocks are making it more attractive, easy and safe to start an e-business. Entertainment and multimedia is also an important market for components; movie and audio plugins, online gaming, interactive educational presentations,...

### 1.3.3 How can it evolve?

It is difficult to predict how software development will be in the next decennia, particular software development for the internet. A few things already can be said for sure: we will have increasing network speeds and more bandwidth. The internet services will evolve towards a broader range of possibilities. Maybe it sounds strange, but the internet nowadays does not offer that much to the users, it could do much better. Although the many benefits and the research done concerning distributed systems, they are still poor in use. There are many possibilities which are only used by a small part of the users, or still in research stadium. Components are already broadly used, but everybody seems to have them on their own PC instead of using them online. When the connection speeds are sufficient, it will be a lot easier using different components online. The user does not have to bother any more if he or she has the right version on disk, or whether some component is missing for the data to be used etc...Besides the evolution of connection speeds and fast security services, the user also will have to think different about its use of the internet. It will not only be as an information pool but as an extension to its desktop; a pool of shared resources. This will probably take some years and some (technical) changes.

# Chapter 2

# Underlying techniques for building components

This chapter discusses some available component models, focusing on their particular capabilities to function in distributed (networked) environments and their security enhancements. There are probably more available component techniques then discussed here, but it is not the goal of this thesis to cover them all. The following component models will be discussed shortly in their general properties and a more in depth investigation concerning distribution and security will follow. The Microsoft Component Object Model (COM) and its later extensions DCOM[1] and COM+, the OMG Component Object Request Broker Architecture and finally JavaBeans, Enterprise JavaBeans and RMI.

## 2.1 The Microsoft Component Object Model

### 2.1.1 COM, DCOM, COM+ and MTS

**COM** COM is the Microsoft Component Object Model. With this technology Microsoft tried to meet the growing demands for component based software development. COM is defined as a binary standard and at first was also announced to be a network standard. A COM component has one restriction: it has to inherit the `IUnknown` interface. In the upcoming subsection we will compare COM with the general component model proposed in section 1.2. This will not be done within details, the purpose of the description is to lay the foundation to explain *internet-able* components like ActiveX, which are build on COM.

**DCOM** DCOM is just COM with a longer wire; DCOM extends interactions between components across networks. DCOM provides enhancements as location transparency across network locations. It is based on DCE-RPC[2] for transport and security mechanisms. DCOM is integrated in the Microsoft Windows system since Windows NT 4.0 and is available for free download. It has the advantage of many available tools for configuring and launching the DCOM Component

[1] also known as "COM with a wire". DCOM is an extension for COM especially for interactions across distributed components.

[2] DCE-RPC: Distributed Computing Environment Remote Procedure Calling.

> Remote Procedure Calling integrates with conventional procedural programming languages in a convenient manner, enabling clients to communicate with servers by calling procedures in a similar way to the conventional use of procedure calls in high-level languages.

See also [Geo94] chapter 5.

or CoClass[3]. For example MTS is built on DCOM for communication with resource managers (SQL servers and other MTS servers) [Ant97].

**COM+** COM+ is an extension of COM upgraded with MTS services extensions and new services. The remote architecture does not change and stays the same as with DCOM, but COM+ is restricted to in-process objects. COM+ primary target was easier deployable and extensible components. COM+ provides default implementation for the `IUnknown` interface, `IDispatch` interface and class factory by using the COM+ runtime environment. For events and packaging there are also defaults available, this eases the work of the developer. Besides this deploying options like remote installation of components and registering are also taken into account [Mar97]. COM+ includes so many functionality it becomes immense to master. Microsoft realised this apparently and lightened the task of the programmer by supplying default implementations, which are depicted in table 2.1 indicated by a $\surd$ mark. Summarising, COM+ goals are [Mar97]:

- make COM programming easier;
- solve problems discovered with COM-based application development and deployment;
- extend COM with new services for developers (implementation inheritance, real garbage collection, access security,...);
- make a extensible component model;

| PROPERTY | COM COMPONENT | COM+ COMPONENT |
|---|---|---|
| Class Factory | | $\surd$ |
| DLL Register | | $\surd$ |
| Reference Counting | | $\surd$ |
| Query Interface | | $\surd$ |
| IDispatch | | $\surd$ |
| Connection Points | | $\surd$ |
| MetaData | | |

Table 2.1: COM and COM+ default implementations

**MTS** The Microsoft Transaction Server includes among others the following features [Mic99]:

- A three-tier application model;
- ActiveX support;
- Thread and process management;
- Object instance management;
- Component management;

[3]Instead of COM or DCOM Component the term `CoClass` is sometimes used in documentation about COM or DCOM. In this thesis the term `CoClass` will be used when discussing implementation specific features.

- Shared resource management;
- Transaction management, maintaining the ACID[4];

The MTS lets us develop distributed, scalable, component based and deployable applications enhanced with a transaction processing monitor, which supports online transaction processing for components based on COM. Through its services MTS offers a framework for server-side components. It is actually the required extension for DCOM to be called a real Distributed Component Model. The MTS is considered as *middleware*[5].

### 2.1.2 Object oriented inheritage

COM defines a binary standard to define the interface. The *blackbox* principle is very well supported here. Microsoft likes to compare it with an Integrated Circuit in their specification (if one can call it a specification). We can send and receive signals to the IC but we can not see how it processes the signals. Only the manual of the IC tells us what effects are reached and which preconditions must be fulfilled for this. The determination of these preconditions and postconditions is also called a *software contract* for a component (see section 1.2.1). COM supports most of the object oriented principles like data encapsulation, instantiation and finalisation. All COM classes must inherit from `IUnknown` which defines three methods in its interface:

1. `QueryInterface` Informing the user about the supported interfaces.
2. `AddRef` Increases reference counter, for garbage collection.
3. `Release` Decreases reference counter, for garbage collection.

Instantiation is handled somewhat different then in normal programming languages. There is a kind of factory available that produces an object if requested. If the user wants to create a new instance of a class he or she must know the GUID[6] or the ProgID of the CoClass . An API-function named `CoCreateInstance` with the GUID passed as argument creates an instance. Multiple instances of the same class can be made with a CoCLass's ClassFactory. Destruction of CoClasses is not done manually but with a garbage collection algorithm called *reference counting*. A CoClasss supports two functions `Addref` and `Release` for garbage collection. For every pointer that points to the particular CoClass this CoClass will call `Addref` and the reference counter increases by one, if a pointer to the CoClass is removed this CoClass will call `Release` and the counter decreases. This is a very simple garbage collection algorithm with a linear complexity (because it is done throughout the whole execution time) and it can not handle circular structures like shown in figure 2.1 and figure 2.2 [Pau92]. COM+ solves the problem with circular structures and still uses reference counting, a disadvantage is the unpredictable lifetime of CoClasses [Mar97].

[4]ACID stands for Atomicity, Consistency, Isolation and Durability. These are four important properties for transactions

[5]In a three tier system architecture (see section 2.3) the *middleware* is the glue to connect the middle tier. It offers connection or integration points to both the client tier and the data tier.

[6]Globally unique identifier, an identifier composed out of the current date and time, an incremental counter, a random number generator and a machine identifier. It is assumed to be unique in the whole world, but no guarantee is given.

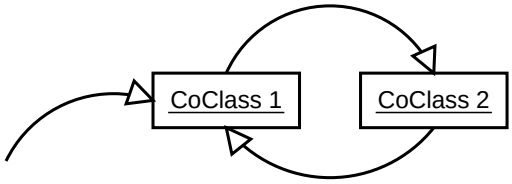


Figure 2.1: Pointer from memory to structure

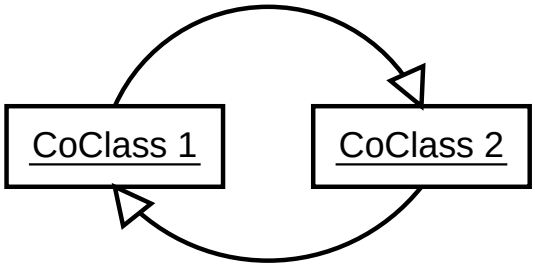


Figure 2.2: unable to clean up objects not referenced from memory

Unfortunately, COM does not support multiple inheritance as it is found by most OOP languages like C++. The Microsoft team states inheritance is a threat to real encapsulation. According to Microsoft there is no need for actual inheritance in CBSD. The creators of the COM saw no need to support real implementation inheritance[7] because it can create many problems in a distributed, evolving object system. Inheritance would not be suitable for creating components because the reuse of each others implementation without knowing the internal structure of this implementation is a difficult and errorprone job. That is an unlikely valid reason to omit it, it is far more likely that by omitting implementation inheritance many problems with COM itself are avoided, like GUID creation and more complex polymorphismtic functionality. Like stated in the general model a complete contract for such a component, containing preconditions, postconditions, invariants and exceptions would help overcome this problem. If Microsoft had begun with a specification of the model before implementing it (they did it the other way around) implementation inheritance would probably be supported. However, they have defined workarounds of two mutual exclusive situations using aggregation or containment, these are described in [KK98].

### 2.1.3 Resource, version and data management

COM has begun its life not as a specification but as an implementation. This has brought with it a lot of drawbacks concerning consistency and uncertainty in particular matters. This is most of all reflected in the documentation describing COM and related technologies, a lot of sources say different things about the technology which is very confusing. But by entering the market with an implementation instead of a specification first, more companies will have direct

[7] two different sorts of inheritance are:

**implementation inheritance** subclassing, inheritance of implementation fragments;

**interface inheritance** sub-typing, inheritance of contract fragments;

(see [Cle97] chapter 7)

access to the technology and are freed from implementing it themselves. This means sooner deployment of their products which is an important issue for a highly competitive market such as for internet products. COM is a widely used technology because of three reasons:

- Microsoft Windows is the most widespread class of Operating System on the market today;
- Microsoft heavily supports COM by integrating it in the basic functionality of their systems and allowing developers to build COM components to be fit in nice and easy accordingly;
- COM gives developers hands-on solutions without them having to bother with incomplete implemented specifications or the need to implement the component model themselves.

The former three reasons prove COM is a successful commercial product (if we may call it a product), and because of it widespread use, items as Resource, data and version management can not be left out the model (or better, the implementation). Knowing this, COM is supposed to have a pretty good support for these items. We will look if this is as stated before.

When describing the general component model in section 1.2, version management is described as one of the consequences of incremental development, practically important when concerning distributed applications. This is not different with COM components, and their makers realized the problems arising with different versions of the CoClasses. COM uses *immutable interfaces* to enhance version management. This means a new version of an existing CoClass will have a new UUID[8] and it is not possible to add new functionality to old interfaces. COM provides a versioning system that allows seamless evolution of components [Pau95]. There are two important items in the COM model for versioning and backward compatibility:

1. immutable interfaces : No new functionality can be added to older interfaces. New functionality should be exported by adding a new interface. Notice in COM interfaces itself are *not* subject to a particular version. It is a completely new interface carrying a new identifier.
2. `Iunknown::QueryInterface`: This method is a sort of introspection that informs the client of the functionality of the components (at runtime). Clients interested in using a totally new interface can use this method.

This is all very clean, but now the problems arise. Without a decent specification of the component a developer can not be sure he/she supports the old interface if they do not have access to the code of this old interface. Although most documentation claim supporting the old interface is enough to ensure backwards compatibility, this is not enough in reality. If a seamless integration or upgrade is desired the developer must also respect the pre- and postconditions as they are defined with the old interface and probably also invariants and exceptions.

Unlike version management, transaction management is not part of COM or DCOM. Microsoft decided to add this functionality apart from the actual component model in an extensive way. It is known as the Microsoft Transaction Server (see section 2.1.1) which is comparable with the Enterprise JavaBeans Server (see section 3.3.3) or CORBA's Transaction Service. The MTS will handle calls to objects or components when they are registered with the MTS service as being transactional. Besides this transaction management functionality the MTS is also used for

[8] an UUID is the same as a GUID. GUID is the Microsoft implementation of the Open Software Foundation's Distributed computing environment (DCE) universally unique identifier (UUID). [Sar94, Mic99]

| |
|---|
| 1)**none**<br>No security, raw RPC |
| 2)**connect**<br>Authentication during connection |
| 3)**call**<br>Authentication for each procedure call |
| 4)**integrity**<br>Authenticate and verify the request packets, they must not be altered during transport |
| 5)**privacy**<br>Perform security level 1,2,3 and 4 and encrypt the packets |

Table 2.2: Secure RPC levels of security

lifetime-cycle management and remote object or component invocation. This means it takes over some of the functionality of DCOM and extends it towards in favor of database management systems functionality. Just like CORBA ORBs using the General Inter-Orb Protocol (see section 2.2.5) the MTS can communicate with other MTS's located on other machines. Summarized MTS offers a *context* for easy lifetime cycle management and transaction management while extending distributed capabilities.

### 2.1.4 Security

COM provides security on several crucial levels:

- It uses the Operating System permissions to determine whether a client is enabled to start some code (a particular class of object).
- It uses the Operating System or application permissions to determine if a client is enabled to load the supervised object, and whether the user has read-only or read-write access.
- COM is based upon DCE RPC and inherits its security architecture that comes with it. This means it provides an industry-standard communications mechanism that includes fully authenticated sessions. Also cross-process and cross-network object servers with standard security information about the client are supported.

We extend our discussion to DCOM here, so we can investigate the security issues with remote connections. DCOM security is, just like COM, based on RPC and the Operating System. The RPC being used is also called *authenticated* or *secure* RPC, created by Microsoft. We know RPC can work with TCP/IP, IPX and named pipes, but only the named pipes can provide authentication support. Microsoft created the *Win32 Security Support Provider Interface*, which is a framework for security providers to plug in. The default security provider at the moment is the NTLM security provider which is clearly a very incomplete security provider (see Appendix A). When replaced by a Kerberos protocol, it should be much safer. The *Secure* RPC specification has 5 levels of security, presented in figure 2.2.

In the previous paragraph, it is obvious that COM fully relies on Microsoft techniques for ensuring security (authentication); besides the *secure* RPC the security also depends on the Operating Systems. Microsoft Windows NT and Windows 9x have different security mechanisms, and this will be reflected in the available security enhancements for COM on those specific platforms. This implies that COM depends a great deal on the OS it is deployed to, and porting

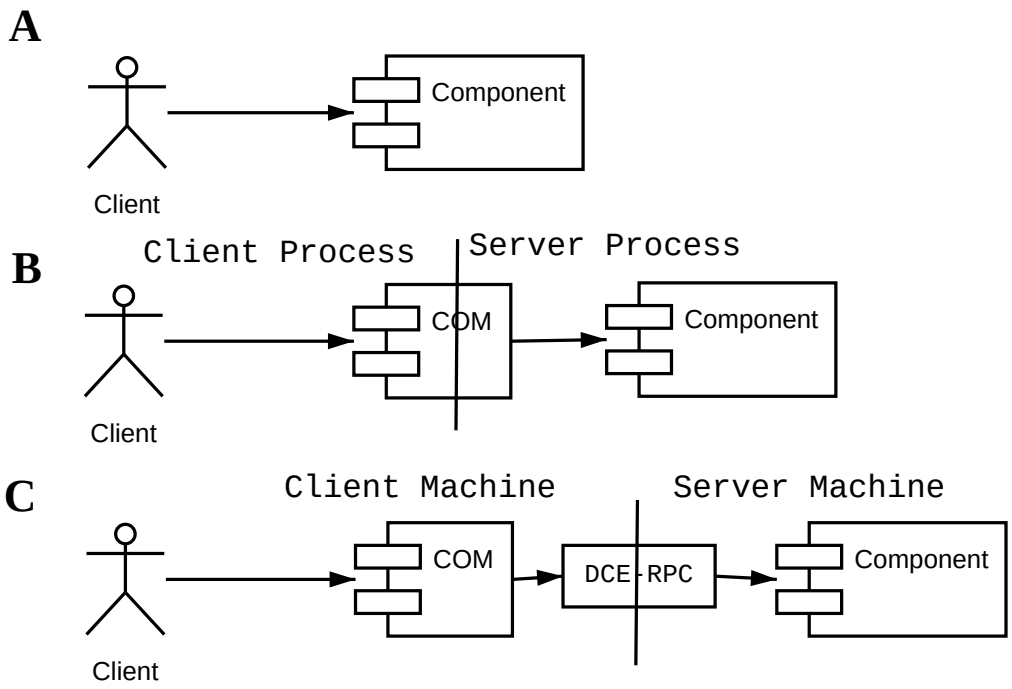


Figure 2.3: Accessing COM services

COM to other OS's is no fun to do, certainly not if you want to preserve the same security functionality. With the introduction of COM+ an extended security mechanism for the COM was introduced. COM+ automates a lot of security issues which would have been done by hand if just COM was used.

### 2.1.5 Transparency

Figures 2.3 and 2.4 shows different ways of accessing a COM service. In figure 2.3 **A** shows the normal way of directly referring to a COM service (better known as a local *in-process* call), where it is being referred in the same address space as the process who is the referee. **B** shows an *inter-process* call (also local, but the COM service resides in another address space) and **C** shows the COM service if it is located on another machine, which means a remote call. The remote calls rely on RPC, like mentioned in section 2.1.4. Location transparency COM offers is not that fantastic; it stays local pretty much. Although COM also use DCE RPC the best it can do is **local** inter-process location transparency, with a little help of a software component manager [Mic99]. For these kind of calls both a **Proxy** (client side) and a **Stub** (server side) component are required. Because RPC is used, there is marshalling[9] involved in parameter passing, and COM offers three different ways for doing this (automatic, standard or custom see [KK98]).

It is a lot better with DCOM, because it has network capabilities which is a requirement for good internet components. DCOM objects requires registration into the local Windows registry. When at the client-side there is a `CoCreateInstance()` call, the local registry can point out to another machine where the DCOM object must be instantiated. There is a pitfall here: what microsoft calls location transparency is not location transparency like we have defined in section

[9]Marshalling is the process of taking data items and transforming them into a form suitable for transmitting. Unmarshallling is the process of disassembling the data on arrival and transforming it back into the original data items.[Geo94]

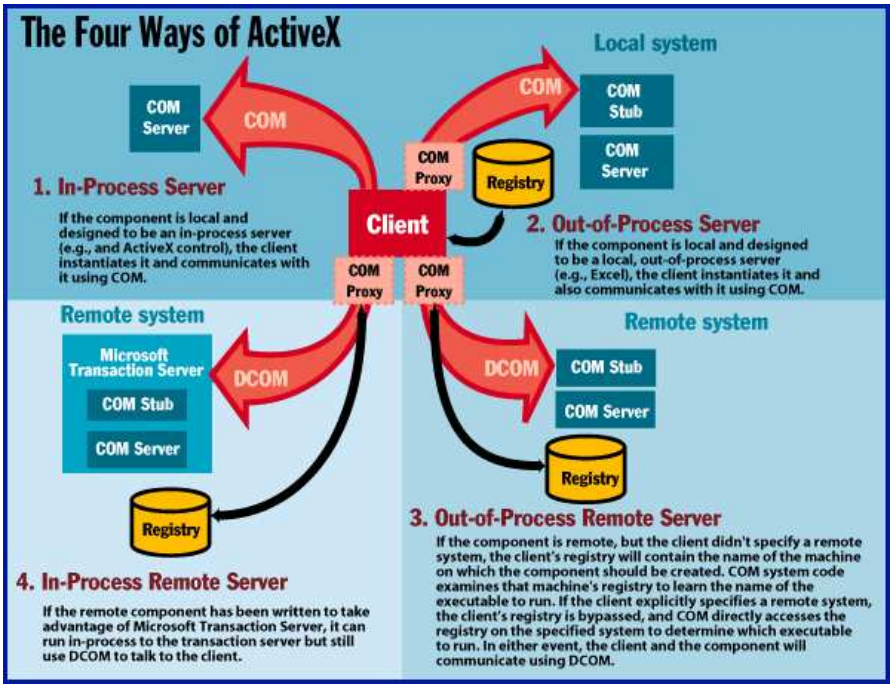


Figure 2.4: COM and ActiveX

1.2.3: the user should not have to know about the location of the remote component. With DCOM a remote component must be located by the user (i.e. By a pointer in the registry), only one remote machine can be defined as server (but different local), and the user must change the reference as the component moves from one machine to another. It is fairly complex to build a distributed system with DCOM [KK98].

## 2.2 Common Object Request Broker Architecture

### 2.2.1 What is CORBA?

The Common Object Request Broker Architecture (CORBA) is a specification designed by an industry consortium, the OMG. It was meant to *integrate diverse applications within distributed heterogeneous environments*[Ste97]. For an excellent introduction into what CORBA stands for reading [Ste97] from Steve Vinoski is a good advise. CORBA is actually a very important part of the Object Management Architecture (OMA). The OMA defines two things: an *Object Model* and a *Reference Model*. The former involves the OMG Interface Definition Language (IDL) for defining interfaces, while the latter is responsible for the Object Request Broker (ORB), which will be explained later on. Describing everything the OMA specifies implies writing a few books, so this introduction is very brief in relation to the existing specifications. CORBA actually tries to tackle everything a real component model must include from within its specification; object oriented concepts, blackbox principle, polymorphism, exception handling, composition, introspection, persistence, versioning, transaction management, portability, transparency, security, GUI, customizability and lots more.

### 2.2.2 Object oriented inheritage

OMG only makes the specification, not an implementation. This is why they must have a way to describe objects available for use in their system without the need to use implementation specific behavior Therefore the OMG introduced IDL for describing the *interface* of an object. The IDL is a very complete language for describing such interfaces based on the ANSI C++ lexical rules and pre-processing [Joh96]. It is strongly typed, but has a type `any` which can hold any OMG IDL type, so if necessary most restrictions of the strongly typed interface can be overruled. The IDL can describe attributes, operations, exceptions, multiple inheritance, references and most other things we are used to using in OOP languages like C++ or Java. Somewhat confusing, but very useful in big projects for example, is one can define a module in the IDL containing several interfaces for grouping them together. Simple, educative examples of an IDL descriptions are shown in figure 2.5. A consequence of this approach is a good support of the blackbox-principle and it stimulates the use of contractual programming. Here the interface definition is the actual contract.

```
module TicketMachine{
        interface FirstClassTicket;
        interface SecondClassTicket;
};
```

```
interface AutoMobile{
        struct CarProperties{
                long speed;
                float oil;
                long serialnr;
        };

        void accelerate(in int accspeed) raises (CarNotStarted);
        long getSpeed();
        void brake();
        float getOil();
        void getCarData(out long serialnr, out float oil,
                                        out long speed);
};
```

Figure 2.5: OMG IDL examples

### 2.2.3 Resource, version and data management

The central key to the management of the different objects available in the system is the ORB. One could see it as a service; a client can request an object to it or store a new object (see figure 2.6). It is comparable with a central database, distributed over the (networked) system, offering objects and registering for objects. CORBA has several services[10] making a programmers life easier described in [Obj98] and summarized in [Jan99]. Particularly interesting is the Object

[10]Naming service, Event service, Persistent Object Service, Life Cycle Service, Concurrency Control service, Externalization service, Relationship service, Transaction service, Query service, Licensing service, Property service, Time service, Security service, Trading Object service

Transaction Service (OTS) which manages transactions, but has poor support for distributed (nested) transactions as described in [Geo94]. This is surprising because CORBA *is* made for distributed systems.

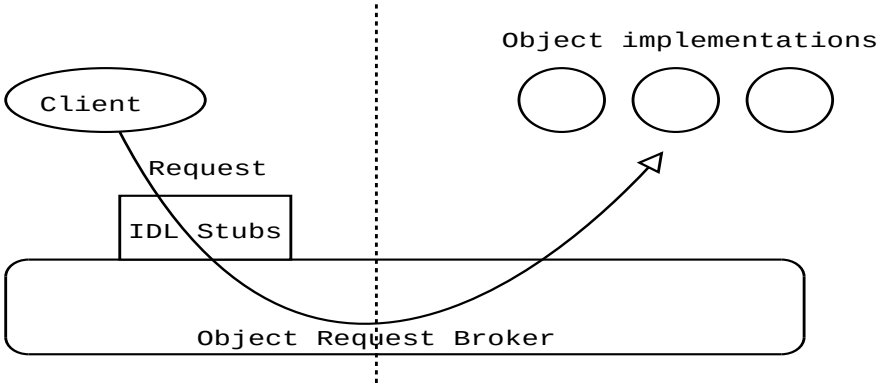


Figure 2.6: A client making a request to an ORB

The Interface Repository is described as "...the component of the ORB that provides persistent storage of interface definitions it manages and provides access to a collection of object definitions specified in OMG IDL." in [Obj99]. This repository is among others concerned with names, identifiers and types. The repository is also important for version management, [Obj99] says: "The new versions will have distinct repository IDs and be completely different types as far as the repository and the ORBs are concerned. The IR provides storage for version identifiers for named types, but does not specify any additional versioning mechanism or semantics." A lot of responsibility concerning version management is delegated to the programmer.

### 2.2.4 Security

It is difficult to define an exact and good security model with only a specification. The security model needs to be conform with with the different programming language bindings which exist for the CORBA IDL and the different OS's where an ORB is available. An important threat is the replacement of a component by a malicious component, offering the same interface (so the client does not notice the change), but doing some nasty thing in the implementation. By consequence authentication of components and impersonation are important issues and will be discussed in greater detail in section 4.3.3.

### 2.2.5 Transparency

The CORBA is clearly the first real distributed component system we encounter in our discussion. Using the ORB it can offer system-wide location transparency and access transparency ( = network transparency), and the other sorts of transparency defined in section 1.2.3 are also pretty good supported through the ORB, except for some programming language dependent kinds of transparency like failure transparency. CORBA and transparency were meant for each other, so it seems when investigating the architecture OMG designed for constructing new objects and

invoke methods on them. The first service which enhances transparency is the *Naming Service*; it returns an object location in exchange for the name of the object. The Naming Service can even return locations based on *externally visible characteristics* of objects such as the last time it was changed. [Obj98] indicates the role of the Naming Service for transparency: "Naming service clients need not be aware of the physical site of name servers in a distributed environment, or which server interprets what portion of a compound name, or of the way that servers are implemented" and "Existing name and directory services employed in different network computing environments can be transparently encapsulated using naming contexts." [11]. The last quotation says the naming service can function in an environment where different ORBs are interconnected.

Two ORBs can communicate with each other using a bridge, either in a immediate[12] or an mediate[13] fashion [Obj99, Joh96]. Objects can be used throughout the different ORBs using the *Interoperable Object Reference*. This requires the following data [Joh96]:

- the object type
- the protocols of the invoked ORBs
- the available ORB services; e.g. transaction and security services must be negotiated.
- the chance that the reference is `null` and avoiding unnecessary work this way, there is no use searching or transporting `null`-values.

Figure 2.7 gives us an overview of the interoperability possibilities.

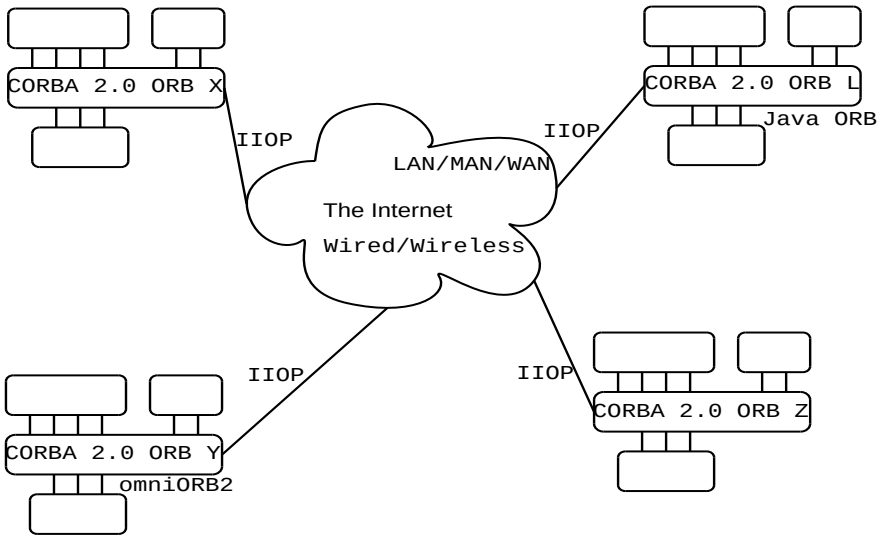


Figure 2.7: CORBA ORB Inter-operability

[11] a naming context is an object that contains a set of name bindings in which each name is unique. To resolve a name is to determine the object associated with the name in a given context.

[12] two ORBs speaking over a single bridge translating from one into the others language directly. Interconnecting $n$ different ORBs needs $\frac{(n^2-n)}{2}$ different bridges

[13] two ORBs have each a bridge connected to a general domain speaking to each other in the language of the common domain. Interconnecting $n$ different ORBs needs *n-1* bridges (assuming the common domain is one of the ORBs languages)

In the scope of this thesis it is particularly interesting how CORBA communicates over the internet. It is possible to connect two ORBs using a TCP/IP connection, and still offer location transparency and other services, as if you were using a single ORB. For this sort of communication between ORBs, OMG has describes a general protocol on which other more specific protocols should be build: the General Inter-ORB Protocol . Figure 2.8 shows how the OMG designed their extensibility for different Inter-ORB Protocols [Obj99, Joh96]. For communication between ORBs using a TCP/IP connection the Internet Inter-ORB Protocol (IIOP) was designed as a child of the GIOP. Actually, the IIOP is like the "IDL language mapping" for the GIOP. By allowing two ORBs to communicate over a TCP/IP connection, they are enabled to co-operate over the internet. IIOP is a mandatory service for the OMG ORB and can be used as the common domain protocol in case of mediate bridging.

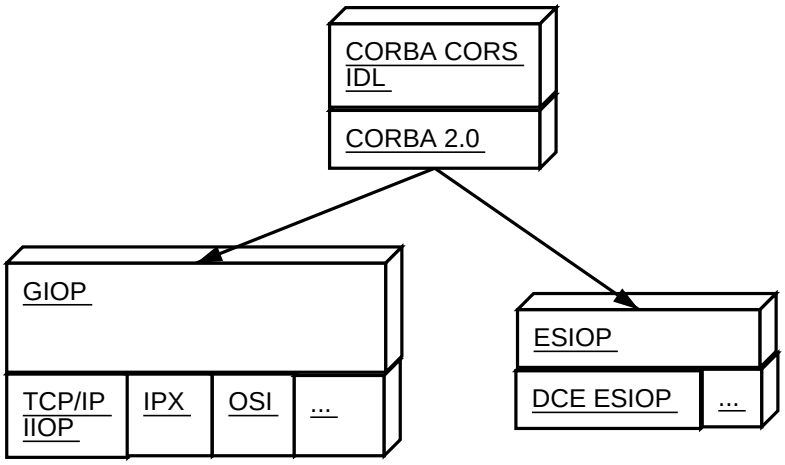


Figure 2.8: The CORBA interoperability specification structure

## 2.3 Java, Java Beans, Enterprise Java Beans and RMI

Java is an Object Oriented programming language, syntactically based on C++, made by Sun. Originally it was meant to serve as a development platform for embedded systems (the language called *Oak* back then). It is an interpreted language to a certain level; Java programs are compiled into bytecode and this bytecode is executed with a virtual machine. By providing such a virtual machine for different platforms Sun claims to support platform independence, thereby Sun releases specifications for their virtual machine so anyone is free to implement their own accordingly. This *platform independence* is a topic for discussion because only the most wide-spread operating systems like Microsoft Windows, Linux and Solaris have up-to-date virtual machines. For convenience we will assume platform independence is a realized objective in this matter. Assuming this we see a first point of importance for internet. Instead of booming in the embedded systems area several internet browser makers began to add a virtual machine in their browser and Java grew out to be an important internet language. Because its possibilities are comparable with the possibilities one has when working with C++, Smalltalk or Eiffel it added great power in making the internet interactive.

The component model based on Java Sun has introduced is known as Java Beans and is defined as follows [Ham97]:

> A Java Bean is a reusable software component that can be manipulated visually in a builder tool.

and must be refined further to exactly describe what a Bean is. During this section we will question if the JavaBeans model is "worthy" to be called a component model, because it lacks many basic functionality a real component model should have. For one it is totally Java dependent[14]. We will consider the server-side component model Java developed, namely *Enterprise JavaBeans* (EJB). These EJB are designed especially for enterprise applications and simplify the job of the developer writing the so-called server-sided components [Bru00]. Being designed especially for enterprises, they support several concepts used in enterprise software development like multi-tiered[15] applications, transactions, security and much more. It is best to avoid confusion between EJB and the normal JavaBeans by stating that EJB actually defines a special model for developing components or even services for the server-side and can use JavaBeans in doing so, but EJB provides the server-sided framework. Searching for similarities between JavaBeans and EJB other then being specification for a component model, are useless; they are totally different. While the former is mostly used in application development tools, the other defines server-side, enterprise-aware services for deployment [lab99, Vla99].

### 2.3.1 Object oriented inheritage

The JavaBeans component model is probably the most simple one a developer can imagine. There are two **guidelines** one must follow writing a Bean:

**Properties** decide which properties of the Bean are accessible/configurable by implementing `getXxx` and `setXxx` methods for each of these properties with the appropriate access identifiers.

**Serializable** to enhance easy serializabilty for preserving the state of a Bean, the Bean must implement the `Serializable` interface.

Because of these simple guidelines for creating Beans, everything what the Java language offers as object oriented concepts is also available for Bean components. In fact, Sun defined most of their visual framework, known as *Swing*, a subset of the Java Foundation Classes, following the JavaBean model. This enables easy visual manipulation of Java UI building components in visual builder tools.

Because the JavaBeans model is so close to Java, it supports the same object instantiation, polymorphism, inheritance, exception handling, aggregation and other object oriented properties, like described in [Jam96]. So also garbage collection instead of an explicit object destructor is provided for JavaBeans. But JavaBeans are distinguished by some features most normal Java classes do not have [Ham97]:

**introspection** exposes the Beans public methods at design- or runtime, see also section 1.2.4.

[14] There is a work-around available, Java Beans can be used with a COM-wrapper enabling it to support better interaction with native programming language environments supporting COM (see 3.2.4).

[15] **Two-tier**:the client tier consists of "fat" clients, which contain the business logic for the application architecture;
**Three-tier**: the client tier consists of only "thin" clients that contain logic to call the business logic appropriately. Three-tier applications helps the scalability problem out of the client and onto a separate tier;
**Multi-tier**: there are as many tiers as needed in the system to separate the necessary services [Mai98].

**customization** enhances the application builder tool with design-time customization of the appearance and behavior of a Bean, see also section 1.2.4.

**events** support for events takes care of connecting different Beans to each other.

**properties** like normal classes, but important for customizations.

**persistence** so the customized state can be saved and restored by the application builder tool.

With these features, JavaBeans approaches the blackbox principles described in section 1.2.1.

### 2.3.2 Resource, version and data management

In the JavaBeans API specification [Ham97] versioning is not mentioned once as an available technique in the model. The developer must program the version management by hand, for example using JARs[16] containing information to ensure a newer version replaces an older one. JAR files can hold data about the files it contains like the packaging date and vendor name[Lis99]. The Beans specification itself does not address revision management (version management) and dynamic replacement. The specification ignores more important issues for component models then version management alone. In doing so the Java Bean model starts behaving like a Java-only component model not meant for serious applications. It is obvious this model focusses mainly on assembling User Interfaces, what does not mean there are no possibilities for using Beans outside visual assemblage.

Sun compensated the original JavaBeans approach by introducing the EJB for server-side purposes which are suitable for the more serious internet enabled applications nowadays. It exists out of an EJB server managing EJB containers and EJB objects. The EJB model includes transaction, resource and security management, which are important for server-side components. EJB is a *middleware* component model [Vla99]. Just like the OMG with CORBA they did not offer an implementation but a specification for this model. By the time Sun began developing their specification, they had an example available; the Microsoft Transaction Server. In the next chapter it will become obvious Sun actually used MTS as an example for the specification and the EJB model will be further explained.

### 2.3.3 Security

Since JavaBeans are written in Java, they have to comply to the security model of the Virtual Machine they are using. Much has changed since JDK1.0 , the first release of the Java Development Kit. This release supported a security model known as the *Sandbox Model* depicted in figure 2.9 and described in [JSF96]. This way, an applet or remote application could access no resources at all, and a local application can have full access. An Applet can do whatever it wants *in* its sandbox, but has no rights at all outside the sandbox. Java2 (the name for the Java version shipped with JDK 1.2) has replaced this with a fine-grained security systems where access to resources can be controlled individually according. A user-specific policy file together with a system global policy file are loaded into a *security manager* and this security manager manages the permissions of the application executing in the JVM, more particular, to the execution space of the user associated with the policy file. This way an applet from a specific

[16] A JAR file is a ZIP format Java archive file that may contain a manifest file with additional information describing the contents of the JAR file. JAR files are used to package class files, serialized objects, images, help files and similar resource files.[Ham97]

origin can be considered trustworthy and gain access to system resources by authentication. For example, it is possible to allow an applet from a certain source to write something in a specified directory on your hard-disk.

We could say Java is a pretty secure language, but the virtual machine implementation can be a dangerous security leak. For example, Microsoft Internet Explorer and Netscape Navigator both use their very own virtual machine. Due to implementation bugs serious holes in the security are possible (it happened before!). For serious (critical) security management, the only trustworthy way is depending on a formal prove of the virtual machine. [Dan98] describes the Java Runtime Stack Inspection as used in the Netscape JVM on a formal basis, using few axioms to obtain a formal model to check access control. It uses a *belief logic* known as ABPL-logic as described in [M. 93], and starts with four primitive functions to check each frame on its privileges:

1. `enablePrivilege()`
2. `disablePrivilege()`
3. `checkPrivilege()`
4. `revertPrivilege()`

In order to access a resource $R$ the system checks `(current stack frame).checkPrivilege(R)` before granting access to $R$. The only way to be sure your system is secure is to have a formal prove of its security correctness. Note this can also be said of a lot of other environments. A mechanism to describe authentication protocols for distributed systems on a formal basis that is used a lot in general is *BAN*-logic (Burrows-Abadi-Needham). This falls outside the scope of this thesis, more information can be found in [Geo94, M. 90].

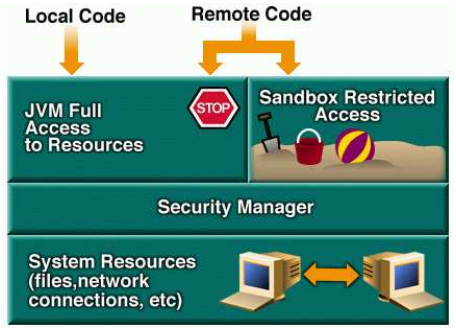


Figure 2.9: JDK 1.0 Security Model

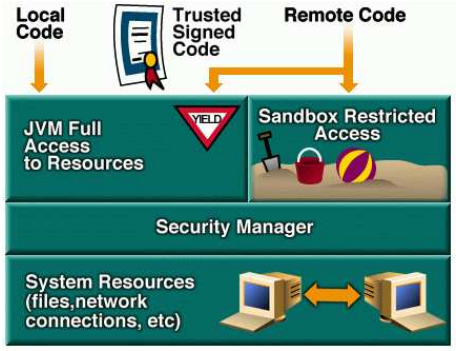


Figure 2.10: JDK 1.1 Security Model

### 2.3.4 Transparency

Pure Java does not offer transparency, the Java Virtual Machine (JVM) has to know where the classes are located to load them. Sun added a mechanism to use objects, located on other machines, and called it the *Remote Method Invocation*. This allows programmers to remotely use objects and use them if they were local objects. The possibility of treating a remote object as a local one, adds some access transparency to the language. However the location of the object must be partially known before one can make use of it; the object will be registered with the *rmiregistry*, a naming service. A client of the object must know the location of the rmiregistry,

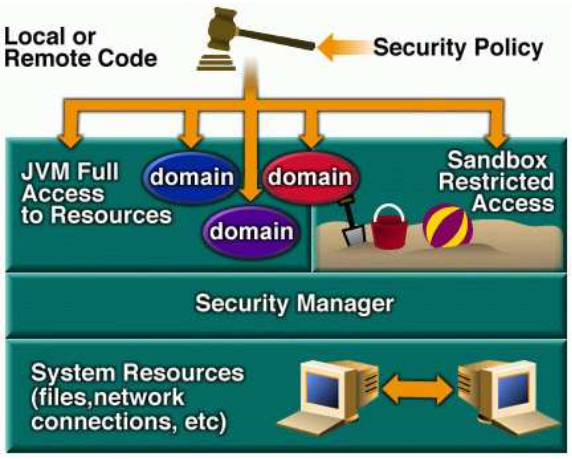


Figure 2.11: JDK 1.2 Security Model

which implies no real location transparency is obtained here. There is only location transparency at the lower level: it does not matter where the objects registered by the rmiregistry are located, as long as it is known where the rmiregistry is located the objects can be used remotely. The architecture of RMI is shown in figure 2.12. In order to make objects suitable for RMI, a server interface has to be written, together with the actual implementation. An example of a time service, giving current time is listed in figure 2.13 (the interface) and figure 2.14 (the implementation). When this is done the developer can use the `rmic` tool to generate the stub and skeleton for client and server.

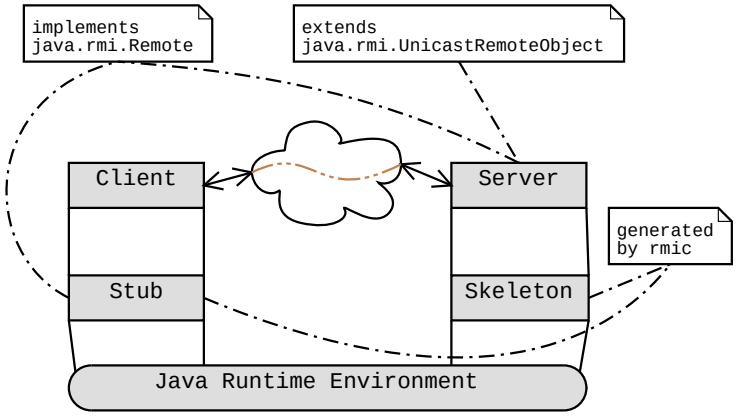


Figure 2.12: the RMI architecture

Since the introduction of JDK1.3 (better know as the *Java SDK Version 1.3*) there are two kinds of protocols RMI can use to communicate [Ses00]. The first one is the *Java Remote Method Protocol* and is a wire level, stream based protocol on top of TCP/IP. For the second RMI version, Sun and IBM have gathered forces and made *RMI-IIOP*; RMI using the Object Management Group Internet Inter-ORB Protocol. This is one of the many factors which makes clear Sun wants to push the Java programmers towards CORBA as an extension to Java. Latest facts of this evolution are the *CORBA Beans*, which means JavaBeans and a CORBA ORB are used together as a component system. This will be discussed further in section 3.2.3.

```
/**
 *  Time Service Interface for distributed object
 */
import java.rmi.*;
import java.util.Date;

public interface TimeServiceI extends Remote{

  static public final String TIME_SERVICE_NAME = "TimeService";

  /**
   * @returns the current time according to the server
   */
   public long getServerTimeStamp() throws RemoteException;
}
```

Figure 2.13: the server interface

```
/**
 * Time Service, a remote time service
 */
import java.rmi.*;
import java.rmi.server.*;

public class TimeService extends UnicastRemoteObject
                                  implements TimeServiceI{
  /**
   * gives back a timestamp coded in a long number
   * @returns long the actual timestamp
   */
  public long getServerTimeStamp() throws RemoteException{
    return System.currentTimeMillis();
  }

  /**
    * the constructor for the timeservice
    */
  public TimeService() throws RemoteException {
    System.out.println("new TimeService object available");
  }
}
```

Figure 2.14: rmi implementation

# Chapter 3

# Comparing components

## 3.1 Starting a battle

If we are comparing components on their capabilities for the internet, there are a couple important properties to highlight. *Transparency*, *security* and *portability* for example. For enterprise components transaction management will also be an important topic in evaluation. This comparison is not intended to be a mutual exclusive one. Most of the times different component techniques can be combined (using bridges or some intermediate technology), this way the best properties of the particular techniques can be deployed on places were they are of the most use. Notice the implicit complementary behavior which can follow considering it this way. Can we actually *compare* different component techniques? I think it is difficult to compare them to each other, because of there complementary behavior and the different backgrounds on which they are build. This would take us to a comparison of their fundamentals, which is not the goal. A comparison is most useful if we limit it to a certain property, and not the component as a whole. It is even more accurate if there is a general model (like defined in section 1.2) to which we can compare in an objective and neutral way. In the early days, a programmer had access to a limited resource of techniques and *adapted* the solution towards the used techniques. Nowadays, this has changed. The developer designs a solution and chooses the best tools and techniques to implement this solution, not changing the premised solution. This chapter focusses on, besides a general description, transparency for components, the following chapter will tackle security properties for components.

Which are the components we can compare? For one we can split up between the "normal" ones like ActiveX components and JavaBeans, and those especially for server-side services like MTS and EJB. This is one level, another level is to compare the component models like (D)COM and CORBA. For example, there is no use comparing JavaBeans and MTS, because there is no direct relation in the services they offer. There are already a lot of comparison written down, unfortunately not all are very specific comparisons. Most of the comparisons say the same things, and make subjective conclusions about which is the *best* component technique. For one, I am convinced there is *no best* component model as a whole, but there can be winners at specific domains. For a detailed comparison between DCOM and CORBA, see [P. 97]. A more general comparison between COM and CORBA, including a description of both, can be found in [Jan99]. A comparison between COM and a general component model like described in section 1.2 including a taxonomy of development environments and tools can be found in [KK98]. [Gop98] compares Java/RMI, CORBA and COM, but only on an introductory level.

The rest of this chapter will include a comparison between ActiveX and JavaBeans and server-side services EJB and MTS. Concentrating on distributed and portability properties, CORBA is included in our discussion where it is revealing.

## 3.2 ActiveX vs JavaBeans

### 3.2.1 ActiveX

ActiveX can be divided in different kinds [Mic99, Zan97, San96]:

**ActiveX controls** reusable software components designed to add specialized functionality to a Web site, desktop application or a development tool

**ActiveX documents** enabling browser to use non-HTML documents

**ActiveX Scripting** enables ActiveX to use scripting languages such as JavaScript or VBScript.

**ActiveX Server Components** enables the Web Server to interface to server software components using techniques such as ISAPI and CGI

Even with the separation of ActiveX in different kinds, it stays a fuzzy definition. This has its reasons: ActiveX is a name the "marketeers" of Microsoft came up with after the OLE2 hype. The first version of OLE was OLE1 and was based on Microsoft Windows 3.0 DDE. DDE was primarily based on communication using the Windows event model [Cha96]. It was an enhancement for embedding one application in another or even link applications to a document. The first steps of Microsoft into the docucentered world. OLE1 was to massive and was split up into OLE2 and COM, the former used no longer DDE for data exchange but was built on the latter, while the latter defined a multifunctional component object model. OLE2 was kind of an extension to OLE1, but with smarter fundamentals, it offered *smart* documents, structured storage (persistence), prebuilt controls, clipboard data transfer and OLE automation (specialized macros). At first, ActiveX was just another nice name for OLE2 with an extension to internet functionality, nowadays I do not think they could give a proper definition for it at Microsoft. Programmers consider ActiveX primarily as a *glue* for combining different techniques, like stated in section 1.2.2. At Microsoft they call it

> A marketing name for a set of technologies and services, all based on COM (the model, the "ORB", and the services).

This way ActiveX is a packaging technology for COM *and* built on COM. Users (non-programmers) are more likely to think of ActiveX as an ActiveX Control. ActiveX is the Microsoft way of bringing CBSD closer to the web.

The ActiveX Control definition has also the closest match with the JavaBeans definition (see section 2.3), so ActiveX and JavaBeans are candidates for a comparison. They are most of all visible components (including GUI functionality) and closer to an intuitive definition of a component, unlike the other kinds. The coming discussion will focus on them for our further investigation. ActiveX Scripting and ActiveX Server Components actually are not divisible: ActiveX can communicate with server components through the use of scripting languages like JavaScript , JScript or VBScript. The beauty of ActiveX scripting one finds in the possibility to define your own scripting language for special purposes. At the time, the Common Gateway Interface (CGI) and Internet Server Application Programming Interface (ISAPI) are the two most

widely used scripting methods for server interaction, where ISAPI is an extension and improvement of CGI. ISAPI can work directly with OLE and can handle more complex functionality, CGI does not provide such functionality. CGI transmits data by using environment variables, ISAPI through an OLE-interface enabling server-side ActiveX this way. A consequence is that the ISAPI can run in the server-process, and CGI only outside the server-process, so CGI needs to start a new thread for every request and ISAPI can use an existing thread to delegate a new request [San96]. ActiveX documents are a consequence of the "OLE past" of ActiveX, like described in the previous paragraph, a document can be embedded in another document without forcing the user to manually start the particular application responsible for that embedded document. E.g. a pdf-file opens in Microsoft Internet Explorer (IE), because IE knows the document type and starts the responsible container for the type (in this case Acrobat Reader). The viewer will work inside IE because it uses a container with a COM interface enabling uniform communication with the "parent" application.

### 3.2.2 JavaBeans

JavaBeans, the component model Sun introduced for Java, is not as successful as ActiveX is for Microsoft. We have already introduced the JavaBeans model in 2.3, so we can limit the discussion here to properties important for comparing them. Sun designed the JavaBeans model with visual manipulation of GUI building blocks in builder tools in mind, but has recognized a wider and more diverse range of use for them. The last couple of years, JavaBeans are extended with the JavaBeans Activation Framework (JAF)[Bar99], which enables *pluggable* components as a prime service. The JAF services include

1. determination of *arbitrary* data types;
2. encapsulation of data *access*;
3. automatic *discovery* of operations available for certain data types;

These services together enhance the extension of existing applications with new plug-ins (e.g. A spell-checker added to a text processor separately). Besides the JAF, Sun has come up with the Drag-and-Drop (DND) specification [Lau98a], which allows java programmers to use OS independent Drag-and-Drop interaction, and integrate with the platform Drag-and-Drop if desired. The DND specification [Lau98a] includes

- uniform data transfer mechanism;
- continuos event-handling (source determination, dropping, dragging over, dragging under,...);
- AWT drag-and-drop;
- possible extensions to input or output devices;

Although this is a GUI feature, it also extends the capability of working with uniform data-transfer methods between different applications. By consequence a *clipboard* can be used just as the clipboard in Microsoft Windows. This uniform data-transfer is enforced by *DataFlavors*; a mechanism for converting native data type to Java MIME types, even across the JVM boundaries. This mechanism is defined in the JAF specification [Bar99] and in the Drag-and-Drop specification [Lau98a].

The next extension to JavaBeans is the Bean Context environment, better known as the extensible runtime containment and service protocol described in [Lau98b]. The specification defines its purposes as

> "...to introduce the concept of a relationship between a Component and its environment, or Container, wherein a newly instantiated Component is provided with a reference to its Container or Embedding Context. The Container, or Embedding Context not only establishes the hierarchy or logical structure, but it also acts as a service provider that Components may interrogate in order to determine, and subsequently employ, the services provided by their Context." [Lau98b, Lis99]

It defines how JavaBeans can be logically structured (in a hierarchical way) in their environment. Like in [Bar99] the primary goal is to provide an extensible service mechanism through the use of a protocol. The Bean Context environment tries to enforce this by defining a container representing the Beans' environment and the actual component (a Bean) resides inside the container. The container allows the Bean to interrogate it about the available services of the environment which are available to the Bean. This means the Bean Context API provides the programmer tools to logically group beans in a context and let them interact with it.

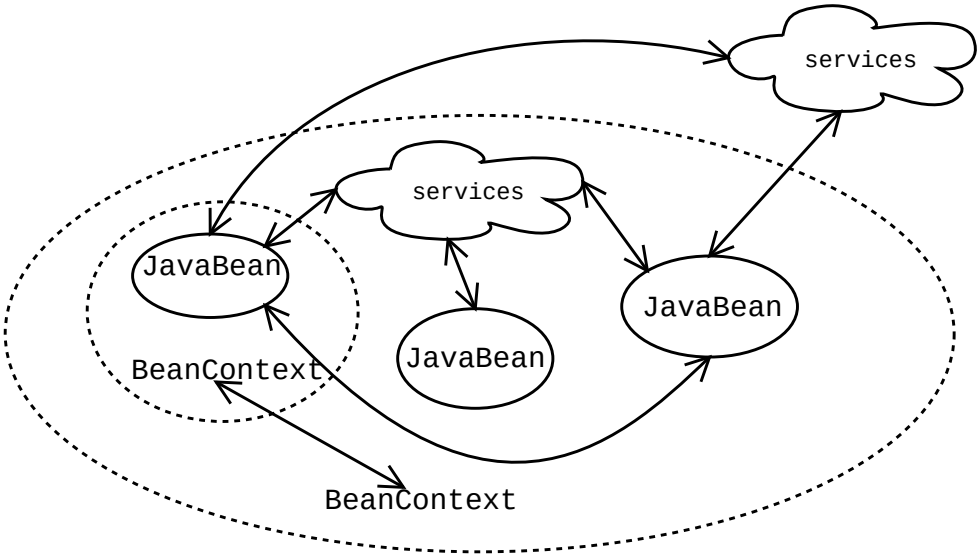


Figure 3.1: JavaBeans grouped hierarchically in contexts

### 3.2.3 Comparison

While the ActiveX market has the largest share of the component market, JavaBeans were never so successful on the component market. JavaBeans are more successful on the free and open source market it seems. I guess this will not change as long as most PC's will be equipped with a Microsoft OS.

#### Transparency

How about location transparency with ActiveX? It is difficult to make an uniform statement, for the different kinds of ActiveX techniques. It is easier to look at the location transparency

at a lower level: COM or DCOM. , like we did in section 2.1.5. In its most natural form, an ActiveX component is just a COM object, so transparency properties for COM or DCOM can be used for ActiveX. As with DCOM it is obvious real distributed programming is not possible with just ActiveX. Because, by consequence, all things DCOM misses, ActiveX misses also. In the current use, internet download-and-run components, that is probably no problem. Consider this example: due to increasing bandwidth and speed of internet communication, the user is no longer obligated to buy and install a new piece of software, lets say a Personal Information Manager (PIM), but use it online. The user pays a company for the use of the component on the company's server and downloads just an interface for communicating with the component. And lets say the PIM is integrated in Microsoft Office. The company is very successful and a lot of people decide to use its PIM. The company decides to use several servers to offer the service. But one server fails, just the one you were using. No problem, the company claims there are lots of other servers available offering the component. But the PIM refuses to work: there is only one reference in the registry, just to that specific server that failed. As long as the server is not back online, your PIM is useless, and because you do not know where the other servers are located, and certainly not how to adapt the reference in your registry, you paid money for nothing. If real location transparency was offered by DCOM, nor you or the company would have a problem: If another component (with less workload) was available elsewhere, it would automatically be used, no matter if the (legal) server was located in Cuba, New York, Belgium, Japan or somewhere in Russia. In a way ActiveX components are transparent for the user. They are self-registering on the client machine and can work as well at the server side (with the help of ISAPI or CGI scripting), as at the client side. ActiveX can register itself in the registry of a Microsoft Windows platform. Nevertheless ActiveX support for transparency is rather poor. The client must know were to find the component if it is located on the server side. This would not be so terrible was it not for the lack of migration transparency and network transparency, because of its (D)COM foundations (see section 2.1.5).

The transparency issue is solved by adding a MTS to the environment. Components can be registered by the MTS and will provide network transparency to the components. There is one simple requirement: one must know were the server process is located to benefit from the offered network transparency (just as one has to connect with the ORB to take when using CORBA). Before proceeding and using the deployed components the client has to obtain a pointer to the server process first. It is sufficient to know the DNS name of the machine where the server process is located for getting the reference to the server process. Once the client has the reference it does not matter whether the component is located in-process on the server, out-process on the server, in-process on another machine or out-process on another machine. What does matter is that the component is registered with the MTS.

The previous paragraphs indicated ActiveX without some other help is not very suitable when there are demands for high network transparency. Its competitor, JavaBeans, does not much of a better job. Being built on Java it uses RMI to have some network capabilities. Where DCOM supports Secure RPC, RMI is hardly a save communication protocol. Sure, it can pass through firewalls using HTTP-tunneling and misguiding the firewall[1] this way, but what we really need is some authentication of the RMI-request and RMI-respond [Mar99]. This will be discussed in the next section. The transparency RMI offers is very "local" like mentioned in section 2.3.4. This is actually not surprising if you consider the way JavaBeans are defined in section 2.3 [Ham97]. They aimed at the GUI market as a primary target. If you would enforce

[1] if the firewall allows TCP/IP connections

location transparency with GUI components this means the component should be able to draw on a canvas located on another machine. Redirection of graphical output is OS-dependent and not al OS's support this[2]. JavaBeans are not network transparent in general, this is a property Sun did not address very well with the JavaBeans model. The question is, can we add network transparency like MTS did with ActiveX? A first requirement is that the component should not include GUI functionality, otherwise a download-and-run approach or possibilities for graphical output redirection are demanded to be portable. Note that ActiveX suffers from the same disadvantages. A second demand is the division between the interface and the implementation (the interface is used by the client to address the implementation and ensure transparency). This is possible using RMI together with CORBA, Sun provides RMI-IIOP in cooperation with IBM especially for this kind of situations.

There is a trend to use JavaBeans with a CORBA ORB, known as CORBA Beans. The ORB handles the distribution of the beans and offers the transparency. We will see EJB resolves some issues, but tends to narrow its services to multi-tier systems for enterprise development and is limited to middleware services, which is a too limited perspective for common JavaBeans. The proposition for a fusion of CORBA and JavaBeans comes from a paper " *CORBA Component Imperatives*" published by Netscape, Oracle, IBM, Sun and the Gang of Four (authors of [Eri95]). The name *JavaBeans++* was born; JavaBeans packaged into JAR's and registered and distributed by the CORBA ORB. The CORBA name for a packaged component (a JavaBean inside a JAR) is a *CAR*. It is even possible to import the CORBA Beans in a visual builder tool, using the Beans proxy and still supporting the introspection features. CORBA enhances the JavaBeans model with [Rob98]

- interoperability among other programming languages (there are OMG IDL mappings for C++, C, Smalltalk, COM, COBOL,...);
- an Object Request Broker, a distribution system;
- a *plug-in* framework, a JavaBean can be added to the ORB at any time;

On the other hand, JavaBeans enhances CORBA with [Rob98]:

- a ready to use component model. JavaBeans are *not* just a specification: there exists an implementation (which is just Java of course);
- usability in visual builder tools (event coupling, introspection,...);
- a proven packaging technology: JAR;

ActiveX controls and JavaBeans are targeting the same market, but are also complementary considering certain services. Although they try to offer the same services, there are differences depending on how Sun or Microsoft interprets these services, like transparency. Apparently, when designing their component models network transparency was *not* an issue. For ActiveX Controls there is only local transparency, because an ActiveX control registers within the Windows registry. JavaBeans do not even offer this kind of transparency on the local machine. It must either be included in the classpath system variable `CLASSPATH` or added as an option the command line starting the application. E.g. `java -cp /usr/local/beans/xml4j.jar -jar`

[2]the X-server on UNIX systems does, but a Microsoft OS does not provide graphical output redirection as a basic service.

`arguouml.jar` means the main class is located in the argouml.jar file and that it needs the bean stored in xml4j.jar located in the directory `/usr/local/beans`; not really what can be called location transparency. Summarizing, this means the actual use of ActiveX controls and JavaBeans is not sufficient to be called *internet* components, whatever the marketing people claim. However, they are important for the internet in their current state: as download-and-run packages. Main differences between transparency are due to ActiveX registry entries, and this takes ActiveX local "transparency" one step further then JavaBeans. A possible solution could be to register existing JavaBeans within the JRE and store the JavaBeans in a sort of "beans database" where the user can add and remove Beans as they wish. An automatic registration (keeping security regulations in mind) should be a possibility for automatic upgrading and extending Java applications. The automatic extensibility is already a fact using the JAF extension, but lacks the necessary transparency.

### Portability

Both ActiveX and JavaBeans are platform independent in their own way. ActiveX can make use of its COM basics to provide platform independence. Microsoft claims COM is platform independent[3], but it is an artificial independence. It is mostly platform independent throughout the different Microsoft platforms because of the OS specific features such as security. JavaBeans are portable as long as there is a JVM available for the target platform. Because the services the JVM offers are platform independent, the Beans will have a consistent behavior no matter on which platform they are deployed. There is one exception to this rule: the Microsoft Java Virtual Machine (msjvm). Microsoft decided to build its own JVM without respecting the Sun specification described in [Jam96]. As a consequence Java applets or applications executing inside the msjvm have access to OS resources. Microsoft introduced the WFC (the Windows Foundation Classes, a GUI api based on the MFC architecture) which runs faster than the default AWT (the msjvm still does not support Java2, so no JFC is available). Faster, because it executes native code and gets the GUI elements out of Microsoft Dynamic Link Libraries (dll). Unfortunately this implies security leaks (the msjvm security is *not* the same as the Sun JVM) and the developer looses all portability. Applets written with WFC can only execute on a Microsoft Operating System. Visual J++, the Java developer tool from Microsoft, uses this WFC and lacks code readability for non MFC-programmers, this readability tends to be an advantage of the Java language. Another example of differences is the discussion about the bound method (used in Visual J++ event delegation and multicasting) and inner classes (advised by Sun) to handle events, discussed in [The98].

JavaBeans are bound to one programming language: Java; developing JavaBeans is settling with the limitations and design of the Java language. Developing ActiveX is almost language independent, again because of the COM basics. In theory COM can use every language capable to address functions through a pointer-table. Figure 3.2 shows how COM defines its binary interface through the use of a function pointer-table (a *vtable*) containing pointers to functions implementing the services of the interface. Languages which can be used with COM include C++, SmallTalk, Ada, Basic, Visual Basic and Java. ActiveX is a step ahead on JavaBeans here, offering full flexibility in choice of language. The developer can choose the best suited language to tackle a problem and still benefit from CBSD using a COM wrapper as interface. The Beans are restricted to one language, Java, which is not the silver bullet and has its drawbacks (like performance due to interpreted bytecode and limited access to system resources like printers).

[3]COM implementations exist for different flavors of UNIX, Solaris, AIX and Macintosh.

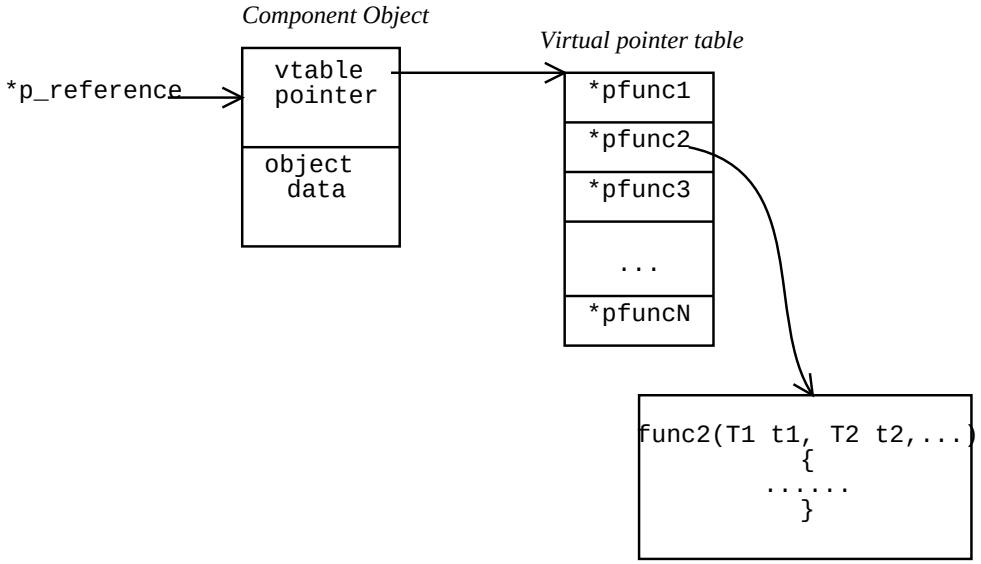


Figure 3.2: COM internal working

### 3.2.4 Building bridges

In the introduction of this chapter the possible complementary behavior of Beans and ActiveX controls was recognized Beans support a better security scheme, ActiveX has performance advantages etc. To make the two work together, *bridges* are built.. There are two possibilities: package an ActiveX control inside a Java class, or package a Bean inside a COM wrapper. An important help is the Microsoft JVM when bridging components.

The first bridging possibility is using scripting to connect ActiveX with Java (JavaBeans or Applets in general). In [Sus98] describes how VBScript can be used. VBScript working with ActiveX is a normal way to interact with ActiveX controls. The ActiveX runtime for Java makes Java Applets also controllable with any kind of ActiveX scripting. With the introduction of Visual J++ with the Microsoft Java SDK 2.0 real bridging was introduced. [Tre] describes how to embed an ActiveX control in a JavaBean or a Bean inside an ActiveX control. It sounds surprising, but packaging ActiveX inside of Beans makes it a lot easier for programmers. Many COM specific features are hidden for the user and tackled by the Microsoft JVM. The Microsoft JVM is actually an ActiveX component! Packaged inside a Bean, the ActiveX is limited to the same security limitations the JavaBean has and takes advantage of the Java security model this way. It works also the other way around: JavaBeans can be packaged into ActiveX components and benefit from the advantages of ActiveX like registration in the registry, importing possibilities in popular development environments Visual Basic, Delphi or PowerBuilder and interaction with ActiveX scripting languages. Sun has also released an ActiveX packager for JavaBeans, described in [JPI99]. It overcomes a disadvantage discussed in 3.2.3; inside ActiveX Beans are subject to a COM wrapper meaning the used programming language for interaction with the component does not have to be limited to Java but can include many others.

CORBA can be used as a bridge between both because it specifies COM mappings, OLE Automation mappings and Java mappings of course. [Obj99] describes COM, DCOM and OLE Automation mappings. As an example listings 3.4 and 3.3 show how the COM *ConnectionPoint Service* is mapped onto the OMG IDL. The ConnectionPoint service is used to support event notification in OLE custom controls (OCX). Besides this example, an example about the OMG

Naming Service is presented in [Obj99]. To use CORBA the programmer needs to wrap the component in an IDL interface and deploy it through the CORBA ORB.

```
interface IEnumConnections;

interface IConnectionPoint:: CORBA::Composite,
                             CosLifeCycle::LifeCycleObject{

   HRESULT GetConnectionInterface(out IID pIID)
      raises (COM_HRESULT);
   HRESULT GetConnectionPointContainer
         (out IConnectionPointContainer pCPC)
      raises (COM_HRESULT);
   HRESULT Advise(in IUnknown pUnkSink,
         out DWORD pdwCookie)
      raises (COM_HRESULT);
   HRESULT Unadvise(in DWORD dwCookie)
      raises (COM_HRESULT);
   HRESULT EnumConnections(out IEnumConnections ppEnum)
      raises (COM_HRESULT);

   #pragma ID IConnectionPoint = DCE:B196B286-BAB4-101A-
                                 B69C-00AA00241D07;
};
```

Figure 3.3: The OMG IDL ConnectionPoint interface

## 3.3 Microsoft Transaction Server vs Enterprise JavaBeans

Next two component models for which are targeted for use in enterprises are compared, namely the *the Microsoft Transaction Server* and *Enterprise JavaBeans*, both *middleware* models. The former is already introduced in section 2.1.1 and the latter in section 2.3. The following sources were of great help: [Gop99], [Bil98] and [Ann98a]. The first provides the reader with a very detailed and objective comparison between MTS and EJB. The second reference is rather subjective and tries too much proving the benefits of MTS over EJB, nevertheless it gives some useful criticism on the EJB model. The third reference mentioned is a reaction on the second one, trying to refute the bad criticism on EJB. [lab99] is a very comprehensible and rather technical introduction to EJB, specific for enterprise developers. Of course tons of technical information about MTS is available in [Mic99]. In both MTS and EJB transaction management is a central service. In all of the previous mentioned information sources this is discussed. In our discussion it will be only slightly touched in favor of transparency and security.

### 3.3.1 Microsoft Transaction Server

The MTS architecture is actually very simple (in theory that is, like COM it tends to be very complex in practice). First of all there is the client, using components deployed through the MTS.

```
interface IEnumConnections;
[object, uuid(B196B286-BAB4-101A-B69C-00AA00241D07),
        pointer_default(unique)]

interface IConnectionPoint: IUnknown {
   HRESULT GetConnectionInterface([out] IID *pIID);
   HRESULT GetConnectionPointContainer(
       [out] IConnectionPointContainer **ppCPC);
   HRESULT Advise([in] IUnknown *pUnkSink,
       [out] DWORD *pdwCookie);
   HRESULT Unadvise(in DWORD dwCookie);
   HRESULT EnumConnections([out] IEnumConnections **ppEnum);
};
```

Figure 3.4: The Microsoft IDL ConnectionPoint interface

The MTS Executive[4] is the core process of the service, it is responsible for the lifetime cycle of objects. The "parent" process in which the MTS Executive lives together with the deployable components is situated in `mtx.exe` [Gop99]. The architecture is visualized in figure 3.5. Some of

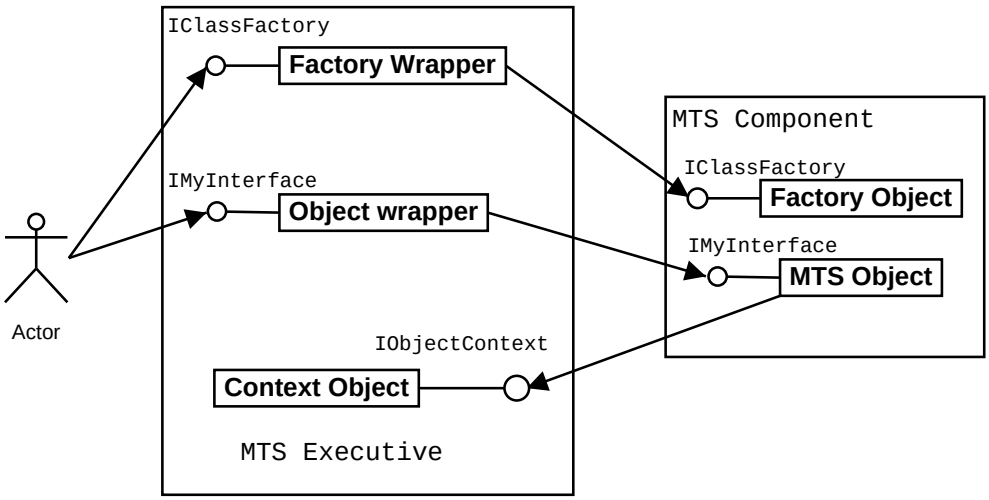


Figure 3.5: The Microsoft Transaction Server architecture

the most important features of MTS, like transaction management for components, are summed together in section 2.1.1. We are in particular interested in security and transparency features, for the security features see next chapter and appendix A.

The default security model MTS uses is NTLMSP, introduced in section 2.1.4 and discussed further in appendix A. Just as COM and DCOM MTS is dependent on the Operating System security services (like NTLMSP for Microsoft Windows NT). MTS can be used as a mean to deploy components, just as an ORB (more in particular the CORBA ORB) does, but MTS is not considered to be a distributed programming model. Its purpose is deployment in a three-tier environment and allow Microsoft Windows and UNIX clients to call its components

[4]the MTS Executive is situated in the `mtxex.dll` dynamic link library

using DCOM. The fact its purpose is a three-tier environment makes it useful for interactive and dynamic web pages. Active Server Pages[5] (ASP) of an Internet Information Server[6] (IIS) can invoke MTS components, so you can call a MTS component from a web-browser resident component. MTS provides an easy integration with the Microsoft IIS. The way this works is showed in figure 3.6. Two possible ways are shown in the figure: one way is the client speaking directly to the transaction server and another is the client using the transaction server through a HTTP connection using Active Server Pages on an Internet Information Server. The IIS is responsible for the connection with the transaction server, the client has no knowledge at all of the transaction server [Mic99].

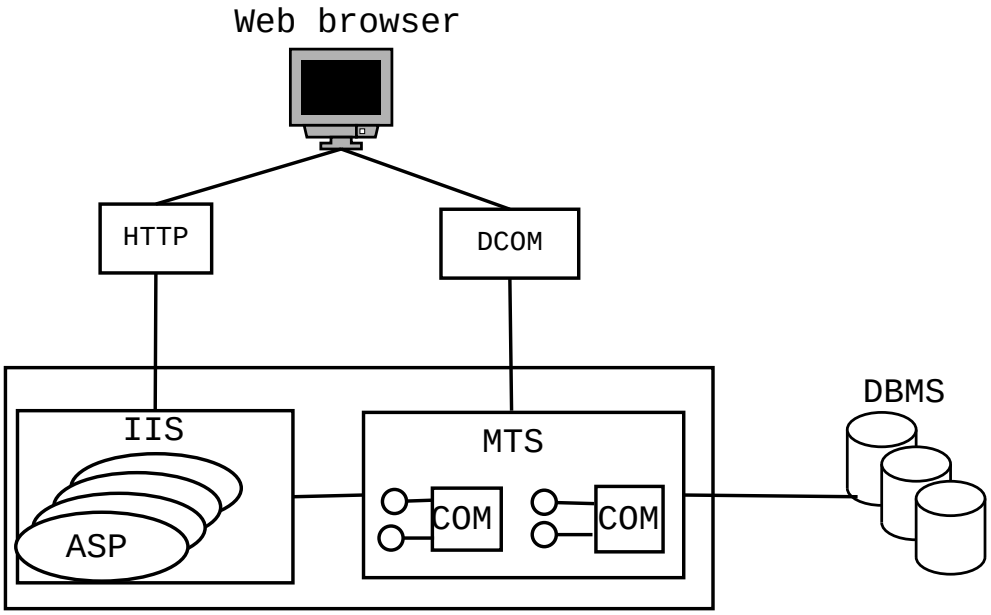


Figure 3.6: The Microsoft Transaction Server and the Internet Information Server

### 3.3.2 Enterprise JavaBeans

The EJB architecture resembles the MTS architecture, both consist of a server environment, a factory, a wrapper for the real component and a context. The server environment is the container, responsible for lifetime cycle management. The factory is called the *home object* which is accessible in a naming service through the use of JNDI[7] and has the same functionality as the MTS `IClassFactory` object. The home object is responsible for the location, creation and destruction of EJB classes. The real Enterprise JavaBean, in which the application logic resides, inherits from the `javax.ejb.SessionBean` class or `javax.ejb.EntityBean` class, the difference between both will be explained later on. The wrapper for this EJB is a remote interface, a proxy providing a stub and skeleton using RMI to offer remote use. This remote interface inherits from the `javax.ejb.EJBObject` class, and this class inherits functionality from `java.rmi.Remote`. It

[5]"Microsoft Active Server Pages is the server-side execution environment in Microsoft Internet Information Server 3.0 that enables you to run ActiveX scripts and ActiveX server components on the server. By combining scripts and components, developers can create dynamic content and powerful Web-based applications easily." [Mic99]

[6]"The Microsoft Internet Information Server is designed to deliver high speed, secure information publishing while also serving as a platform for developers and independent software vendors (ISVs) to extend the Internet's standard communication capabilities." [Mic99]

[7]this means the name space where the home object is located has to be known.

is with this interface a client can access the services offered by the Enterprise JavaBeans. The architecture is shown in figure 3.7.

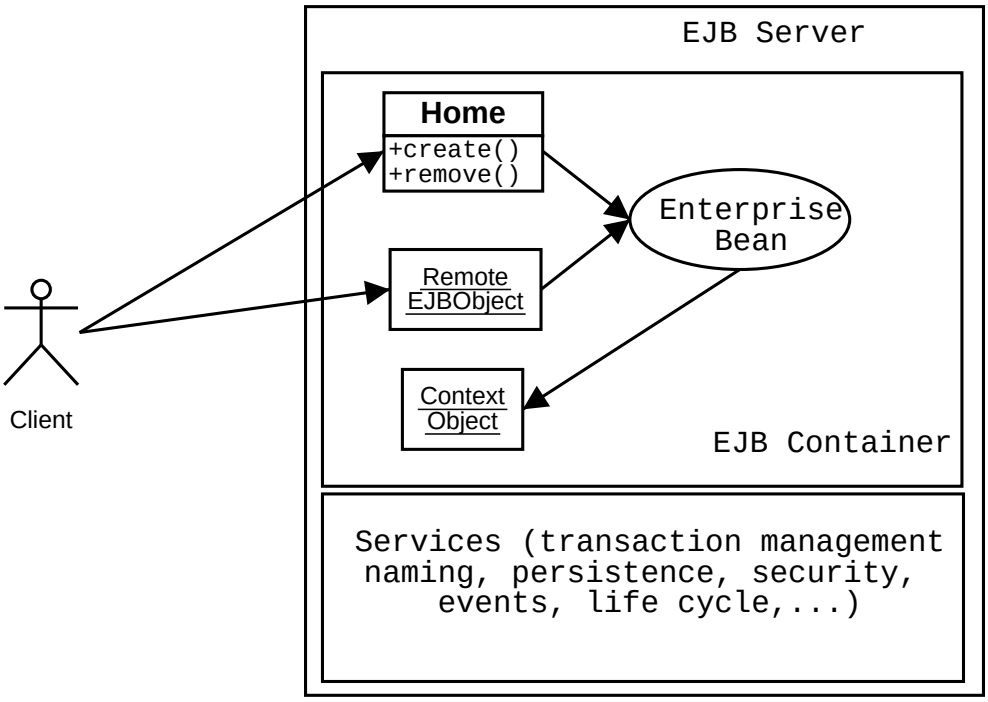


Figure 3.7: The Enterprise JavaBeans architecture

The EJB specification [Vla99] assumes IIOP will be used for communication between different vendors and/or over the internet. This implies CORBA would be a good choice and Sun apparently tries to push EJB developers towards CORBA with a EJB-to-CORBA mapping described in [San99]. Whether this is a positive evolution is not so sure: CORBA is also a specification and this implies there is no warranty to have a complete interoperable implementation. So the client can connect to the home object using RMI if the client is written in Java or use CORBA and IIOP if the client is written in another programming language. Of course, CORBA can also be used together with Java clients [lab99].

### 3.3.3 Comparison

It is obvious the designers of the EJB specification were inspired by the MTS architecture, the two architectures are basically the same as the reader can see in figure 3.7 and figure 3.5. Where MTS defined four choices to start an automatic transaction for a component call, EJB uses the same choices. The four values MTS defines for a components involvement in a transaction are [Bil98] (a client is the principal invoking the method):

**requires new** a new transaction will be started when the client calls a method on an object of the component;

**required** if the client is already involved in a transaction the invoked method will become part of the transaction, otherwise a new transaction will be started for the invocation;

**supported** if the client is already involved in a transaction, the new invocation will become part of the transaction. If the client is not already part of a transaction, the new invocation will be executed without being part of a transaction;

**not supported** each invocation will execute without being part of any transaction, even not if the client is already part of a transaction;

EJB has a different view on the responsibility over transactions: the specification defines three sorts of "managers" for transactions [lab99, Bil98]:

**client managed transactions** a client is responsible for starting and finishing a transaction using explicit calls;

**bean managed transactions** the bean is responsible for starting and finishing a transaction (session bean only). The bean creates a transaction and returns immediately. Any subsequent calls will be received by the container. The container decides whether the new call (of a thread) to one of the methods of the bean can re-enter using the following set of rules:

- if the bean-created transaction is still in effect and the calling thread carries a transaction which is the same, the thread is granted access;
- if the bean-created transaction is still in effect and the calling thread carries a transaction which is different, the container will suspend the association from the calling thread's transaction and associate it with the bean's transaction;
- if the bean is not involved in any transaction and the calling thread is involved in a transaction, the container will suspend the association from the calling thread's transaction and allow it to enter.

The thread's former transaction (if it had one) is restored when it finishes the call.

**container managed transactions** the bean can not perform the transaction, all responsibility is given to the EJB container to start and finish the transaction. EJB has used the same values as MTS defined, and added two [lab99, Vla99]:

**Mandatory** The client must already be part of a transaction, otherwise the EJB container will throw an exception;

**Never** The client is never allowed to do an invocation while it is already involved in a transaction.

The bean has a deployment description where these six values can be set for the container to read to manage the appropriate sort of transaction behavior

EJB has a far more complex transaction management, because of the broad range of choices in responsibility and transactional behaviour. For flexibility this can be an advantage, but limiting choices like MTS did could hide some complex decision making for the developer.

There is one fundamental difference between both: MTS is an *implementation* while EJB is a *specification*. Although there is a specification for EJB, the implementation is still vendor specific. EJB claims to be portable but the specification leaves room for vendor specific services (compromising the portability by consequence); the specification states: "specialized containers can provide additional services beyond those defined by the EJB specification. An enterprise Bean that depends on such a service must be deployed only in container that supports this service" [Vla99]. With MTS Microsoft provides an uniform available implementation and maybe that is just what developers want. When using EJB, they need to choose a vendor for an

implementation and most of the time take the vendor-specific features into account. The allowed flexibility by the EJB specification was probably a bad idea, and does not fully support the portability principle.

Unlike MTS, EJB components can have a state. MTS components do not have a state; they are stateless and ensure database consistency this way [Gop99]. MTS components only have the notion of a state when being in a transaction. When a transaction is finished, the state will be destroyed. The EJB model can use stateless components as well as stateful components. The EJB model knows two kinds of Beans: the *Session* Bean and the *Entity* Bean. The Entity Bean represents data from the domain layer; in most applications this means it represents data from the database (a row, a joined table, a view on a joined table,...). The Session Bean is the Bean created through the use of the Home Interface in figure 3.7 and is the one the client works with using the remote interface. This means it can hold a state on behalf of the client (stateful), but it is as much possible it does not maintain a state and can be shared with other client connections [lab99, Vla99]. Figure 3.8 depicts the architecture and shows the different uses of Entity and Session Beans.

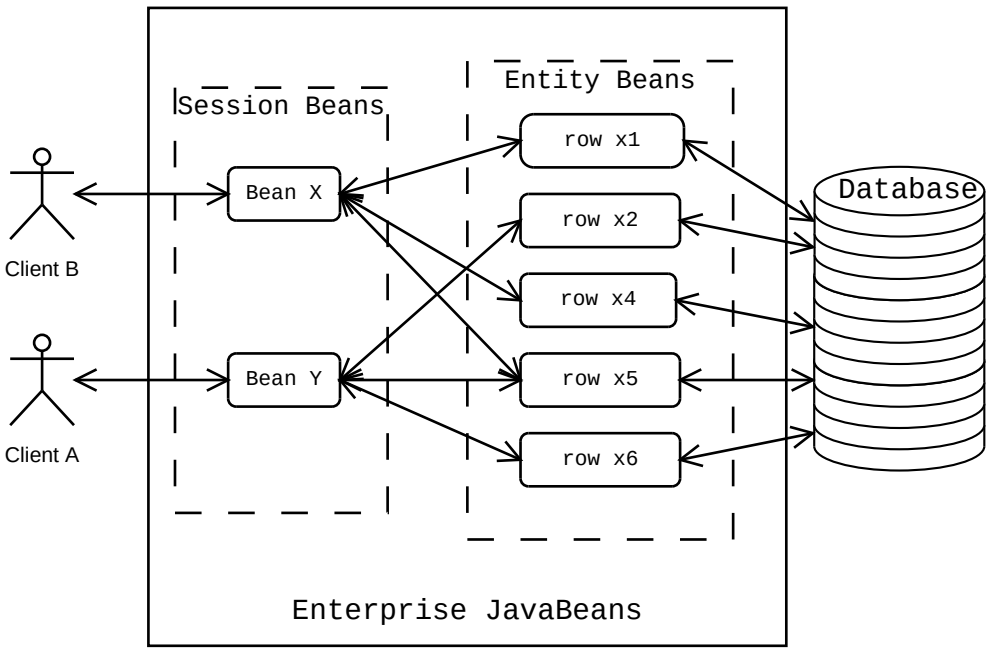


Figure 3.8: Session and Entity Beans

# Chapter 4

# Security issues of components

## 4.1 Security matters!

There are a lot of examples available to prove security is a very important issue for internet users. That is why we must take care to make the internet as safe as possible. I do not use the word *safe* on purpose because most of the time making an 100 % waterproof safety-system is not a realistic thing to do. That is why most of the time we choose to make things as safe as possible. Maybe the words "as safe as possible" need some explanation. Why can we not make things totally safe, without any threats. There are several reasons:

- The only real safe system is a stand-alone system, without any network connections and locked physically by the owner. (If physical violence is out of order.)
- The encryption algorithms used today are almost always based on writing down very large numbers as multiplication of prime numbers. This is known to be an NP-Complete problem. It is most likely there will never be an algorithm to do this very fast, but we can not be sure because the formula $NP \neq N$ is not proven *yet*. On the other hand we have the increasing computing force, which may not be a solution for an exponential problem, but reduces the calculation time for finding these prime numbers nevertheless. In the Case-study, some of these encryption algorithms are described.
- We are humans and we are not perfect. We make mistakes and these mistakes can be opportunities for malicious people.

## 4.2 Security threats and attacks

Before getting into the security concepts of components, lets first take a look what the actual threats or methods of attack in a networked environment are. The possible *methods of attack* in network communication to be taken into account are [Geo94]:

**Eavesdropping** stealing copies of information without permission. For example: a network station **A** on the internet uses the address of another workstation **B** for receiving the information meant for workstation **B**.

**Masquerading** sending or receiving messages using another identity (without permission). For example: a network station **A** asks authorization to a key server **B**. Station **C** uses the identity (address) of **A** to let **B** think it is **A**. **B** resents authorization to **C** thinking it is **A**. **C** can use this authorization key for some services previously inaccessible for **C**.

**Message tampering** a big problem in a store-and-forward network. A message can be intercepted, altered and forwarded to its destination. The changed message arrives at the destination and the destination thinks it comes from the originator, but it has changed somewhere in between. Ethernet networks are no problem, but when routers are involved this may be a problem. Also known as the "man-in-the-middle" attack.

**Replaying** a message is stored and used some time later. For example: A key server **A** gives workstation **B** permission to access service **C**. **B** receives an encrypted key of **A** and stores it on a persistent medium. Sometime later **B** is excluded any permission to access service **C**. **B** uses the stored message containing the authorization key received from **A** for accessing **C**. **C** assumes **B** has permission because it has the key, while **A** would not grant permission to **B** again.

Now we have described possible methods of attack, it is handy to classify the threats for computer systems. These *threats* are:

**Tampering** unauthorized alteration of information. For example: somebody could change a boolean into "yes" in a message while its value was "no" and prolong a license this way or alter the amount of a bank transaction and transfer the deposit to its own account;

**Leakage** unauthorized acquisition of information;

**Resource Stealing** the use of resources without authorisation;

**Vandalism** Doing harmful operations to a system without any gain for the perpetrator;

To end this discussion note most attacks originate of legitimate users of a system. 90 % of attacks on networked software systems begin with *social contacts*. Imagine an employee of a university department receiving a call of somebody claiming to be a system supporter. This system supporter says the systems has some technical problems and he or she has to solve these. He or she asks a login and password for access to the system and enabling him or her to "repair" the system. The university employee, badly informed, gives the malicious repairer a login and password and the repairer has access to system resources that were impossible to access without the login-password pair. Conclusion: it is important users of the systems are informed of this sort of threats and are careful giving information to informal defined persons. Identity is a very important concept in security matters, not only in the world of computer connections.

## 4.3 Components and security

How do our component models consider security? ActiveX with (D)COM, CORBA and JavaBeans, EJB are all very likely to be used in networked environments, but are they secure enough? To examine this, the threats and methods of attack described in the previous section will be used in order to observe the capabilities of the components to handle these.

### 4.3.1 COM and security

The first thing about COM and security mentioned in section 2.1.4 told us that security for COM is

- Operating System dependent (permissions)

- based upon DCE-RPC

The COM security service build upon DCE-RPC is more important for DCOM. Table 2.2 described the various security levels possible for DCOM. The COM specification ([Mic99] states COM actually knows two sorts of security: *Activation Security* and *Call Security*. Activation security restricts how objects are created, connected to others and secured. Call security restricts how an object can use a server using an established connection. These are equally important to DCOM of course. COM+ security is a combination of COM security (see 2.1.4), MTS security (see 2.1.1), Authenticode (see 4.3.2) and the Microsoft JVM security. The most important security addition COM+ ships is *code access security*. These access checks are based upon *roles*[1] and *privileges*. Besides access checks, *code-security* is also provided by COM+ using Authenticode for this purpose [Mar97].

### 4.3.2 ActiveX and security

ActiveX is built on COM, so it inherits the security features of COM. The way ActiveX is deployed at this time is not distributed, but in a *download-and-run* fashion, and this is were new security enhancements are needed. If the user visits a website which uses an ActiveX control, the system checks the registry if the control is already on the computer. If it is not local available, the system needs to download it and register it first before the control can be used. This means there must be authentication of the sender before downloading the control[2]. Suppose there would be no control at all. A user would download an ActiveX control, without suspicion, and the control would register itself and be stored at a local storage medium. When the control executes the next time, it has enough permissions to delete a couple of files (*vandalism*), send some private documents into the world (*leakage*) and alter sensible information like initialisation files (*tampering*). Everything happens within the users permissions, which can be restricted in Windows NT, but are pretty much extended in Windows 95. A lot of harm can be done this way and this indicates the kind of OS platform is important for ActiveX security.

Unfortunately Windows 9x and Windows NT have different security settings; a user using windows 9x as platform is more vulnerable then a NT user. Windows NT (v3.51 and v4.0) is subject to the *C2* security guidelines submitted by the US government for governmental systems[3], also known as the *Orange book* guidelines. These are depicted in table 4.1.

A lot of the effective use of these guidelines are dependent of the administrators local policy of course.

The most important and probably also most visible security service to the users of ActiveX components is the *authentication service*, also referred to as *code signing*. The download-and-run method requires to precisely identify the originator of the component. This is done by *certificates* which can prove the originators identity. These certificates are generated by *certificate authorities*, third parties trusted by the user. If the system needs to download the control which is not signed it will warn the user and ask its explicit permission. Otherwise, if it is signed it will show the certificate if this certificate is not yet known and trusted by the system and download the component after confirmation [Zan97, San96]. The code signing mechanism also protects the user against altered or tampered code since the release and verifies the principals identity.

[1] "a role is an abstract group of users" [Mar97]

[2] Only ActiveX controls will be considered here

[3] it seems these guidelines are insufficient. A computer worm called "I-LOVE-YOU", sent around the world via e-mail, even got into the US governmental systems during the first week of May 2000.

| |
|---|
| 1)**user identification and authentication**<br>A principal must have a way to define its real identity before access is granted to the system. |
| 2)**auditing**<br>All the actions (read, write, change, delete and execute) of a principal can be logged. |
| 3)**discretionary access level**<br>Resources (including software components) have owners who can grant and restrict access at various levels. |
| 4)**object reuse**<br>Terminated, deleted or discarded objects can not be used by other principals. |
| 5)**system integrity**<br>resources (memory, files,..) owned by a particular principal can not be read, changed, deleted, executed or written by another principal. |

Table 4.1: C2 security guidelines

The mechanism Internet Explorer uses is Microsoft property and is called *Authenticode* and is derived from a public key signature algorithm. A hash-value is calculated from the bytes in the code and this value is signed using a private key. This hash-value is inserted into the file and will be controlled if it is downloaded by a user. The user calculates a hash-value from the file and compares it to the hash-value which was inserted in the file by the originator; a match means the file is not tampered with. To authenticate the originator a certificate encrypted with the private key of the originator is inserted in the file. Figure 4.1 shows the nesting of the public keys required. If the user wants to proceed to an inner level it must have obtained the public key of the current level in the package and apply this to get the next one [Mic97].

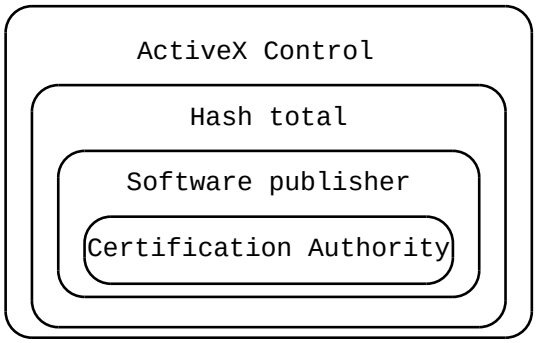


Figure 4.1: Public key nesting with Authenticode

Authenticode is built on the *Windows Verify Trust* API (WinVerufyTrust API), a Microsoft Win32 API. The API has a *Software Trust Provider* for checking digital signatures of components which is used by Authenticode to check the integrity and authentication of a component. The trust verification tries to answer whether a component or resource can be trusted for something specific according to a specific authority [Mic99]. Notice the Authenticode approach is complementary with the sandbox model, moreover it is a tool to enforce restrictions (or to allow more than the usual restrictions) in a sandbox model. The sandbox problem can decide to limit resource access with the information retrieved from Authenticode (can the principal be trusted?). Microsoft has also recognized the need for fine-grained security settings and to allow trusted principals to access some predefined system resources.

### 4.3.3 CORBA and security

CORBA defines a security and licensing service, but was pretty late in defining it. The OMG released a first document concerning security entitled *The OMG Security White Paper* in 1994. It took more than three years to complete the specification. The CORBA Security Service specifications include [Obj98] :

**authentication** to prove the identity of a principal;

**authorization and access control** to only allow the principals with the right permissions to make actions without breaking the permissions;

**auditing** accounting information about the actions a principal makes;

**secure communication** to avoid message tampering;

**non-repudiation** the ability to be certain of a principals identity and actions, so denial is impossible;

**administration of security information** for example, security policies

The Security Service Specification does not include the specification of cryptographic services. This means the provision of cryptographic functionality is totally dependent on the provider of the ORB. It is not easy to consider security issues for a security service enabled to work in heterogeneous environments. The security has to be independent of the environment where processing must be done, but on the other side [Obj98] claims:

> "If the system is installed in an environment that also includes a procedural security regime, the composite system should not require dual administration of the user or authorization policy information."

So, the ORB must be able to use local security policies anyway. To allow complete portability of CORBA objects, the particular security services should be hidden behind a well defined interface. This is a difficult task: ensuring portability *and* allowing usage of system dependent security policies will lead to conflict situations one way or another. Besides portability, security properties for interoperability are interesting for internet components. How secure is ORB-to-ORB communication over IIOP?

If two ORBs connect to each other over the internet (using IIOP) it is possible they both have different security mechanisms and properties on different platforms. OMG has defined an overall security framework supporting different kinds of security policies to overcome this difficulties. First of all it must be able to identify principals in a way there is no doubt possible. For example, consider a "fake" object using the same interface as the "real" object but doing some nasty things. A client should be able to know if the connected component is the one it wants to use and not the fake one offering the same interface. There are three security levels for interoperability. The Common Secure Interoperability (CSI) packages describe the level of security that can be used with ORB-to-ORB communications. They are defined to be used with different ORBs and on different operating systems [Obj98]. The following items can also be found summarized in [Jan99] as a description of the CORBA security service:

**level 0** Identity-based policies without delegation; only the identity of the initiating principal is transmitted from the client to the target, and this cannot be delegated to other objects.

**level 1** Identity-based policies with unrestricted delegation; only the identity of the initiating principal is transmitted from the client to the target. This can be delegated to other objects (and have impersonation as consequence).

**level 2** Identity- and privilege-based policies with controlled delegation; attributes of initiating principals passed from client to target can include separate access and audit identities and a range of privileges such as roles and groups. Delegation of these attributes to other objects is possible, but is subject to restrictions.

These levels define how tight the authentication protocol must be. To make Inter-ORB communication safer, there must be a layer underneath the GIOP (see 2.2.5) offering secure message transmissions and an establishment of *Security Context*[4] objects. OMG has specified such an extra layer for the GIOP, named the SECIOP; the secure inter-ORB protocol.

The SECIOP layer is divided in two layers: a *sequencing layer* and a *context management layer*. The sequencing layer is the interface used by the GIOP layer, it is responsible for the secure and reliable communication. The SECIOP encapsulates the fragments it receives from the GIOP layer into other, SECIOP-specific frames for security. The sequencing layer passes the fragments to the context management layer where they are encrypted and encapsulated in a *context management* message. The cryptographic enhancement is produced by the *data protection* layer by inserting tokens to protect the data. It is the context management layer which uses the transport layer to communicate with the destination [Obj98]. Figure 4.2 depicts the position of the SECIOP layers. It is the intention to host security protocols with SECIOP, which can be selected depending on requirements and facilities. There are three security protocols defined in [Obj98]:

**SPKM Protocol** Supports identity-based policies without delegation (CSI level 0) using public key technology for keys assigned to both principals and trusted authorities.

**GSS Kerberos Protocol** Supports identity-based policies with unrestricted delegation (CSI level 1) using secret key technology for keys assigned to both principals and trusted authorities. It is possible to use it without delegation (providing CSI level 0).

**CSI-ECMA Protocol** Supports identity- and privilege-based policies with controlled delegation (CSI level 2). It can be used with identity, but no other privileges and without delegation restrictions if the administrator permits this (CSI level 1) and can be used without delegation (CSI level 0).

Of course, there are also security protocols with no need for SECIOP to be hosted on. CORBA support the *Secure Socket Layer* (SSL) which is hosted on IIOP.

### 4.3.4 Beans and security

In 2.3.3 there was described how the security of JavaBeans was dependent on the JVM used. If the security of the Java components is to be investigated namely JavaBeans and EJB, consider most of all the security of the Java language itself which have been partially done in 2.3.3

[4] "For each security association, a pair of Security Context objects (one associated with the client, and one with the target) provide the security context information" [Obj98]

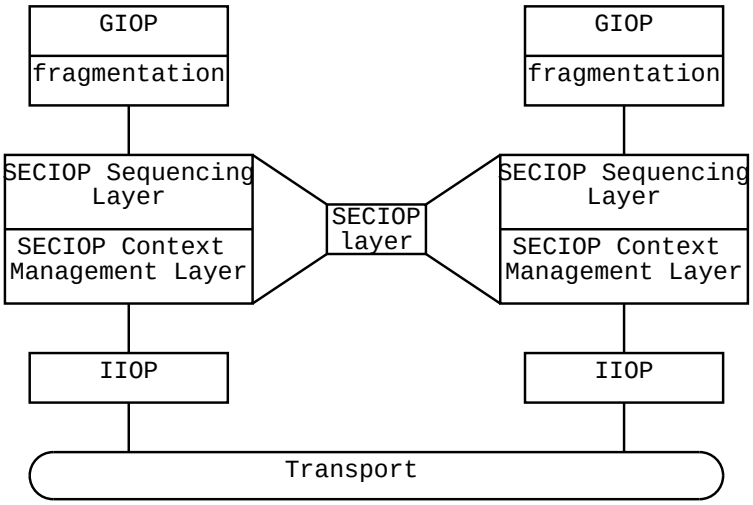


Figure 4.2: GIOP extended SECIOP

To configure the security properties, Sun ships the `policytool` with the JDK. This tool allows the user to configure the Java security properties within details and allow certain principals to access the system and use system resources which fall outside of the old *sandbox model*. The nicest thing is all of this is OS independent and the Java2 security manager defines classes of resources like filesystem, sockets etc. to be configured individually. The last couple of years Sun expanded their security system with API's for certificates, code signing, ciphers, digital signature generation and verification and lots of other stuff. For creating and managing keys and certificates the `keytool` is included with the JDK. The `keytool` can be used for managing a *keystore*, which is a database containing private keys and X.509 certificates (see also section **??**) authenticating the corresponding public keys. For a complete and comprehensible overview about security in Java2 reading [Mar99] and the security section in [Lis99] is recommended. Though the Security Extension is platform independent there are slight differences between some implementations. These differences are based on the export regulations the US government has laid on cryptographic software (cryptography is considered as a possible weapon, thus falls inside the weapons export regulations of the US). E.g. The RSA (see also **??**) algorithm is not included in the Java Security Extension outside the US, but can be used by programmers inside the US.

Like ActiveX code signing, authentication and sandboxing discussed in the previous section, JavaBeans offer the same set of features. Unlike ActiveX, the code signing is not part of the component itself; it is not a JavaBean feature. For secure component exchange a Bean wil be deployed inside a JAR which is signed using the signing and verification tool. Such a signed JAR contains a *signature file* (with extension .SF) and a *signature block file* (with extension .SDA). The signed file describes the file name (of the signed file), the name of the digest algorithm and the *SHA* digest value[5]. The SDA file is also responsible for the certificate or certificate chain, authenticating a public key which corresponds to a private key, used for signing the file [Mar99]. If a JAR is signed using the `jarsigner` tool and the `keytool`, it can be verified using the `jarsigner` tool. The following procedure has to be followed (as described in [Lis99, Mar99]):

- The originator (sender) of the JAR has to do the following:

  1. use the `keytool` to generate a keypair and a certificate;

[5]A digest is a *digital fingerprint* generated by a cryptographic hash functions for an input string. This hash value can be used to uniquely identify the input string. [A. 97]

2. sign the JAR file with the generated private key;
3. make the certificate (say `safefile.cer`)publicly available: export it to a file and make this file available for users of the JAR;

- The user (receiver) of the file has to do the following:

1. import the certificate (`safefile.cer`) and store it into a keystore;
2. use the policytool to grant the JAR some permissions, based on its authenticated identity;
3. (automatically) verify the signature;

Besides JavaBeans, Java Applets also benefit from JAR signing. It is currently common practice to store all files making up an applet in a JAR file to reduce download latency. This can also be used to identify these applets through the JAR signing mechanism and allow them to "get out of the sandbox" and use more system resources. The `policytool` allows a user to give some extra permissions to a certain identified originator. Either way, Sun advise developers to write JavaBeans in a way they can be used in untrusted applets. There are three points to take into consideration [Ham97]:

**introspection** a developer should assume less permissions in a run-time environment and virtually all permissions in a design-time environment;

**persistence** at both design-time and run-time persistence is possible. At run-time the developer should assume less or no control at all over the serialization stream;

**GUI merging** GUI merging will be restricted to the "container" where beans reside next to each other and will be forbidden towards the parent container (the parent application of the untrusted applet).

### 4.3.5 Enterprise JavaBeans and security

The Enterprise JavaBeans Specification defines three general rules for security management ([Vla99] chapter 15):

1. It would preferable if the container handles security management instead of the Enterprise JavaBean implementation, this way security management is transparent for the business services the Bean supports;

2. Deployers, application assemblers and system administrators should be permitted to set the security policies instead of the Bean developer or Bean provider to worry about the security settings;

3. Enterprise JavaBeans should be portable across platforms with different security architectures.

The Bean deployer has the greatest responsibility in creating the security policies. The application assembler can define *security views* as sets of *security roles*. Security roles are defined as "a semantic grouping of permissions that a given type of users must have in order to successfully use the application" [Vla99].

### 4.3.6 Microsoft Transaction Server and security

Most of the security concepts of MTS are already discussed in sections 2.1.4, 4.3.2 and Appendix A. MTS relies on DCOM and RPC to provide basic security but extends it with its own mechanisms. "Microsoft Transaction Server provides a distributed security service for component-based solutions. This security service relies upon Windows NT security to authenticate users, and it maps on top of the Windows NT domain topology." says [Mic99]. Like discussed Windows NT provides us with the NTLMSP protocol, which MTS MTS provides two additional security models, namely *declarative* and *programmatic*. Declarative is meant for components which are already built, there is no involvement from programmers of components. The security description will be added while packaging a component for deployment. However, in-process components do not benefit from the declarative security model. The declarative model uses MTS security roles to represent a logical group of users (which are related to the Microsoft Windows NT security domains). Access privileges will be checked every time a call is made crossing a package boundary. Programmatic indicates programmers have the possibility to enforce access control through their code. [Mic99]

In the declarative model there is a resemblance with the Java2 security model (see section 2.3.3) and its fine grained security possibilities. MTS, however, defines the access rights in the package, so no matter where the component will be used the access rights stay the same. This way, it becomes important to group components in need to access each other and package them in a way they can call each other. In case of a callback function for example, there is a need for the caller to be accessed when the function returns. So both the caller and the subject of the call must be accessible for each other.

# Chapter 5

# Conclusion

We see a wide difference between the different component models nowadays, though the fundamentals are quite the same. The forthcoming years will probably be spent on optimising the component models, making bridges between the different kinds of components and extend the several services they offer. In bridging the different kinds, CORBA can play a key role because its nature as a specification and well defined services, which are pretty complete all together. If for every pair of different components, two-by-two, there must be a bridge, we would have to build $\frac{(n^2-n)}{2}$ bridges. CORBA can become a common glue, a intermediate protocol for different components on different platforms. OMG has had a lot of criticism concerning the CORBA specifications. The biggest criticism is that CORBA was outstripped, old-fashioned before it was well introduced into the market. This is partially true; the OMG, an industry consortium, only makes specifications public if a proposed specification has been proven as a real-world application (this eliminates the criticism saying it would be a specification without real-world experience). This can lead to long waiting periods for new, usable specifications or standardization, which implies longer waiting periods for new vendor-specific implementations of these specifications. But the quality of the specification is underestimated and because it is defined considering all key elements to make good, distributed components (as discussed in section 1.2) supporting a wealth of inter-operability it is not surprising if it would become the future bridge between different component techniques and enhance existing component models with distributed properties, without changing the original component model. A primary requirement to make this allegation true is the existence of a CORBA compliant ORB with bindings for most programming languages like Java, C++ and C. It has to fully support the services which enable inter-ORB and secure communication among others.

Network transparency is a feature which could only be detected in the CORBA model; in general it still needs a lot of work. The network transparency is a key for real distributed usage, for future component usage on the internet. There will always be need for special transparency services, that allow migration and network transparency like the CORBA ORB. The fact such transparency services are not already widespread is a drawback for developing real distributed applications using software components. A general usage of Object Request Brokers equipped with inter-ORB facilities would be a step in the right direction.

The different component models discussed have their own security specification. CORBA and Enterprise JavaBeans are specifications so it depends on the implementation whether security is built in decently. Fortunately, there exist test suites for testing these kind of things. The Java language has also a well-defined security model. The weak spot there is the virtual machine: if

the virtual machine is not secure, everything running on the virtual machine will also be insecure. On the other hand gives a safe virtual machine a firm basis to build security services on. While the previous mentioned languages and models have published their security models clearly and open, Microsoft did not publish uniform, well defined guidelines for their security model concerning COM, DCOM, COM+, ActiveX and MTS. They decided to use the security model as it is provided by Microsoft Windows NT, which means the favorite and only really suitable platform for deployment of software based on COM is a Microsoft OS. If security facilities are added, it is advisable to deploy only for Windows NT, because on the security implementation of this platform relies COM. Reading Microsoft documentation about their security model leaves the reader with dubious information. For the future evolution of security models of components, a formal specification will be of great importance. There is a need for solid foundations from which a secure solution can be deduced. Security is a property in need of a clear, uniformly defined model, so it can be reasoned about. This means security should be something "open", algorithms should be publicly released. An open publication allows to find and fix security holes faster and to reassure the users for now they can verify the security model they use themselves. One could argue publishing security protocols is feeding information to malicious persons trying to break in a system, but there are several published protocols in use which have proven to be more secure as the ones hidden for the big audience. In particular for internet components it is important they are *not obligated* to rely on OS specific features, but can live in a "secure" surrounding on the host machine. This enhances migration and network transparency for components with security restrictions or demands.

The internet is not what it should be, but things are coming. It is a fast growing medium, and will soon offer a lot more then it offers today. Yes, there is interactivity and yes, there are distributed applications but still we are using it as a pool of "local" resources. Most of the time it is still more "download-and-use" and less "use". EJB and MTS have improved the situation but are focused on middle-tier management and not distributed use. Combined with an ORB, like the CORBA ORB, things can get even better. JavaBeans++ (JavaBeans and CORBA) takes it one step further: real distribution of components. We need faster communication possibilities if we want to exploit the internet fully and use component based programming for the internet in a way it shows its full advantages. An evolution towards a distributed component usage will be accompanied by a great concern for privacy and be based on a great demand for network transparency. This thesis offered the reader an overview of the security features some component models offer. Most of the security features used by these component models are sufficient and will need some optimizing (concerning execution speed) if they are to be used within such a distributed environment. Most of the time security leaks are human errors, programmers who do not take security into account or system administrators who configure weak policies. Of course, part of the security leaks also exist inside systems (they are programmed by human beings) and there will always be such security leaks. Maybe it is just to easy to say "you can never make a fully prove system" and it *is* worth trying; if it can not be perfect at least try to *approximate* a perfect state.

# Appendix A

# NTLMSP vs Kerberos

## Introduction

NTLMSP is a authentication service developed at Microsoft Windows NT4. Kerberos is an authentication protocol developed at MIT, based on the Needham-Schroeder public-key authentication protocol. To prove (in an informal way) the incompleteness of the NTLMSP authentication protocol we compare it with Kerberos (Version 5, using timestamps or sequence numbers as nonces[1]).

## Kerberos at work

see also [Geo94], chapter 16. The Authentication Service knows every members private and public key. First a few symbol declaration:

| | |
|---|---|
| C | Client |
| $K_C$ | C's secret key |
| A | authentication server |
| T | ticket-granting server |
| S | a random server |
| $Z_P$ | a message Z encrypted with key P |
| $K_{CT}$ | session key |
| $K_{CS}$ | random session key |
| TGS | ticket granting service |
| n | a nonce |
| t | a timestamp |
| $\{C,S,t_1, t_2,K_{CS}\}_{K_S}$ | A Kerberos ticket valid from time $t_1$ until $t_2$ |

The algorithm looks complex at first but is actually not so difficult to understand. Kerberos uses an *authentication service* ($A$ in the discussion) for proving a principals identity and a *ticket granting service* ($T$ in the discussion) for allowing principals to connect to servers and/or services.

1. $C \xrightarrow{\text{request TGS ticket: } message(C,T,n)} A$
   $C$ request ticket for communication with $T$.

2. $A \xrightarrow{\text{TGS session key and ticket: } message(\{K_{CT},n\}_{KC},\{ticket(C,T)\}_{KT})} C$
   $A$ returns an encrypted ticket (using the private key of T) and a session key (using the

[1]This implies the servers to hold a list of all nonces to check replay of messages

private key of C) for $T$ to $C$, including the nonce to ensure authentication of the recipient of message 1.

3. $C \xrightarrow{\text{request ticket for service S: } \{auth(C)\}_{KCT},\{ticket(C,T)\}_{KT},S,n} T$
   $C$ requests $T$ a ticket for $S$.

4. $T \xrightarrow{\text{ticket for S: } \{K_{CS},n\}_{K_{CT}},\{ticket(C,S)\}_{K_S}} C$
   $T$ checks $\{ticket(C,T)\}_{K_T}$ and generates a random session key $K_{CS}$ and a ticket for $S$.

5. $C$ tries to connect to $S$ now
   $C \xrightarrow{\text{service request: } \{auth(C)\}_{K_{CS}},\{ticket(C,S)\}_{K_S},request,n} S$
   $C$ sends a request with the ticket, an authenticator and a nonce to $S$.

6. This is an optional step: the server $S$ proofs its identity
   $S \xrightarrow{\text{Server authentication: } \{n\}_{K_{CS}}} C$

Following all the steps of the protocol we can say it is quite safe, but can consume many resources (two server-side services, one who holds all the keys, a lot of keypair generation going on, a lot of connections). The Kerberos system has been successfully tested with more than 5000 users (at MIT). There is one possible security leak: if the lifetime of the nonces is too long, replaying messages can become a threat. That is why there were attempts to use logical clocks instead of nonces.

## NTLMSP at work

see [Mic99], "COM security overview"
The Microsoft authentication protocol NTLM is not so safe as it looks at first. A description of the protocol using the same symbols used for the Kerberos explanation goes as follows:

1. $C$ sends some information to the server $S$, which controls some resources.
   $C \xrightarrow{\text{send information: } \{domain\ info,user\ info,machine\ info\}} S$

2. $S \xrightarrow{\text{sending challenge: } \{challenge\}} C$
   $S$ generates a challenge out of $\{domain\ info, user\ info, machine\ info\}$ and sends it to $C$

3. $C \xrightarrow{\text{return encrypted challenge: } \{challenge\}_{password_C}} S$
   $C$ returns the challenge encrypted with its password (!)

4. The server $S$ gets the password belonging to *user info*: $password_{user\ info}$, and encrypts the challenge with it. Then it compares $\{challenge\}_{password_{user\ info}}$ with the encrypted challenge $C$ has returned.
   $if(\{challenge\}_{password_C} == \{challenge\}_{password_{user\ info}})$
   $allow\ access$

Microsoft has some criticism on its own implementation namely: because the server never receives the actual client password (the password is never transmitted), it can not do all possible tasks for the client and the client has a bigger workload.

## What differences?

- Kerberos uses public-key encryption, NTLMSP does not. Thus a NTLMSP-encrypted message is can be attacked by a brute-force algorithm. When a malicious user gets the challenge and the returning encrypted challenge, it is *not* a difficult task to find the encryption key (because passwords are not forced to be composed out of big prime numbers) which *is* the password. Said otherwise: if a malicious user obtains $\{challenge\}$ and $\{challenge\}_{password_C}$, through the use of *eavesdropping* it is pretty simple to retrieve the password (which was used as an encryption key) an use it for *masquerading*. To calculate the password we can solve an equation with one unknown variable; the password as a key. $E_{password_C}(\{challenge\}) == \{challenge\}_{password_C}$, E stands for the encryption function and we know the challenge and the encrypted challenge, the only unknown variable is $password_C$. If public-key cryptography was used there was no reasonable solution for finding the corresponding private key, due to the exponential complexity of dividing the number in primes. The best bit-wise complexity for division in prime numbers at this moment one can get by using the *Number Field Sieve* found in 1990, it has a bit-wise complexity of $4O(2^{\sqrt[3]{\log n}\sqrt[3]{\log^2 \log n}})$, which is still exponential [A. 97]. An example of a public-key encryption algorithm can be found in **??**. The authentication can be secure, even with the encryption using the password as a key. The NTLMSP can use a one-way hash function into a cryptographic key for encrypting the challenge. Passwords tend to be short and are an easy subject for a brute-force attack, using the cryptographic key makes a brute-force attack a more difficult and time-consuming task because the encryption key is longer, e.g. A 128-bit DES key could be used instead of the simple password ([A. 97], chapter 10).

- the NTLMSP documentation says nothing about including a timestamp in the challenge, so replay is here also a possible threat if no time-outs are taken in consideration. Fortunately, all messages here are between one client and one server, making an easy checking procedure possible for this threat.

- NTLMSP compares two encrypted bit-strings in order to check ones identity, Kerberos decrypts a signed challenge which is a big difference. Kerberos can be sure only the owner of the corresponding private key has encrypted the challenge (remember the authentication service also knows all private keys). There is no decryption in NTLMSP and the passwords are not supposed to be generated using primes as a base.

- In the Kerberos protocol, services identify themselves. In NTLMSP, a service does not identify itself making impersonating a server quite easy for malicious users

There are more differences between Kerberos and NTLMSP, which reveals more weaknesses of the Microsoft authentication protocol. It is not surprising they are switching to the Kerberos v.5 implementation for authentication checking. Windows 2000 already provides an "enhanced" Kerberos authentication protocol. If NTLMSP was a secure protocol they did not have to switch to another one! NTLMSP had the advantage of being a fast protocol, because less checking and less calculations are used.

About hashing [A. 97] says:

> "...a hash $h$ function maps bit-strings of arbitrary finite length to strings of fixed length, say $n$ bits. For a domain $D$ and range $R$ with $h : D \to R$ and $|D| > |R|$ , the

function is many-to-one, implying that the existence of collisions (pairs of inputs with identical output) is unavoidable. Indeed, restricting $h$ to a domain of $t$-bit inputs ( $t > n$ ), if $h$ were random in the sense that all outputs were essentially equiprobable, then about $2^{t-n}$ inputs would map to each output, and two randomly chosen inputs would yield the same output with probability $2^{-n}$ (independent of $t$)."

This implies a hashing function is less safe then using public key cryptography, but the probability of a collision is so small it is unlikely to happen when the number of bits the hash function outputs is large enough.

# Appendix B

# Legal notice

COM, DCOM, COM+, OLE, MTS, Internet Explorer, Authenticode, ActiveX, Visual J++, Visual Basic, VBScript, Windows 2000, Windows NT, Windows 95, Windows 98 are trademarks of *Microsoft*;

OMG, ORB, Object Request Broker, OMG IDL, CORBA, IIOP, SECIOP, CSI, GIOP, OMA, CORBAservice, CORBAfacilities are trademarks of the *Object Management Group*;

X/OPEN, OSF are trademarks of *The Open Group*;

Java, Java2, Swing, JFC, JavaScript, JavaBeans, Enterprise JavaBeans, Java Activation Framework, RMI are trademarks of *Sun Microsystems*;

UNIX is a registered trademark licensed exclusively through *X/Open Company Limited*;

SET is a trademarks of *Secure Electronic Transaction LLC*;

# Appendix C

# Nederlandse samenvatting

Deze thesis behandelt component gebaseerde software ontwikkeling voor het internet. De funderingen van componenten worden beschouwd, waarbij de klemtoon gelegd wordt op specifieke eigenschappen die componenten geschikt kunnen maken voor het internet. Verschillende bestaande component modellen en technieken worden onderzocht waarbij deze specifieke gedefiniëerde eigenschappen een sleutelrol spelen. Er wordt speciaal dieper ingegaan op de beveiligingsaspecten die heden ten dage een grote rol spelen in commerciële component modellen.

## Internet componenten gedefiniëerd

Om een internet software component te definiëren, is het noodzakelijk om eerst een gewoon software component te definiëren. [Cle97] geeft de meest bekende definitie van een software component:

> " Een software component is een eenheid van samenstelling, met een contractueel gespecifiëerde interface en een expliciete context. Een software component kan onafhankelijk op de markt gebracht worden en voor compositie door derden gebruikt worden."

Voor componenten die gebruikt worden voor het internet volstaat deze definitie echter niet.

Er kunnen verschillende eigenschappen onderscheiden worden voor componenten. Deze eigenschappen kan men als richtlijnen gebruiken om "goede" componenten te bouwen. Een eerste belangrijke eigenschap (en de fundering van heel het concept van componenten) is het eerbiedigen van *object-geöriënteerde principes*. Belangrijke kenmerken van OO zoals overerving, polymorfisme, compositie, associatie, behandelen van uitzonderlijke situaties (exceptions) en dergelijke moeten terug te vinden zijn in componenten. Vooral het specifiëren van een interface is belangrijk, waarbij een interface kan gedefiniëerd worden als:

> " Een groep semantisch gerelateerde functies die samen een logische groep van diensten vormen welke het component aan een systeem kan aanbieden. "

Het belang van de interface is zeer groot; omdat de gebruiker van een component enkel met deze interface te maken krijgt is het noodzakelijk dat deze interface zeer grondig en foutloos gespecifiëerd wordt. Een component kan in feite met een zwarte doos vergeleken worden waar men enkel gegevens kan naar toe sturen en van opvangen maar nooit te weten komt hoe de interne verwerking eraan toeging. Om deze reden wordt er voor elke service (een functie of methode in deze context), geboden door het component, de volgende informatie bepaald:

- precondities
- postcondities
- uitzonderlijke omstandigheden (exceptions)
- verwacht resultaat
- invariante eigenschappen

Naast de OO principes hebben componenten, om ten volle te kunnen functioneren in hun omgeving, mogelijkheden nodig om om te gaan met *data, hulpbronnen en versies*. Versies om verbeteringen en uitbreiding toe te staan zonder dat de functionaliteit van de voorganger in conflict is met de nieuwe versie. Hulpbronnen om te kunnen profiteren van mogelijkheden die hun gast omgeving hun biedt ten volle te benutten. Omgaan met data is dan weer belangrijk in situaties zoals bijvoorbeeld gedeelde data met andere componenten, waarbij transacties een belangrijke rol spelen. Transactie-beheer heeft de laatste jaren aan belang gewonnen in de component wereld, dankzij de bloei van componenten die speciaal voor een server gemaakt zijn en door vele gebruikers gelijktijdig gebruikt kunnen worden. Gedeelde toegang tot databases en netwerkbronnen hebben transacties op het niveau van deze componenten noodzakelijk gemaakt.

Uiteindelijk zijn ook gedistribueerde eigenschappen belangrijk geworden voor componenten. Vooral voor componenten voor het internet hebben deze veel belang. Ten eerste is *overdraagbaarheid* van componenten aangekaart. Hoe goed kunnen componenten gebruikt worden in verschillende omgevingen? Zijn ze platform afhankelijk? Des te minder ze aan een bepaald platform gebonden zijn des te beter. Een tweede punt is *transparantie*, waar men 8 verschillende soorten transparantie kan onderscheiden, namelijk [Cas89]:

- toegangstransparantie (access)
- locatie transparantie (location)
- transparantie voor gelijktijdige uitvoering en data manipulatie (concurrency)
- replicatie transparantie (replication)
- falingstransparantie (failure)
- migratie transparantie (migration)
- uitvoeringstransparantie (performance)
- schalering transparantie (scaling)

Toegangstransparantie samen met locatie transparantie wordt ook wel *netwerk transparantie* genoemd. Dit soort transparantie is iets waarmee men zeker rekening moet houden voor componenten voor het internet.

In gedistribueerde systemen en netwerken in het algemeen is *beveiliging* een niet te verwaarlozen zorg. Afhankelijk van de functionaliteit zal beveiliging van componenten minder, meer of heel belangrijk zijn (bijvoorbeeld bij e-commerce moet er veel belang aan beveiliging gehecht worden). Beveiliging van componenten kan op verschillende niveaus bekeken worden: verticaal van taal afhankelijke beveiliging tot taal onafhankelijke beveiligingstechnieken. Horizontaal kunnen we de verzameling technieken en algoritmen voor beveiliging beschouwen die hun eigen belang hebben op elk verticaal niveau.

Er zijn nog andere, minder belangrijke eigenschappen van componenten die hun gebruik aantrekkelijker maken voor de gebruiker en/of ontwikkelaar, zoals *introspectie* waarbij tijdens het samenvoegen van componenten de staat en diensten die het component aanbiedt kunnen geraadpleegd en soms veranderd worden. Er wordt ook veel aandacht aan *User Interfacing* besteed, componenten kunnen user interface bouwblokken zijn, die tot een hele user interface samengevoegd kunnen worden.

# Gebruikte technieken en funderingen

Er zijn 3 belangrijke component modellen op de markt vandaag: het *Microsoft Component Object Model* (COM), de *Common Object Request Broker Architecture* (CORBA) en het *JavaBeans* model. Alle drie hebben ze overlappingen in functionaliteiten, maar eveneens zeer sprekende verschillen.

## Het Component Object Model

COM is een Microsoft component model. Het wordt gekenmerkt door het gebruik van *binaire interfaces* waardoor dit model zich grotendeels taal-onafhankelijk mag noemen. Het is een component model dat toch de meeste OO principes goed ondersteunt, behalve overerving (maar daar zijn andere oplossingen voor voorzien). Opvallend is het gebrek aan netwerk transparantie: er is locatie transparantie voor componenten onderling op een enkele computer op voorwaarde dat het Microsoft Windows platform gebrukt wordt. Tussen verschillende computers onderling is er, ondanks het RPC-protocol waarop COM communicatie steunt, geen netwerk communicatie voor COM mogelijk. Hieraan probeert men, zonder veel succes, een mouw aan te passen door middel van een uitbreiding van COM: *Distributed COM* (DCOM). Communicatie tussen verschillende componenten die op verschillende computers verblijven via een netwerk verbinding is nu wel mogelijk, maar van transparantie is nog altijd geen sprake. Pas bij de introductie van de *Microsoft Transaction Server* wordt er netwerk transparantie bijgevoegd.

Uitgebreid met MTS is COM ook in staat zich in een transactie te betrekken. Transactie beheer is een van de belangrijkste diensten die MTS aanbiedt. COM heeft (zonder enige extra's) ook een uitstekende ondersteuning voor versie beheer. Elke nieuwe versie van een component is eigenlijk geen nieuwe versie maar een totaal nieuw component. Oude versies worden "bevroren" en het is niet toegelaten nog wat bij te voegen aan de oude interface. Deze methode laat toch een naadloze integratie van nieuwe veries toe.

De beveiliging waar COM op steunt is grotendeels platform-afhankelijk en afhankelijk van het gebruikte communicatie protocol. Dit betekent dat Windows NT, welke een betere beveiliging voorziet als Windows 95, een betere kandidaat is om veilige componenten op te installeren dan

Windows 95. De RPC die DCOM gebruikt is verrijkt met verschillende beveiligingsniveaus en wordt *secure RPC* genoemd.

## De Common Object Request Broker Architecture

CORBA is het enige component model dat er in slaagt de voorheen gedefiniëerde richtlijnen tot op een redelijk niveau te benaderen. Het grote verschil met COM is dat CORBA een specificatie is en geen implementatie. CORBA gebruikt een Interface Definition Language (IDL) waarmee het de interface van componenten specificiëert. Door middel van deze interface definitie kan men elke taal gebruiken voor de implementatie waarvoor de desbetreffende CORBA *Object Request Broker* (ORB) een mapping voor voorziet. CORBA is dus grotendeels taal-onafhankelijk. CORBA ondersteunt zeer goed de OO principes door de IDL, inclusief overerving. Opmerkelijk is dat er veel CORBA mappings bestaan voor niet OO-talen.

Het gebied waarin CORBA uitblinkt is de transparantie die deze kan voorzien door middel van de ORB. Netwerk transparantie evenals migratie transparantie worden goed tot zeer goed ondersteund. De mogelijkheden om verschillende ORBs met elkaar te laten communiceren over een netwerkverbinding breidt deze transparantie nog uit over netwerken. Standaard wordt er ook een protocol voor inter-ORB communicatie over een TCP/IP connectie gedefiniëerd; het *Internet Inter-ORB Protocol* (IIOP), afgeleid van het *General Inter-ORB protocol* (GIOP).

Er zijn een hele hoop diensten beschikbaar voor CORBA, beschreven in [Obj98]. Spijtig genoeg is er geen echte versie-beheer, maar wordt de verantwoordelijkheid in handen van de ontwikkelaar gegeven. CORBA ondesteund door de service transacties, maar geeft geen garanties voor de ondersteuning van geneste transacties. Er is zelfs een service voor licentie beheer, welke interessant is naar de praktische implementatie die gepaard gaat met deze thesis.

Goede security definiëren in een specificatie is zeker geen gemakkelijke opdracht. Zeker als de specificatie niet mag steunen op de veiligheid van het praktisch gebruik van programmeertalen of van de gast-omgevingen. Een eerste vereiste voor CORBA security is de authenticate van componenten.

## JavaBeans

Het JavaBeans model is een component model van Sun, gebaseerd op Java (een programmeertaal). In feite is het JavaBeans model enkel een restrictie van de vorm van een klasse zoals die in Java kan ontwikkeld worden. Dit heeft als gevolg dat JavaBeans dezelfde OO kenmerken heeft als de taal Java. Een JavaBean wordt gedefiniëerd door Sun als [Ham97]:

> " een herbruikbaar software component welke visueel gemanipuleerd kan worden in een ontwikkelings applicatie."

Hiermee wordt de eerste intentie van JavaBeans duidelijk: vooral user interfacing, wat niet weglaat dat er ook andere mogelijkheden waren. Swing componenten, de Java user interface API, zijn in feite een collectie JavaBeans waardoor ze gemakkelijk bruikbaar zijn voor visuele manipulatie. Naarmate het JavaBeans model volwassener werd, werden er meer en meer mogelijkheden kenbaar van het model, en kent het nu veel meer toepassingen dan enkele user interface bouwblokken zoals plug-ins voor tekstverwerkers bijvoorbeeld.

# Een vergelijking van componenten

In de vergelijking van verschillende componenten kan er ten eerste opgemerkt worden dat de vergelijking over een gedeelde eigenschap van de componenten gaat zoals netwerk transparantie of security bijvoorbeeld. Verschillende component modellen als geheel is zeer moeilijk zoniet onmogelijk. Er zijn geen winnaars, er is geen "beste" component model, maar er kunnen wel winnaars zijn op bepaalde domeinen. Een nadeel is dat dit ook weer open voor discussie kan zijn, afhankelijk van het standpunt dat men ten opzichte van de eigenschap aanneemt

## ActiveX vs JavaBeans

Eerst worden twee component modellen naast elkaar gelegd die qua aangeboden diensten zeer goed op elkaar gelijken namelijk *ActiveX* en *JavaBeans*. ActiveX is een "herwaardering" naar het internet toe van het vroegere OLE2 van microsoft. Daarmee is ActiveX ook gebaseerd op COM (net zoals OLE2). ActiveX is meer een marketing term en heeft geen eenduidige defintie. ActiveX is meer een overkoepelende term voor technieken voor integratie van componenten en communicatie tussen componenten. ActiveX kan ingedeeld worden in 4 groepen:

- ActiveX controls
- ActiveX documents
- ActiveX scripting
- ActiveX server components

Er is enkel een zinnige vergelijking mogelijk tussen *ActiveX controls* en JavaBeans. JavaBeans is reeds geïntroduceerd in de vorige sectie.

We zien dat beide component modellen de richtlijnen opgesteld in het eerste deel wel heel hard afzwakken. Van netwerk transparantie komt niet veel in huis, maar daar heeft ActiveX een beentje voor omdat het registreerbaar is in de Windows registry. Beide componenten ondersteunen een "download-and-run" methode, zodat het component altijd lokaal is als het gebruikt wordt. Een voordeel van JavaBeans is dan weer dat het beter overdraagbaar is (binnen de beperking van de aanwezigheid van een virtual machine) naar andere platforms, terwijl ActiveX Microsoft gebonden is. JavaBeans is dan weer een taalgebonden model (Java), terwijl ActiveX door de COM fundering van verschillende programmeertalen gebruik kan maken. ActiveX wordt dan ook wel eens de *lijm* genoemd om een integratie tussen verschillende technieken mogelijk te maken.

Een opmerkelijk feit is het bestaan van *bridges* (bruggen) voor ActiveX en JavaBeans. Een ActiveX Control kan in een JavaBean gestoken worden en omgekeerd. ActiveX in een JavaBean steken kan met behulp van de Microsoft Java Virtual Machine en een JavaBean in een ActiveX control steken gebeurt door een COM wrapper te voorzien voor de desbetreffende JavaBean. Op deze manier kunnen ze simpel gebruikt worden in meerdere visuele programmeer omgevingen. Alleen het feit al dat deze bridges bestaan duidt erop dat beide component modellen complementair zijn ten opzichte van elkaar voor bepaalde eigenschappen.

### Microsoft Transaction Server vs Enterprise JavaBeans

*Middleware* modellen mogen niet uit de discussie weggelaten worden. Deze middleware modellen zijn de lijm, de middle tier, tussen de data tier en de client tier. Weerom zijn er twee belangrijke middleware modellen op de markt op dit moment, van Sun en Microsoft. Sun heeft *Enterprise JavaBeans* (EJB) op de markt gebracht: een op Java geënt middleware model dat *niet* gebaseerd is op het JavaBeans model. Microsoft heeft *Microsoft Transaction Server* (MTS) uitgebracht, welke, in tegenstelling tot EJB, zelf geen component model definiëert.

Het eerste en grootste verschil is dat Microsoft met MTS een kant-en-klare oplossing voorziet; een implementatie, terwijl EJB "slechts" een specificatie is. In de EJB specificatie zit zelfs iets wat men als bug zou kunnen beschouwen. Enerzijds wil men absolute overdraagzaamheid van EJB componenten garanderen tussen de verschillende implementaties, maar anderzijds vermeld men in de specificatie dat de producent van een EJB model vrij is om producent afhankelijke diensten te voorzien. Het is opvallend hoeveel het EJB model van MTS heeft afgekeken, beide architecturen zijn praktisch identiek.

EJB en MTS verschillen ook nogal wat in hun voorzieningen voor transacties: terwijl MTS de gebruiker veelal afschermt voor de complexiteit van transactie beheer bij componenten maar daarbij veel flexibiliteit verliest, gooit EJB alles open maar stopt dan weer veel verantwoordelijkheid bij de gebruiker. Transactie management is voor de ontwikkelaar dus moeilijker bij EJB dan bij MTS. MTS heeft dan weer het nadeel alleen met toestandsloze componenten te kunnen werken. MTS kan zo geen gebruikers specifieke componenten voorzien, terwijl EJB dit wel kan in de vorm van *session beans* die een toestand hebben.

## Beveiliging van componenten

Vanwege het belang van beveiliging wanneer men spreekt over het internet, wordt er een apart hoofdstuk voor gereserveerd. Om vanaf een goede basis te vertrekken worden eerst de bedreigingen en soorten aanvallen opgesomd, zodat er een idee kan gevormd worden waarmee men hier te maken heeft. De verschillende soorten aanvallen waarmee we te maken kunnen hebben zijn:

- afluisteren
- vermommen als een andere identiteit
- knoeien met berichten
- opnieuw gebruiken van berichten

We kunnen nu de bedreigingen van computersystemen in netwerken classificeren als:

- ongewenst veranderen van informatie
- ongeoorloofd verkrijgen van informatie
- stelen van syteembronnen
- vandalisme

ActiveX security heeft vooral te maken met het bewijzen van de herkomst (authenticatie) en juistheid (is ermee geknoeid?) van het ActiveX component. *Authenticode* is de naam van de techniek die dit voor ActiveX doet. Het probeert de afkomst met zekerheid te bepalen door middel van certificaten, en checkt de ActiveX code of er mee geknoeid is. Dit laatste doet het door code signing (een hash waarde van de code bij de code voegen waaruit kan afgeleid worden of er met de code geknoeid is). ActiveX heeft ook de notie van een *sandbox* (zandbak), dit zorgt ervoor dat niet vertrouwde componenten geen vrije toegang hebben tot systeembronnen. JavaBeans ondersteunt dezelfde beveiligingsprincipes als ActiveX: certificaten, code signing en sandboxing en doet dit door middel van de Java packaging technology: *JARs*. Meestal vraagt dat echter meer werk met Java van de ontwikkelaar (ook al omdat JavaBeans geen centraal beheer op een systeem kennen zoals ActiveX met de Windows registry). Java laat wel een gedetailleerde beveilingsconfiguratie toe zodat voor geïsoleerde componenten exact gedefiniëerd kan worden tot welke systeembronnen (zoals sockets, files,...) ze toegang krijgen en tot welke niet.

De specificatie voor de CORBA security service is een zeer uitgebreide, volledige specificatie die rekening houdt met authenticatie, toegangsrechten en controle, administratie, veilige communicatie en het met zekerheid identificeren van gebruikers. Omwille van de context van deze thesis (internet componenten) is het interessant om eens te bekijken hoe veilig CORBA communicatie tussen twee ORBS over een internet verbinding kan maken. CORBA gebruikt hiervoor een extra laag op IIOP, namelijk de *Secure Inter-ORB protocol* (SECIOP), bestaande uit twee delen: een *context management* laag en een *sequencing* (opeenvolgings) laag. De management laag communiceert met de transport laag en de sequencing laag is verantwoordelijk voor het veilig verpakken van de te verzenden frames. Hiernaast worden er nog 3 verschillende beveiligings niveaus vermeld in de specificatie in verband met authenticatie en deligatie van identiteiten. De SECIOP laag kan verschillende soorten protocols gebruiken, die elk hun eigen gewenste beveiligingsniveau kunnen instellen.

De door MTS ondersteunde veiligheid steunt op (weerom) het Operating System Microsoft Windows, meer bepaald Microsoft Windows NT. Een belangrijk beveiligingsprotocol dat voor authenticatie zorgt, maar geen delegatie van toegangsrechten toestaat, is NTLMSP (NT LAN Manager Security Provider). Dit protocol is in de laatste Windows versie (Windows 2000) vervangen door Kerberos. Kerberos is omslachtiger maar veiliger als NTLMSP. MTS heeft ook de mogelijkheid om packages van toegangsrechten te voorzien. Beveiliging bij EJB kan afhankelijk zijn van verschillende personen: de programmeur, degene die de packages installeert of degene die door middel van verschillende Beans een applicatie samenstelt (application assembler). Zoals bij MTS kunnen er toegangsrechten gespecifiëerd wordenvper component.

## Een praktische uitwerking: de licentie-module

Als praktische uitwerking van deze thesis werd gekozen om een component te maken dat in staat is om via een internet connectie een of meerdere licenties te checken. Er moet met verschillende bedreigingen rekening gehouden worden, en een oplossing voor een bedreiging kan een opening zijn voor een andere bedreiging. Het grootste probleem is dat de module zich aan de klantzijde bevindt en er niet altijd een internet verbinding beschikbaar is. Verschillende mogelijkheden met betrekking tot verschillende situaties worden bekeken en er worden enkele oplossingen voorgesteld, elk met voor- en nadelen. Na een grondige analyse van het probleem

werd gekozen voor Java als programmeertaal, vanwege de reeds aanwezige en handige APIs voor beveiliging en netwerkcommunicatie. Er werd gekozen om zelf certificaten te genereren per client van de module, geïnspireerd op X.509 certificaten. De module kan in een JAR verpakt worden, om zo een betere beveiliging te voorzien.

## Conclusie

Het internet staat nog altijd in de kinderschoenen, zeker op het vlak van component gebaseerde software ontwikkeling voor het internet. De meeste componenten gebruikt de dag van vandaag zijn "download-and-run" componenten en ondersteunen niet echt gedistribueerd gebruik. MTS en EJB hebben al verbetering in deze situatie gebracht, maar richten zich ook niet echt op het gedistribueerd gebruik van componenten en meer op middle-tier beheer. CORBA kan de vorm geven aan toekomstige oplossingen: de mogelijkheden om verschillende technieken samen te laten werken (er bestaan C++, C, Smalltalk, COM en Java mappings) en echte distributie te voorzien. Er is snellere internet communicatie nodig als men ten volle van CBSD voor het internet wil profiteren. Tenslotte is de beveiliging van componenten een belangrijke vereiste (vooral authenticatie van componenten) voor een gedistribueerd gebruik van componenten, maar dit wordt al goed ondersteund door de meeste component modellen. De meeste beveiligingsfouten zijn "menselijk" en gebeuren door slordigheid of nalatigheid. Beveiliging kan praktisch gezien ook niet perfect zijn, maar het moet tenminste een perfect veilige toestand proberen te benaderen.

# Bibliography


[A. 97] A. Menezes, P. van Oorschot, S. VanStone. *Handbook of Applied Cryptography.* CRC Press Inc., 1997. `http://www.carc.math.uwaterloo.ca/hac`.

[Ann98a] Anne Thomas. Comparing MTS and EJB. *Distributed Computing*, pages 52–54, December 1998.

[Ann98b] Anne Thomas. Enterprise JavaBeans Technology. *Patricia Seybold Group*, 1998.

[Ant97] Anthony Frey. Is dcom truly the object of middleware's desire? World Wide Web, `http://www.nwc.com/813/813r1.html`, 1997.

[Bar99] Bart Calder, Bill Shannon. *The JavaBeans Activiation Framework Specification.* Sun Microsystems, 1.0a edition, May 1999. `http://java.sun.com/beans`.

[Bil98] Bill Schneider. Comparing Microsoft Transaction Server to Enterprise JavaBeans, 1998. `http://www.microsoft.com/com`.

[Bru00] Bruce Eckel. *Thinking in Java, second edition.* Prentince-Hall Inc, 2000.

[C. 97] C. Szyperski and J. Gough. Special issues in object-oriented programming. In *ECOOP96 Workshop Reader*, pages 99–114. Alpha Books, Bristol, 1997. Workshop on Component-Oriented Programming, Summary.

[Cas89] Castle Hill, Cambridge England: Architecture Project Management. *The Advanced Network Systems Architecture (ANSA) Reference Manual*, 1989".

[Cha96] Charles Petzold. *Programming Windows 95.* Microsoft Press. Academic Service, 1996.

[Cle97] Clemens Szypersky. *Component Software.* Addison-Wesley, 1997.

[Dan98] Dan S. Wallach, Edward W. Felten. Understanding Java Stack Inspection. In *IEEE S & P'98*, May 1998.

[Don98] Don Box. *Essential COM.* Addison-Wesley, 1998.

[Dou92] Doug Lea, Marshall P. Cline. The Behavior of C++ classes. In *SOOPPA'92*, 1992.

[Eri95] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[Eri98] Eric Steegmans, Yves Willems. *Informatie- en programmastructuren.* Katholieke Universiteit Leuven , Wina, 1998.

[Geo94] George Coulouris, Jean Dollimore and Tim Kindberg. *Distributed Systems; Concepts and Design.* Addison-Wesley, 1994.

[Gop98] Gopalan Suresh Raj. A detailed comparison of CORBA, DCOM and Java/RMI (with specific code examples). World Wide Web, `http://gsraj.tripod.com/`, 1998.

[Gop99] Gopalan Suresh Raj. A Detailed Comparison of Enterprise JavaBeans (EJB) and The Microsoft Transaction Server (MTS) Models (with specific code examples). World Wide Web, `http://gsraj.tripod.com/misc/ejbmts`, 1999.

[Gre92] Gregor Kickzales, John Lamping. Issues in the design and specification of class libraries. In *OOPSLA92*, pages 435–451. Xerox Palo Alto Research Center, ACM press, 1992. ACM SIGPLAN Notices.

[Ham97] Graham Hamilton. *JavaBeans*. Sun Microsystems Inc., 1.01 edition, 1997. `http://java.sun.com/beans`.

[Jam96] James Goslin, Bill Joy, Guy Steele. *The Java Language Specification*. The Java Series, Sun Microsystems. Addison-Wesley, 1996.

[Jan99] Jan Van den Bergh. Distributed component systems, a comparison. Master's thesis, School voor Kennistechnologie,Universiteit Maastricht,Limburgs Universitair Centrum, 1999.

[Joh96] John Siegel. *CORBA, Fundamentals and Programming*. John Wiley, 1996.

[JPI99] Java Plug-In Scripting; Using JavaBeans with Microsoft ActiveX Components. World Wide Web, `http://java.sun.com/products/plugin/1.2/docs.script.html`, October 1999.

[JSF96] Marianne Mueller J. Steven Fritzinger. Java security, 1996.

[KK98] Peter Petersen Kåre Kjelstrøm. Components and development tools. Master's thesis, ?, 1998.

[lab99] Nova laboratories. *The Developer's Guide to Understanding Enterprise JavaBeans Applications*. Nova Laboratories, 1999. `http://www.nova-labs.com`.

[Lau98a] Laurence P.G. Cable. *A proposal for a Drag and Drop subsystem for the Java Foundation Classes*. Sun Microsystems, final draft 0.96 edition, 1998. `http://java.sun.com/beans`.

[Lau98b] Laurence P.G. Cable. *Extensible Runtime Containment and Services Protocol for JavaBeans Version 1.0*. Sun Microsystems, 1.0 edition, December 1998. `http://java.sun.com/beans`.

[Lis99] Lisa Friendly, Mary Campoine, Kathy Walrath and Alison Huml. *The Java Tutorial*. Sun Microsystems, 1999. `http://java.sun.com/docs/tutorial`.

[M. 90] M. Burrows, M. Abadi and R. Needham. A logic of authentication. In *ACM Transactions on Computer Systems*, pages 18–36. ACM, February 1990.

[M. 93] M. Abadi, M. Burrows, B. Lampson and G.D. Plotkin. A calculus for access control in distributed systems. In *ACM Transactions on Programming Languages and Systems*, volume 15, pages 706–734, September 1993.

[Mai98] Mai-lan Tomsen. Build reliable and scalable N-tier applications that run on both Windows NT and Unix. *Microsoft Systems Journal*, 1998.

[Mar97] Mary Kirtland. Object Oriented Software Development Made Simple with COM+ Runtime Services. *Microsoft Systems Journal*, November 1997.

[Mar99] Marco Pistoia, Duane F. Reller, Deepak Gupta, Milind Nagnur and Ashok K. Ramani. *Java2 Network Security*. IBM, 1999. `http//:www.redbooks.ibm.com`.

[Mic97] Microsoft Corporation. Deploying activex controls on the web with the internet component download, April 1997.

[Mic99] Microsoft Corporation. *Microsoft Developer Network*, 1999. `http://www.microsoft.com/msdn`.

[Obj98] Object Management Group. *CORBAservices: Common Object Services Specification*, December 1998. `http://www.omg.org`.

[Obj99] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, October 1999. `http://www.omg.org`.

[Org92] International Standards Organization, editor. *Basic Reference Model of Open Distributed Processing, Part 1: Overview and guide to use*. International Standards Organzation, 1992.

[P. 97] P. Emerald Chung, Yennun Huang, Shalini Yajnik, Deron Liang, Joanne C. Shih, Chung-Yih Wang and Yi-Min Wang. DCOM and CORBA Side by Side, Step by Step, and Layer by Layer, 1997. `http://www.bell-labs.com/emerald/dcom_corba/Paper.html`.

[Pau92] Paul R. Wilson. Uniprocessor Garbage Collection Techniques. In *International Workshop on Memory Management*. Springer-Verlag lecture notes in Computer Science series, Springer-Verlag, September 1992.

[Pau95] Paul Stafford, Joe Powell. *COM: A problem solver. Binary standard lays foundation for component software*, 1995.

[Pet98] Peter Müller. Use of pre- and postconditions; using conditions to prove correctness, 1998.

[Rob98] Robert Orfali, Dan Harkey. *Client/Server Programming with Java and CORBA, second edition*. Addison Wesley, 1998.

[San96] Sanders Kaufman, Jr., Jeff Perkins and Dina Fleet. *Teach yourself ActiveX Programming in 21 days*. Academic Service, 1996.

[San99] Sansjeev Krishnan. *Enterprise JavaBeans to CORBA mapping*. Sun Microsystems Inc., 1.1 edition, August 1999.

[Sar94] Sara Williams , Charlie Kindel. The Component Object Model, a technical overview. *Dr. Dobb's Journal*, December 1994. the newsstand special edition.

[Ses00] Govind Seshadri. Fundamentals of RMI, Short Course. `http://www.jGuru.com`, 2000.

[Sø97] Søren Brandt. *Towards Orthogonal Persistence as a Basic Theory.* PhD thesis, Aarhus University, Computer Science Department, February 1997.

[Ste97] Steve Vinoski. Corba: Integrating diverse applications within distributed heterogeneous environments. In *IEEE Communications Magazine*, volume 35, February 1997.

[Sus98] Susie Adams. Using ActiveX and Java Applets together. *Active Server's Developer Journal*, 1998.

[The98] The Java Language Team, JavaSoft. About microsoft's "delegates". World Wide Web, `http://java.sun.com`, September 1998.

[Tre] Trevor Harmon. Java SDK 2.0: A Two-Way Bridge between ActiveX and Java. World Wide Web, `http://www.informant/com/ji/jinewupl.htm`.

[Vla99] Vlada Matena ,Mark Hapner. *Enterprise JavaBeans Specification, v1.1.* Sun Microsystems Inc., 1.1 edition, 1999. `http://java.sun.com/ejb`.

[Zan97] Zane Lang. *ActiveX All In One, a Web Developer's Guide.* The Open Group (X/Open - OSF), 1997.

# Glossary

| | |
|---|---|
| ACID | Atomicity, Consistency, Isolation and Durability |
| ANSA | Advanced Network System Architecture |
| API | Application programming Interface |
| ASP | Active Server Pages |
| AWT | Abstract Window Toolkit |
| BDK | Bean Development Kit |
| CBSD | Component Based Software Development |
| CGI | Common Gateway Interface |
| CICS | Customer Interface Control System |
| COM | Component Object Model |
| CORBA | Common Object Request Broker Architecture |
| CSI | Common Secure Interoperability |
| DCE | Distributed Computing Environment |
| DCOM | Distributed Component Object Model |
| DDE | Dynamic Data Exchange |
| DES | Data Encryption Standard |
| dll | Dynamic Link Library |
| DND | Drag-and-Drop |
| DNS | Domain Name Service |
| EJB | Enterprise Java Beans |
| GIOP | General Inter-Orb Protocol |
| GUI | Graphical User Interface |
| GUID | Globally Unique Identifier |
| HTML | HyperText Mark-up Language |

| | |
|---|---|
| IC | Integrated Circuit |
| IDL | Interface Definition Language |
| IE | Internet Explorer |
| IIOP | Internet Inter-ORB Protocol |
| IOR | Interoperable Object Reference |
| IPX | Internet Package Exchange |
| ISAPI | Internet Server Application Programming Interface |
| JAF | Java Activation Framework |
| JAR | Java Archive |
| JDK | Java Development Kit |
| JFC | Java Foundation Classes |
| JNDI | Java Naming and Directory Service |
| JRE | Java Runtime Environment |
| JRMP | Java Remote Method Protocol |
| LAN | Local Area Network |
| MFC | Microsoft Foundation Classes |
| MTS | Microsoft Transaction Server |
| NTLM | New Technology LAN Manager |
| NTLMSP | New Technology LAN Manager Security Provider |
| OLE | Object Linking and Embedding |
| OMA | Object Management Architecture |
| OMG | Object Management Group |
| OO | Object Oriented |
| OOP | Object Oriented Programming |
| ORB | Object Request Broker |
| OSF | Open Software Foundation |
| OWL | Object Window Library |
| pdf | portable document format |
| PIM | Personal Information Manager |

| | |
|---|---|
| RMI | Remote Method Invocation |
| RPC | Remote Procedure Calling |
| SCM | Software Component Manager |
| SECIOP | Secure Inter-ORB Protocol |
| SHA | Secure Hash Algorithm |
| SQL | Structured Query Language |
| SSPI | Security Support Provider Interface |
| TCP/IP | Transmission Control Protocol/Internet Protocol |
| UI | User Interface |
| US | United States |
| UUID | Universally Unique Identifier |
| VB | Visual Basic |
| VBScript | Visual Basic Script |
| WFC | Windows Foundation Classes |

# Index