Tools for Distributed Software

Tommi Lukkarinen

Tampere university
Department of Computer Science
Master's thesis
June 2000

Abstract

Tampere University
Department of Computer Science
Tommi Lukkarinen:
Master's thesis, 81 pages

June 2000

This book is a master's thesis on distribution technologies. This book concentrates on the common distribution technologies and design and architectural possibilities using those technologies. I have attempted to cover as much width as possible and only show advantages of technologies in some depth. A practical view is taken to show advantages on using each technology.

Technologies that are taken in the view are Transmission Control Protocol, Hypertext Transfer Protocol, Common Gateway Interface, Common Object Request Broker Architecture, Component Object Model, Remote Method Invocation, Remote Procedure Call, COM/CORBA Interworking and RMI over IIOP.

This book enlightens the reader not by introducing one technology in depth but by showing several of them together. It is shown that different distribution technologies have advantages in different application areas and that they are good for different kinds of purposes. It is also shown that these technologies can and even might be preferred to work together.

Acknowledgements

I was funded for about six months by Nokia to do this book. During that time I did my initial research and finally the actual subject started to form in my mind. I also almost got the book finished, but ended throwing it to trash can as the subject changed. ICL Data, currently ICL Invia, funded finishing this book.

Thanks to Jyrki Nummenmaa and Roope Raisamo for support and guidance. Thanks for patience after the first book went its way.

Abbreviations

Abbreviation	Meaning	Explanation at page
ASP	Active Server Pages	12
CGI	Common Gateway Interface	12
COM	Component Object Model	14
CORBA	Common Object Request Broker Architecture	14
DCE	Distributed Computing Environment	13
DCOM	Distributed COM	16
DII	Dynamic Invocation Interface	29
DLL	Dynamic Linking Library	45
DSI	Dynamic Skeleton Interface	29
ESIOP	Environment Specific Inter-ORB Protocol	25
GIOP	General Inter-ORB Protocol	24
GUID	Globally Unique Identifier	36
HTTP	Hypertext Transfer Protocol	13
IDL	Interface Definition Language	23
IFR	Interface Repository	29
IIOP	Internet Inter-ORB Protocol	24
IOR	Initial Object Reference	75
IP	Internet Protocol	12
JLIM	Java Language to IDL Mapping – not standard abbreviation	57
JNI	Java Native Interface	45
JSP	Java Server Pages	14
MIDL	Microsoft IDL	35
MTS	Microsoft Transaction Server	35

Abbreviation	Meaning	Explanation at page
ODL	Object Definition Language	35
OLE	Object Linking and Embedding	37
OMG	Object Management Group	16
ONC	Open Network Computing	15
ORB	Object Request Broker	16
OSF	Open Software Foundation	15
OV	Objects by Value – not standard abbreviation	47
RMI	Remote Method Invocation	17
RPC	Remote Procedure Call	15
TCP	Transmission Control Protocol	12
TLB	Type Library	36
URL	Uniform Resource Locator	21

Table of contents

Table of Figures

1 Introduction to the Book

This book is a catalogue of popular network programming tools. You will also find advice on what the tools are good for and a few examples, which show the power of these tools. Emphasis rests on communication enabling Internet tools - middlewares.

You should have prior practical knowledge of at least one middleware, like CORBA or COM, as I did have when I started studying the subject. You should read this book if you want to learn about other middlewares and distribution tools and the reasons why you should consider using them. I have always thought that real people will read this book besides those of academic interest and that is the reason I have included trivial matters in beginning chapters.

First all the subjects are more thoroughly introduced. The introductory part makes things easier to understand as some things are said twice and it also shows the organisation of the book. Book goes first to the basics of distribution explaining about protocols. Protocols are important to understand when different borders need to be crossed like performance limits and security locks. Protocols are left to little attention, as they are easiest to understand and to learn. Protocols are therefore presented only in the introductory part. Chapter 2 contains the introduction.

In chapter 3 the concept of middlewares is explained along with most popular middlewares. Middlewares are in the focus of this book, protocols in the root of middlewares and bridges serving between middlewares.

Chapter 4 deals with bridges. Bridges are a kind of an advanced topic. Bridges are middlewares between middlewares. Bridge is an extra layer that brings compatibility to distributed programming.

Chapter 5 has examples. I have this abstract idea that when someone makes a tool she is thinking of some task the tool will be used for. Although some other tool may have more effort put into and be better designed, it is probably not as good for that specific task unless developers were thinking the same task. My examples reflect this thought, they show the ideas I had about some technology and its strengths.

Chapter 6 has my conclusions, in which I promote advantages of knowing the field of distribution thoroughly.

2 Introduction to Distribution

In a way distribution is a relatively old thing, but it is also developing every day. There are every day more applications using distribution and I believe that relatively applications that are distribution enabled are increasing in number related to applications that are not distribution enabled. General progress brings out more tools to distribution with superior features compared to the old systems.

Tools are a good acronym for the field distribution. There are tools for basic communication, specific kinds of communication, abstract communication, software integration, service grouping and all kinds of things. Prolonged stay in the field makes quite certain that one will always have to learn more, even if the known tools do not get outdated.

This is an introductory chapter where subjects are taken a quick look at before a more thorough study. Things that are taken only a quick look at are shown only here.

In this book distribution tools are divided to protocols, middlewares and bridges. There is a quick look at protocols, a longer look at middlewares and related tools and a mediocre look at bridges. Each group has an introductory section and middlewares and bridges have their own sections.

Protocols are language and platform independent specifications, which focus in the transmission of data.

Middlewares are language specific, although there can be several languages, and platform independent specifications, which focus on the distribution of functionality.

Bridges are middleware specific and platform independent specifications, which enable middlewares to distribute functionality over middleware borders.

Each group has more tools aiding the main tools. Tools to integrate with other technologies, components with general functionality and tools to automate software engineering process.

Protocols are discussed in the following order: Transmission Control Protocol, Hypertext Transfer Protocol and Common Gateway Interface.

Middlewares are discussed in order: Remote Procedure Call, Common Object Request Broker Architecture, Component Object Model and Remote Method Invocation.

Bridges are discussed in the following order: COM/CORBA Interworking, RMI over IIOP.

2.1 Protocols

Protocols are discussed here because middlewares are built on top of protocols. Often middleware applications will have to monitor the protocol level and the protocol level causes often problems. Protocols are also discussed because some things can be done as well with simple protocol level as on middleware level.

There are quite a few protocols but it was easy to limit the set as I took only the ones that are strongly associated with middlewares. Transmission Control Protocol is the base protocol under middlewares. Hypertext Transfer Protocol and Common Gateway Interface protocol are associated with firewall penetration.

Strength of protocol over middlewares can be derived from my previous statement – protocols focus on the transmission of data, middlewares focus on the distribution of functionality. If the main feat of the system is on pure data transmission, preferably of simple types, protocols could be considered.

Smaller infrastructure, simplicity, availability and fine control are advantages of protocols. I have seen protocols in use even in new designs, namely in non-standard platforms where no advanced distribution mechanisms have been implemented.

2.2 Transmission Control Protocol

Transmission Control Protocol (TCP) [Internet RFC 793] is the base communication protocol for most of the Internet technologies. It is a streaming protocol, which is simple and straightforward to use. With a streaming protocol reading and writing are done continuously regardless of data packaging. Most modern operating systems have built-in TCP services/libraries, offering a possibility to use TCP by pre-made libraries and simplifying application building. TCP can be used to build a distributed system on its own and other distribution technologies are usually built on top of it.

TCP is a connection based protocol, meaning that once the connection is established, data can be transferred and received through a pipe without any care of internet addresses or such. TCP is built on Internet Protocol (IP), although it is not restricted to it. In a way IP takes care of the low-level data transmission and addressing and TCP takes care of the data representation.

One of the good sides of TCP and other connection based protocols is that once connection is established it can be assumed that the service offered through the TCP connection is valid as long as the connection holds or a service provider signals otherwise. Connectionless protocols do not know about service unavailability until it is used.

TCP loses to the higher level protocols and distribution systems built on these higher level protocols in that to achieve complex information exchange, protocols will have to be built on top of TCP. If a TCP connection would be simply used to transfer byte arrays of the same kind from server to client, it would serve its purpose better than the higher level protocols. But if the byte arrays would differ in type, there would indeed be a need for a higher level protocol, which would define what kind of an array is in transfer. If there would be other things transferred beside arrays, and the direction of transfer would vary, and there would still be various clients and servers in the system, there would be a point in raising the question if it was time to use component architectures.

As TCP is the de facto standard on Internet communication, services on it have advanced in leaps. There are object serialisation and encryption services that work on top of TCP. As these services work as transparently as the ordinary TCP implementations, higher level protocols and component technologies can take advantage of them.

2.3 Hypertext Transfer Protocol

Hypertext Transfer Protocol (HTTP) [Internet RFC 2068] works over TCP. HTTP is a protocol designed for accessing Internet content. It is not restricted to TCP or to Internet, though. HTTP is a response/request protocol working on Internet addresses (Uniform Resource Locators). Usually HTTP is used to access Hypertext Mark-up Language (HTML) documents.

HTTP defines methods used to access and manipulate documents like GET, DELETE and POST. These methods like the other information in HTTP requests is sent as plain text from client to the server and response also is returned as plain text. Text is parsed in both ends to get meaningful responses for the user or the software.

The two things usually available for the Internet user are HTTP connections (surfing WWW pages) and mail. Some Internet service providers do not allow other connections outside because of security or commercial reasons and companies often have firewalls preventing other kinds of communication. Mail hardly works at real time and therefore can not really be considered for real-time distributed programming. HTTP also usually works trough proxies complicating things a little. Imagine a time service and that time being stored in a proxy for future uses. Regardless, HTTP request will generally be handled immediately and almost all of the Internet connections allow them.

Availability of HTTP makes it an ideal choice for any real time Internet communication, including distributed application communication needs. Application designer will face other problems, like to bend the code to use HTTP and to extract client requests from the WWW server, but at least the connection becomes theoretically possible.

Distribution technology implementations sometimes support the use of HTTP on the communication, taking the burden away from the programmer. Some WWW server implementations nowadays understand direct programming technologies like Java Servlets and Microsoft Active Server Pages (ASP). Both would still leave an extra layer to the server component unless it was made with an ASP compliant scripting language or with Java.

2.4 Common Gateway Interface

Common Gateway Interface [Coar et. al.] scripts are executable programs. These scripts can be executed for example from an HTML page using HTTP or directly through HTTP. A client posts the data to a web server that executes the CGI script with data as a parameter. Therefore CGI scripts are resources addressable by URLs.

CGI can be implemented with different languages; it is up to the web server to use the right tools to read the script. A CGI script itself can start a C or a Java program in the server. For example, a CGI script can be conventional shell script, which in turn can start other programs. CGI is often used just to pass parameters to actual software that processes the data and returns results. Results are often returned as a generated file that can be read by client.

A script could generate web pages without parameters; a CGI script could generate all ordinary Internet pages. Usually CGI scripts can be found processing web page forms. MS Active Server Pages (ASP), Java Server Pages (JSP) and Java Servlets are probably better suitable for simple web page generation.

In effect, CGI is the way to publish software that is accessible in Internet regardless of firewalls. CGI can be used by distribution technologies to pass firewalls and it is used through the HTTP. I believe it is still relevant technology, but maybe not the choice if it is used for the usual: generating different kinds of web pages.

It is also not necessarily natural just to make all distributed systems work through HTTP and CGI to make them work anywhere and anytime. Extra layers make the calls slower by at least an order of magnitude: the web server accepts the request, parses it, invokes a CGI script, which invokes the process instead of just directly invoking the process waiting for requests.

2.5 Middlewares

Previous sections deal with basic distribution technologies that offer the ground to build upon. Following chapters discuss advanced tools, middlewares. What kind of tools are these? With the exception of Microsoft Component Object Model (COM), all are based on public specifications. Public specification allows anyone to build an implementation of

the tool. Most clearly this can be seen with the Common Object Request Broker Architecture (CORBA) that has a dozen of existing implementations.

The purpose of these tools is to make distributed software building fast and easy. Generally this means that distribution is made transparent: an application developer can make the software just as if it was using a single platform; the tools hide the difficulties of distribution. Besides doing distribution fast and easy in different ways these tools have different histories and different sources making them different; like RMI was made to be the distribution technology of the Java platform, hindering its use cross platforms, but making it extremely easy to use with Java.

It is good to remember that although some of the middlewares may seem more attractive each of them has their own sides. Also as each of these middlewares is a specification, true evaluation has to be made with actual implementations.

Here is an attempt to describe these middlewares in few words. Remote Procedure Calls (RPC) are the first of these middlewares. CORBA was made as a joint venture of several software companies; this venture is called Object Management Group (OMG). CORBA implementations are usually used on Unix platforms. CORBA looks very much like RPC. COM was built as the distribution technology for Windows. COM is built upon RPC, but the practical use has nothing to do with RPC. Remote Method Invocation (RMI) is built as the distribution technology for Java. RMI is by the far the easiest to use, but is not so popular as COM or CORBA. RPC has the honour to start the introductions as the forerunner.

2.6 Remote Procedure Call

One should say Remote Procedure Calls, as there are two widely used RPCs in the software industry. The more advanced RPC is a part of the Distributed Computing Environment [OSF 1996] and is called henceforth DCE RPC. The other RPC [Internet RFC 1831] is the first widely used middleware and its specifications are nowadays taken care by Open Network Computing – therefore ONC RPC.

There are Windows and Unix RPC implementations and they are widely used. It is easy to say that newer distribution technologies have been made with RPC as the example, there are so many common features – and truly, Distributed Component Object Model (DCOM) has been made on top of DCE RPC. RPCs can hardly be recommended for use, as even the DCE system today wields CORBA instead of RPC.

Although support for RPC is likely to continue as there are lot of systems that still need the services, it is unlikely that the RPC will be developed further like CORBA, COM and RMI will be. A software developer might find herself in a situation where she has to switch middlewares as the old one does not support the new features.

2.7 Common Object Request Broker Architecture

CORBA [OMG 1998 b] is a joint project of dozens of software vendors. It is an attempt to make a standard distribution technology. Software vendors have striven to make their CORBA implementations interoperable, so it is even possible to make two different CORBA implementations work together. There are lots of CORBA implementations, so in reality interoperability will have to be checked from product to product and from feature to feature. The organisation specifying CORBA and related technologies is called Object Management Group (OMG).

CORBA follows object oriented approach closely and is being developed in interests that are seen as good modern design principles. This is supposed to mean that CORBA is a tool for large and robust systems. CORBA is in fact being developed all the time in contrast to technologies presented before, which are more or less standardised already.

CORBA is gaining width; it is covering more approaches to system development, like graphical component development. It seems like common playground, on which everyone can bring her own toys. As long as the toys somehow fit to the common picture of course. Constant growth of specifications defining CORBA makes it increasingly hard for CORBA vendors to keep their products up to date. To me it seems that each vendor brings his own ideas to the specifications, implements them in their own products, and does not necessarily take any concern on many other specifications. This means that CORBA implementations are diverting into different directions, serving different purposes.

2.8 Component Object Model

Component Object Model [Microsoft 1995] is a Microsoft product allowing distributed programming. COM means software component distribution inside a single machine and Distributed COM means component distribution across machine boundaries. DCOM has been shipped with every Microsoft operating system since Windows 95. Windows 95 also has an extension that supports DCOM. COM is a standard mechanism for in machine distribution for all Windows systems after Windows 3.1.

COM is a mechanism for invoking out of process and in process components. Generally libraries used with COM are included in the process and libraries embedded in executables are invoked after launching the executable out of process. As much as the COM is the implementation allowing distribution it is all the development tools Microsoft has built around it. COM without Microsoft Developer tool family is complex and difficult to learn, and these tools without COM lose much of their power. For an example COM allows the use of Excel functionality inside applications.

Strong sides of COM are that it is virtually on every PC and that it makes available a lot of implementations on those computers – as long as those computers have a Windows operating system and Microsoft Office applications are installed. It is quite easy to see advantages of COM when application programming is done with Microsoft tools like Visual Basic, Visual C++ or Visual Java. These tools automate COM development, dozens of components become readily into use.

On the other hand COM is not necessarily a strong choice on Unix platforms, where support has so far remained limited. There is even a port on Linux, though [Foley 1997]. Currently COM is seen as a rapid application development tool for Windows – as an office application development tool. Whether it could or should be used on mainframe systems, in which Unix and clones hold domination, is a separate thing from the public image of COM.

2.9 Remote Method Invocation

RPC was developed before CORBA and CORBA was before developed Remote Method Invocation [Javasoft 1998]. While RPC and CORBA have a lot in common, in the same way RMI and CORBA have a lot in common.

RMI is in my opinion by far the easiest choice for a beginner to start in the world of distribution. RMI can afford to be simplest as it works only on Java platform and as it had the advantage to be developed latest of the technologies presented here. Difference can be seen in lines, which are needed to enable distribution, in steps to be taken to create distributed objects, and in clarity. RMI follows Java programming conventions closely and is easy to learn for anyone who has worked with Java before.

Sun, the primary Java vendor, has also been working to make RMI and CORBA to work together. It remains to be seen if these two integrate and what kind of parts they have in new architectures. RMI still has a place in Sun software portfolio

2.10 Bridging Technologies

CORBA, COM, RPC and RMI are not cooperative with each other. Having similar technologies does not help when protocols and architectures are different. Still there is a need for interworking as it would be a waste to write same components again and again for different systems. Bridges allow building of a system using different middlewares.

Bridges are pretty straightforward to build as middlewares have a lot in common. Object oriented approach, C language like types and same kind of infrastructure bring middlewares quite close to each other. Main trends generally in software industry seem to be in the same direction; variety is created in smaller decisions.

Middleware bridges come in varieties – there are static bridges and dynamic bridges, there are bridges working in one direction and there are bi-directional bridges, there are

bridges as separate components and there are bridges embedded into a server and a client.

Software systems are expensive to build and it could be uneconomical if a large system would have to be rebuilt to be interoperable with an another one. Some technologies also fit better for other things and other technologies for other so it would make sense to build one system using two or three different distribution technologies.

2.11 COM/CORBA Interworking Specification

CORBA 2 was focused to get CORBA implementations working together. CORBA 1 did not guarantee any kind of operability between different CORBA implementations. A suitable addition to the specification was a COM/CORBA Interworking Specification. The specification has fitted into the actual CORBA specification [OMG 1999 b].

Actions of OMG seem to expect that COM and CORBA will be the middlewares to be used in the future.

In that sense the interworking specification is more important than for example CORBA services – if the organisation of documents can be seen as description of importance. COM and CORBA are dominant middlewares and therefore it is natural that also interworking implementations are mainly for these two.

COM/CORBA interworking specifies type mapping; error handling and matching of object oriented features. Implementation of the bridge is left completely open, except for that interworking with different CORBA implementations is assured.

2.12 RMI over IIOP

There aren't too many differences between RMI and CORBA. Both handle remote references in a similar fashion. CORBA IDL is mapped to Java interfaces in the same fashion as RMI interfaces are. The main difference is in that RMI allows transportation of real objects instead of just object references.

RMI over IIOP means that RMI software is developed as before, and the generated proxies simply use CORBA protocol in place of RMI native protocol. In the other side of network there can be another RMI over IIOP application or CORBA application using extensions. Extensions are needed in CORBA to enable object transfer by actual value rather than reference.

RMI over IIOP is based on two OMG documents: 'Java Language to IDL Mapping' [OMG 1998 d] and 'Objects by Value' [OMG 1999 a].

3 Middlewares

The concept ‘middleware’ covers really all kinds of three-tier like architectures. I think a better way to name the subject here would be ‘distributed component technologies’ – but ‘middleware’ is shorter and shorter is better. Middleware allows merging of functionality in physically separate locations by making remote function calls and object invocations transparent in the eyes of the developer.

Most of the middlewares are object oriented, like Component Object Model, Remote Method Invocation and various Common Object Request Broker Architecture implementations. There are also non-object oriented middlewares – Remote Procedure Calls. Middlewares use the proxy pattern [Gamma et. al. 1994] in implementing the transparency of remote invocations.

The whole idea of middlewares is to make software development easier and faster. In many cases middlewares offer a solution to a design problem, ease system designs and makes systems more understandable.

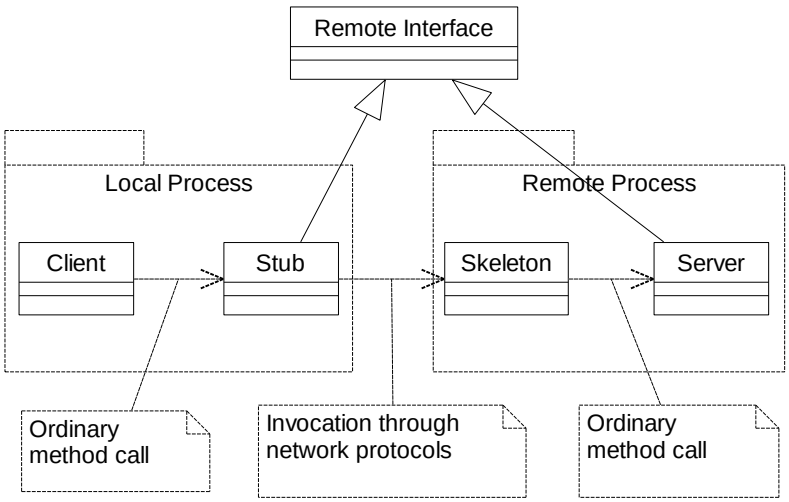
Middlewares also offer a totally new approach to software design. As middlewares allow component size abstractions as the common building block, abstraction level can be raised from object level to a step higher. High abstraction levels are commonplace in large architectures, but middlewares make that desirable even in smaller applications.

This chapter first introduces the proxy pattern [3.1] on which middlewares are based and then goes through common middlewares. Middlewares handled here are ONC RPC [3.3.1], DCE RPC [3.3.3], CORBA [3.4], COM [3.5], RMI [3.6] and Jini [3.6.5].

3.1 Proxy Pattern

Patterns are object oriented design solutions [Gamma et. al. 1994]. The idea of patterns is that they are design solutions that have been used and proven good in that use. Design patterns are basic solutions, to be used in everyday programming. There are also larger scale design patterns like CORBA Design Patterns [Mowbray & Malveau, 1997], which have solutions for distribution systems.

Proxy pattern is a solution to a problem, where communication can not be done towards the target with a standard way, such as simply calling the implementing function. Such situation could arise when a database is used – an application is written with C, but the database must be used with SQL. If the database stores an object, it would ease the design for the database client to be able to use the data as an object. The solution is to build a mediator, which acts like an ordinary object, but actually keeps its state in the database. The same solution can be used in remote applications, and it becomes an attractive choice, when middleware takes care of the work.



1. The Proxy Pattern

Figure 1 shows the proxy pattern used in remote communication. A client calls an object that implements the services the client needs. Without the client needing to be aware of it, stub (client side of the proxy) forwards the call through the network to a skeleton (server side of the proxy). The skeleton calls the actual implementing object as if it is the client.

If middleware tools have been built properly, the task of building a proxy can be the same as if services had been in the same process as a client. One of the criteria on which to evaluate middlewares in my opinion is how transparent they can make the existence of a proxy.

3.2 Middleware Infrastructure

In a single process system objects are created with some kind of a new operator and stored for a later use for instance in a container. Remote objects have their own processes or are created inside some other process.

If a process is to be activated by an action of a remote invocation, the middleware needs an agent on the local host. Remote invocation calls this agent, which starts up the target and returns a reference to the caller. Even if the target process has been started before the client starts to make invocations, the use of the agent is normal.

These middleware object reference handlers are usually called directory services or naming services/registries. Remote object skeletons are registered to them, either by code address or by network address.

Besides naming services middlewares usually have security services. Security services usually make ordinary user/password checks.

These two services make up the minimal middleware runtime infrastructure. These services should be as transparent to the user as proxy generation. Security can be made transparent by integrating the remote object user identity with the ordinary network identity [OSF 1996]. A naming service can be made transparent with a global resource directory [Javasoft 1999] – like with Uniform Resource Locators (URL).

Often a middleware infrastructure has a dozen different helper components – some transparent, some used as needed.

3.3 Remote Procedure Call

There are two middlewares with the name Remote Procedure Call (RPC). Open Network Computing (ONC) RPC originated from Sun Microsystems and is now based on three Internet standards: RPC [Internet RFC 1831], RPC Binding [Internet RFC 1833] and External Data Representation Standard (XDR) [Internet RFC 1832]. Distributed Computing Environment (DCE) [OSF 1997] RPC has been developed and is being updated by Open Software Foundation, currently Open Group.

ONC RPC makes distributed computing possible using the XDR protocol, which takes care of bit conversions. The RFC RPC Binding defines security measures for ONC RPC. ONC RPC was the first de facto standard for distributed computing which allowed distributed interfaces. ONC RPC is best known for Network File System (NFS) [Internet RFC 1094], which is built upon RPC.

DCE is a package of middlewares of which RPC is the most important part. Other parts in DCE are directory services, a security service, threads and a time service and a file service. Open Group, which develops DCE, is a consortium of IT vendors. Open Group is currently being involved with Object Management Group in an effort to integrate DCE and Common Object Request Broker Architecture [OMG 1999 b].

DCE is developed on fruits seeded by ONC RPC, but these two middlewares are not related to each other. These two technologies are out of date as they have little support for object oriented approach. There are still lots of implementations for both DCE RPC and ONC RPC and both have commercial support.

3.3.1 Open Network Computing Remote Procedure Call

ONC RPC is a straightforward and simple middleware specification. RPC/XDR interface definition written by a developer is generated into stubs and skeletons to be used in the client and the server. The stub and the skeleton take care of remote communication and XDR conversion of function calls in the same principle as the proxy pattern [Gamma et. al. 1994]. ONC RPC is not object oriented and does not allow pointers to be passed over remote invocations.

XDR is not a programming language but a data representation language. XDR types are integer, unsigned integer, enumeration, boolean, hyper integer, hyper unsigned integer, floating-point, double-precision floating point, quadruple-precision floating-point, fixed-length opaque data, variable length opaque data, string, fixed-length array, variable-length array, structure, discriminated union, void, constant, typedef and optional-data. XDR is a C type language.

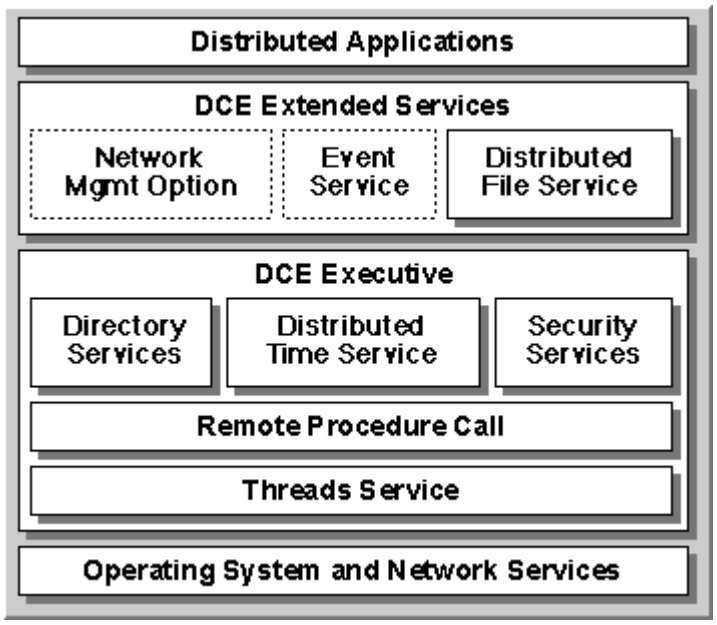
RFC does not specify how messages are transported but how messages are interpreted. Therefore RPC is transport independent. As RPC is transport independent, two applications built with RPC are not necessarily interoperable. RPC RFC does define how to pass data to lower level protocols, so two RPC implementations built for example over TCP would be interoperable.

RFC does not specify how to map XDR into some programming language. RPC understands the concept of a method call and a method argument. What method arguments mean in the programming language, is up to a RPC vendor. In Distinct software vendors [Distinct 1999] implementation there is support only for Java compatible types.

ONC distributed component recognition is done with unique numbers. Each closed environment can have its own set of component numbers and Sun Microsystems holds a registry for public component numbers.

3.3.2 Distributed Computing Environment

DCE is composed of several different components. Components directly related to RPC are Directory Services, Distributed Time Service, Security Services and Thread Service. Extended Services are components to be used as their own, built on top of RPC and DCE Executive components. Of Extended Services Network Management Option and Event Service have not been really used.



2. DCE architecture [Brando 1995]

Directory Service stores references to RPC components. Directory Service is a gateway between RPC components. RPC server makes a reference in the Directory Service. RPC client looks up the server from the Directory Service. Afterwards Directory Service is no longer needed by the client or by the server, unless new connections are needed.

Distributed Time Service keeps up synchronised time in the DCE system. DCE system is divided into cells, all connected cells have synchronised time when Distributed Time Service is active.

Security Service takes care of authentication of RPC calls. As Security Service is common in the whole DCE environment, distributed components do not need a separate authentication system – system user profiles can be configured on which distributed calls are allowed and which are not.

3.3.3 Distributed Computing Environment Remote Procedure Call

DCE RPC has its own interface definition language (IDL). DCE IDL looks like C++, but instead of ‘class’, the definition ‘interface’ is used. IDL definitions are driven through a compiler to produce stubs and skeletons as with ONC RPC XDR definitions.

RPC is not object oriented and does not allow object references or copies to be transferred over method calls. RPC does allow most of the arguments available in C, including pointers. RPC clients can access and use pointers in servers. RPC takes care of addressing.

DCE RPC has been included in the Windows system. Distributed Common Object Model (DCOM) which is the standard middleware on Windows relies on the Windows RPC. DCOM extends RPC with object oriented features.

First thing I thought when I saw my first RPC IDL after a year of CORBA programming was that they look very much alike. I have not studied the historical perspective, but I guess RPC has been as much as a model for CORBA as RPC has been the base for DCOM.

3.4 Common Object Request Broker Architecture

CORBA is a specification built by Object Management Group. OMG is a joint venture of several software vendors, IT organisations and institutions. The specification describes an environment and a protocol for distributed computing.

CORBA is a middleware for objects that can exist anywhere on the network, and that could have been implemented with different programming languages. CORBA also specifies different kinds of support for co-operation with objects using other middlewares and specifies general features that support distributed architecture.

Most CORBA implementations use Internet Inter ORB Protocol [IIOP], which is based on General Inter ORB Protocol [GIOP]. IIOP is TCP based protocol, and it is required in CORBA specification, that each CORBA implementation must implement it.

Interests of software vendors have brought width into the CORBA specification. As an example Sun brought the Java RMI to IIOP mapping into the specification to promote their own product. Richness of CORBA is the width of the specification and that also sets it apart from the other middlewares. It should also be noted that CORBA is a specification, it is not a product. OMG members use CORBA to build their products – there are dozens of CORBA implementations on various stages.

This chapter is mostly based on OMG CORBA documents [OMG 1998 c] and on my experiences with CORBA implementations.

First sections, ‘CORBA 2.3’ [3.4.1], ‘Runtime View’ [3.4.2] and ‘Communication Synchronisation’ [3.4.3] deal with the specification.

Section ‘Practical CORBA’ [3.4.4] shows how specification has been brought into reality.

Following sections ‘Dynamic Invocation Interface, Dynamic Skeleton Interface and Interface Repository’ [3.4.5], ‘Layers in an Implementation’ [3.4.6] and ‘CORBA Services’ [3.4.7] deal with advanced topics in CORBA.

Last section ‘Evaluating Between CORBA Implementations and Other Middleware Implementations’ [3.4.8] presents my view as a software developer towards CORBA.

3.4.1 CORBA 2.3

CORBA specification 2 concentrates on interoperability between different CORBA implementations and interoperability of CORBA with other middlewares such as COM and RMI. The CORBA specification is mostly meant for CORBA vendors, not CORBA

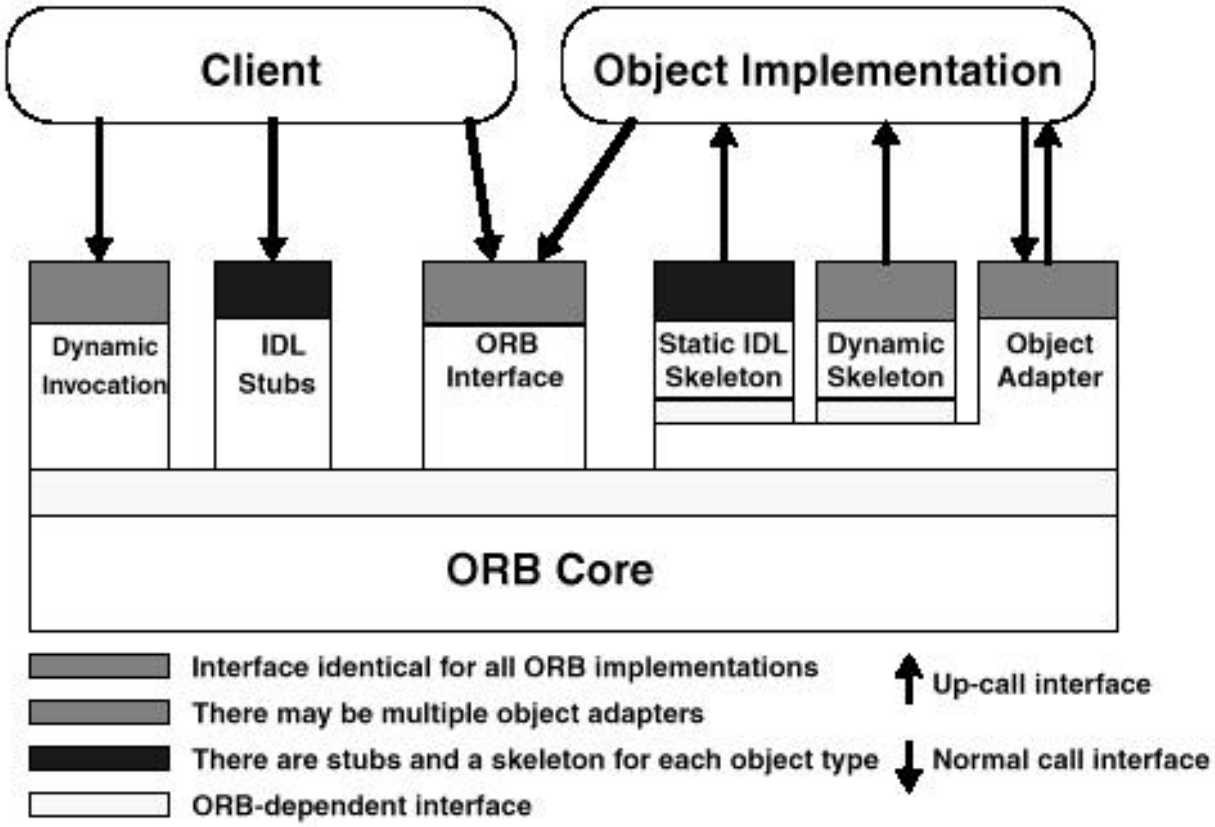
users. It is up to the CORBA vendor to tell the users how to use the specific CORBA implementation. Actual specification defines how Object Request Broker should be implemented.

Issues specifically handled in the specification are

- object model,
- overview of the basic architecture,
- the Interface Definition Language (IDL) syntax,
- the interface to Object Request Broker,
- value types such as CORBA null,
- the Dynamic Invocation Interface,
- the Dynamic Skeleton Interface,
- the Interface Repository,
- Interoperability,
- Interoperability specifically with other CORBA implementations,
- Interoperability specifically with Component Object Model (COM) and
- ORB building over Distributed Computing Environment (DCE) as an example of Environment Specific Inter-ORB Protocol (ESIOP).

3.4.2 CORBA Runtime view

The Object Request Broker (ORB) offers an interface to each participant. The ORB Interface makes available the very basic services like fetching the initial object references. An IDL Stub is a client side proxy object derived from an Interface Definition Language (IDL) file. A static IDL Skeleton Object Adapter is a server side proxy object derived from the IDL file.



3. The Structure of Object Request Interfaces

In the case of Dynamic Invocation and Dynamic Skeleton Object Adapter the developer takes the responsibility of translating a function call into an ORB message in the client side and translating the ORB message into a function call in the server side. Usually dynamic request handling is not used.

Object Request Broker, ORB Core in the figure, must be able to store object references to object implementations and to convey function calls to these objects from clients.

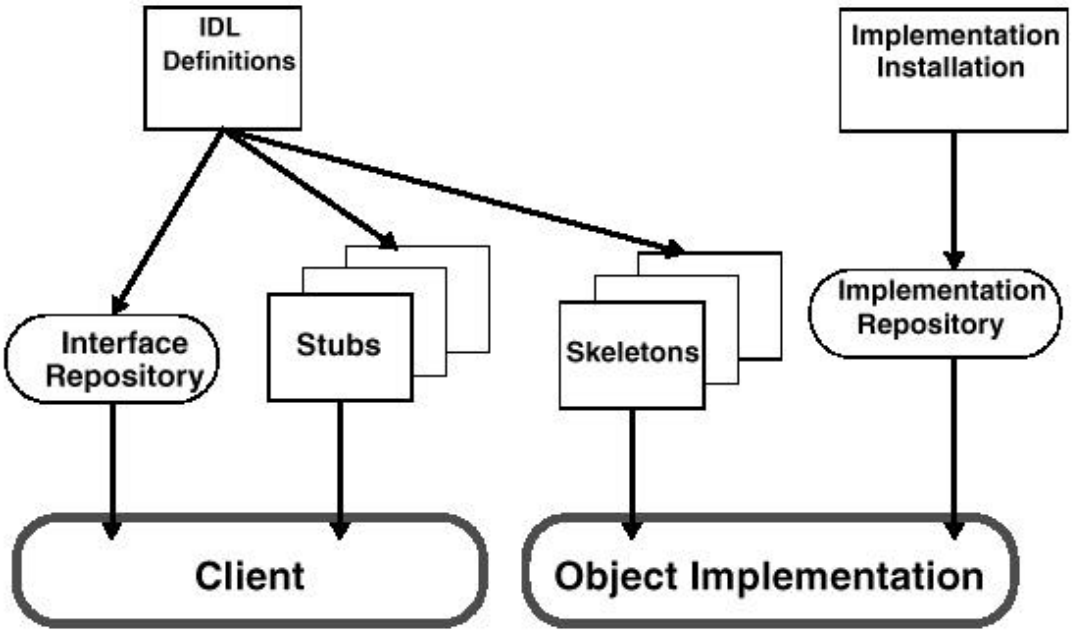
The following sequence shows what happens in a basic situation, from the point that the ORB is activated to the point where the client gets the response from its call.

- The object registers to an ORB using ORBs interface.
- The client gets an object reference to the object from the ORB interface.
- The Client makes a call against the IDL stub.
- The ORB is activated.
- The stub conveys the call to the static IDL skeleton using information from the object reference.
- The ORB activates the object and the object IDL skeleton.
- The IDL skeleton invokes the implementation of the actual object.
- The Implementation responds to the static IDL skeleton.
- The Static IDL skeleton conveys the answer to the IDL stub.
- The IDL stub returns the response to the client.

If the client makes subsequent calls, it will simply call the stub again. The ORB and the object may or may not be active between calls.

3.4.3 Communication Synchronization

The Interface Definition Language is used to lock communication from the client to the object. For each programming language there is a mapping from the IDL to that language. The mapping of the IDL to a programming language produces stubs and skeletons of the proxy design pattern.



4. Interface and Implementation Repositories

The Implementation Repository is the place where the Object Request Broker finds the Object Implementation. The Interface Repository holds raw IDL files which clients can use to query available Object declarations. In a way the Interface Repository shows what kind of classes are available, and the Interface Repository shows where objects which implement these classes are.

Figure 4 can be used to illustrate what has to be developed and as prepared, before actual processes can be started. This is a somewhat basic case, leaving out the use of the interface repository and the dynamic invocation and the dynamic skeleton.

- The Interface Definition has to be written with the Interface Definition Language.
- The Definition has to be compiled into a stub and a skeleton.
- An implementation that will be accessed by the skeleton has to be written to complete the server.
- A client will be written, which will access the stub, as it would be the actual object.

- An implementation will be installed into the repository.

After these procedures a client can access the Object Request Broker, which will find the object through the implementation repository. In some implementations the ORB and server will have to be started before the client.

3.4.4 Practical CORBA

Things in engineering can be presented on two levels: the abstract and the practical. With CORBA it really makes sense to present use on both levels, as CORBA has one specification and dozens of implementations. The CORBA specification ties the CORBA implementations together, but in my opinion the specification does not give a good picture of what a CORBA implementation can be like, as implementations I have seen have so much variation between each other.

The IONA Orbix architecture is quite recognisable with CORBA. It serves quite nicely as an example of a CORBA implementation.

Orbix component	**Function**
IT Daemon	Process. Conveys initial object references. Offers services for querying current status and such.
IDL compiler, different for each language.	Tool. Compiles IDL files into, for an example, C++ header and source files.
ORB Configuration	File. Contains address for IT daemon, and Implementation Repository directory path
Implementation repository	Directory. Contains file for each object implementation registered to ORB. Offers persistence, as IT Daemon would lose reference information when terminated and restarted.
Registration program	Tool. Registers object into Implementation Repository.

5. Orbix components

Table 5 shows the components Orbix has for the implementation of basic distributed functionality. The IT Daemon together with compiled IDL stubs and skeletons form the ORB. Other components just support the functionality. The following sequence shows what has to be done with these tools to produce a distributed object system and how that system works.

1. An IDL file is compiled into stub and skeleton.
2. A client is programmed to use the stub as it would use the target object, except that it has to initialise that object using the host and the port in which it resides.
3. The server is programmed to have an object that inherits from the skeleton. An abstract skeleton forces the object to implement necessary functions as in the proxy

pattern [Gamma et. al. 1994]. In a single threaded model a server is left in a loop to listen for remote requests.

4. Using the Registration program, the server is put into the implementation repository, directed by the information in the ORB configuration file.
5. The IT Daemon is started, it starts to listen to the port directed by the ORB configuration file.
6. The client is started, it contacts the IT Daemon directed by the ORB configuration file.
7. The IT Daemon launches the server based on the implementation repository information.
8. The server is started and it contacts the IT Daemon based on the information in the ORB configuration file.
9. The IT Daemon assigns a port for the server and the server enters a loop, in which it listens to that port.
10. The IT Daemon gives the server a port to the client.
11. The client connects to the server, and gets an object reference as a pointer to a stub.
12. The client invokes the stub that takes the request over to the server.

Green steps [1-3] go through development phase, blue steps [4-5] go through administration phase, red steps [6-10] go through initialisation phase and black steps [11-12] are the actual invocation.

3.4.5 The Dynamic Invocation Interface, The Dynamic Skeleton Interface and The Interface Repository

The ORB offers an interface that allows dynamic sending and receiving of requests. These features are called the Dynamic Invocation Interface (DII) and the Dynamic Skeleton Interface (DSI). With DII the parameters are put into a list and the ORB interface is called with the function name and the object reference to the implementation. The DSI on the other hand receives the parameter list and the function name asked. Firstly these interfaces define a standard way to make the static stubs and skeletons. Secondly these interfaces allow building of dynamic responses. The Interface Repository (IFR) allows storing and querying of IDL definitions. These three tools work nicely together in a dynamic environment.

The communication resulting from an invocation does not vary regardless of whether dynamic or static invocation is used. The communication towards the object is the same regardless whether the object handles it using the DSI or static skeletons. In the case of DII, the developer has simply taken the responsibility of constructing the request and in the case of DSI the responsibility of parsing the request. The ORB still comes to the aid offering the interfaces to ease the job; the developer does not have to start parsing protocols.

The fact that the ORB has these interfaces for building and parsing requests makes stub and skeleton building easier. A proxy generator can make code that works directly with IIOP, but mostly generators rely to the existing ORB interface. So as a standard the proxies generated from IDL definitions use DII and DSI. This fact is not publicly documented, but apparent on inspection of the generated code [Javasoft 1999 a], [IONA 1998 b].

The Interface Repository wraps the interface definitions into the object world. An interface can be accessed from the repository as if it was an object querying methods, variables and types. Interfaces are usually stored into the repository in their raw form, as IDL files, but they are accessed from repository through special interfaces. The IR makes it easy to handle interfaces, specially generating runtime IDL stubs for CORBA objects. The component Object Model (COM) has a similar feature, the Dispatch Interface.

3.4.6 Layers in an Implementation

In the eyes of a developer it does not matter what is between the client and the target object – as long as it works. The DII and DSI may have uses for the developer, of course, but often they are not used either. This layering I present here presumes that the CORBA implementation itself uses DII and DSI.

Invoking Client Code	Object Request Broker (ORB)
Stub	
Dynamic Invocation Interface (DII)	
Internet Inter-ORB Protocol (IIOP)	
TCP/IP	
IIOP	
Dynamic Skeleton Interface (DSI)	
Skeleton	
Target Object Code	

6. Layers involved in a dynamic CORBA remote invocation

The IIOP is an Internet Inter-ORB Protocol. In the level of abstraction, it is roughly the same as Hypertext Transfer Protocol. The IIOP is an implementation of General Inter-ORB Protocol (GIOP) for TCP/IP communication. IIOP may be replaced by the Environmental Specific Inter-ORB Protocol (ESIOP) such as the Open Software Foundation (OSF) Distributed Computing Environment (DCE) [OSF 1996] ESIOP [OMG 1999 b]. The older CORBA implementations may use a protocol designed by the vendor.

I think the figure of layers shows quite nicely how remote invocations can be abstracted from basic communication protocols to object level. Layers abstract communication to higher levels, leave details from the developer step by step. The IIOP solves the problem of data representation. The DII and DIS ease the use of The IIOP into simple interface invocations and the stub and the skeleton customise the protocol for the developer use. The ORB has to be initialised, but on other hand it automates things and can bring additional services to the developer.

3.4.7 CORBA Services

The Naming Service is the most common of the CORBA services. The Naming Service specifies a CORBA object, which can hold references to other objects and which can structure the references in an ordered way. Without naming service each CORBA object would need contact information to each object it need during its lifetime. With naming service the object needs only contact information to the naming service and it can get the rest of the information from the service. There are fifteen services in the CORBA specification. Most of the services are more complex than the Naming Service and they are of varying usefulness.

Implementing CORBA does not require implementing any of the services. Services are not a part of the CORBA infrastructure, but standard building blocks that can be used in building CORBA applications. The idea of having these blocks standardised is that one can use the CORBA of one vendor and some services from another. Most of the CORBA vendors have implemented the Naming Service; none has implemented all of the services. Therefore a developer might want to use a CORBA of personal preference, but use in addition services from another vendor. Implemented services are listed at the end of this subsection.

The CORBA services are Naming, Event, Persistent Object, Life Cycle, Concurrency Control, Externalization, Relationship, Transaction, Query, Licensing, Property, Time, Trading Object and Object Collection [OMG 1999 b].

The Event service offers a public space on which to inform about events. Instead of making requests, states are published. There can be several publishers for a state, several consumers for a state and publishers and consumers of a state do not have to be aware of each other.

The Persistent Object Service standardises object storing and retrieval. It offers a standard interface for this purpose.

The Life Cycle Service defines services and conventions for creating, copying, deleting and moving objects in remote locations.

The Concurrency Control service and The Transaction service together makes it possible to create more reliable systems. The Transaction service offers atomic operations; The Concurrency Control service makes sure that operations do not overlap

when handling critical data. The Concurrency Control service and The Transaction service are often used in combination.

The Externalization service enables streaming of object data.

The Relationship service offers relational services between objects.

The Query service allows gathering of objects as collections. Collection gathering is based on a query, which can be based on SQL or a similar method.

The Licensing service standardises the use of licenses. Each object or object group may have a different license.

The Property service allows adding of properties to objects. The Object data and methods are usually defined compile time, but the Property service allows adding of properties runtime.

The Time service offers accurate time.

The Security service combines different security perspectives like encryption and authentication.

The Trading Object service offers another method for the object access beside The Naming service. The Trading Object service publishes services contained within; the client can decide which object to choose based on ‘advertised’ services.

The Object Collection service offers different kinds of containers in which to store remote objects. The Service offers also iterators and ordering services for these containers.

It is likely that new CORBA services are published, as OMG participants see need for that. OMG also has a policy for removing parts of the specification if they are not used [Reaz Hogue 1999]. Although it can be seen that all of these services might have a use in some situation, that situation could be so rare that there would be no point in keeping a common specification

The following figure of services by Ben Eng [Eng 1999] says a lot about the situation of services today.

Vendor	Nm	Lf	Ev	Tr	Cc	Ex	Po	Tx	Qr	Tm	Pr	Sc	Li	Av
Expersoft	Y→	#→	Y→	→	→	→	#→	→	→	→	→	→	→	→
Sun	Y→	Y→	Y→	→	→	→	#→	→	→	→	Y→	→	→	→
IONA/ILOG	Y→	?→	Y→	Y→	?→	→	?→	Y→	→	→	→	Y→	→	→
Visibroker	Y→	?→	Y→	→	?→	→	?→	Y→	→	→	→	Y→	→	→
BEA	→	Y→	+	→	→	→	→	Y→	→	→	→	Y→	→	→
PeerLogic	Y→	Y→	Y→	Y→	?→	→	?→	Y→	→	→	→	Y→	→	→
HP	Y→	Y→	Y→	Y→	→	→	→	Y→	→	→	→	?→	→	→
IBM/TANDEM	Y→	→	→	→	→	Y→	→	→	→	→	→	→	→	→

Chorus	#→	→	→	→	→	→	→	→	→	→	→	→	→	→
OOT	Y→	+→	→	→	→	Y→	→	→	→	→	→	#→	→	→
DNS	Y→	Y→	→	→	→	→	→	+	+	→	→	→	→	→
Prism	Y→	Y→	Y→	Y→	→	→	→	→	→	Y→	Y→	→	→	→
Electra	Y→	Y→	Y→	→	→	→	→	→	→	→	→	→	→	→
U Colorado	→	→	→	→	→	→	→	→	→	→	→	→	→	→
Xerox	#→	#→	→	→	→	→	→	→	→	→	→	→	→	→
BBN	Y→	Y→	→	→	→	→	Y→	→	→	→	→	→	→	→
SNI	Y→	Y→	Y→	→	→	→	→	→	→	→	Y→	→	→	→
TRW	Y→	→	→	→	→	→	→	→	→	→	→	→	→	→
ParcPlace	Y→	Y,#	Y→	→	Y→	#→	#→	Y→	→	→	#→	#→	→	→
TIBCO	Y→	→	Y→	→	→	→	→	→	→	→	→	→	→	→
Suite	Y→	Y→	Y→	Y→	→	Y→	Y→	→	→	Y→	Y→	Y→	→	→
B&W	→	→	→	→	→	→	→	→	→	→	→	→	Y→	→
Fujitsu	Y→	Y→	Y→	→	Y→	Y→	Y→	Y→	+	+	+	+	+	→
Nortel	Y→	Y→	Y→	→	→	→	→	→	→	→	→	→	→	→
Camros	Y→	→	Y→	→	→	→	→	→	→	→	→	→	→	→
TAO	Y→	Y→	Y→	Y→	Y→	→	→	→	→	Y→	Y→	→	→	Y
JacORB	Y→	→	Y→	Y→	→	→	→	→	→	→	→	→	→	→
Vendor	***Nm***	***Lf***	***Ev***	***Tr***	***Cc***	***Ex***	***Po***	***Tx***	***Qr***	***Tm***	***Pr***	***Sc***	***Li***	***Av***

7. CORBA Services Feature Matrix [Eng 1999]

Y	This feature is supported but not necessarily conformant to CORBA standards
#	This feature is supported in a non-standard implementation
+	Real soon now
-	This feature is **not** supported
?	This information is not known

8. Matrix Key [Eng 1999]

Nm Naming	Ev Event	Cc Concurrency	Po Persistent Objects
Lf Lifecycle	Tr Trading	Ex Externalization	Tx Transactions
Qr Query	Cl Collections	Tm Time	Pr Properties
Sc Security	Li Licensing	Av Audio/Video Streaming	Cm Configuration Management

9. Matrix Explanations [Eng 1999]

3.4.8 Evaluating Between CORBA Implementations and Other Middleware Implementations

For some time now CORBA has been compared with other middlewares – mostly with COM and RMI. I suppose that marketing departments representing involved vendors try to use this discussion to their advantage. The fact is that each widely used middleware has several software vendors, therefore there are several implementations of each middleware. A debate could rightly be taken therefore if one middleware has been specified better than the others have but it can hardly be said that CORBA implementations in general are better than COM implementations.

When choosing a CORBA implementation there are lots of choices. Some vendors have implemented several services and other features in their product. Others have a high performance ORB, which can relay requests magnitudes faster than competitors ORBs. Although CORBA is platform and language independent, implementations are often only for few different platforms or languages. Reliability and quality are hard to measure. There are also several implementations under the Gnu licence making them more desirable in some cases.

Comparing CORBA implementations with other middlewares can be done quite as straightforwardly as CORBA implementations between each other. COM is well-supported only on Windows platforms. COM in itself is free, but developing COM applications needs commercial tools. COM applications can be developed with vast existing libraries, of which most are commercial. COM is supported on Visual Basic, C++ and Java. If a middleware were to be chosen, in my eyes COM could be thought as one CORBA implementation among the others with its quirks like poor interoperability with other middlewares.

3.5 Component Object Model

The COM [Microsoft 1995] is a Windows operating system native object model. Today COM is not limited to Windows, but other implementations are insignificant compared to the Windows application mass. Libraries which enable Distributed COM (DCOM) have been shipped with each Windows operating system after Windows 95. The Windows 95 also has a free extension for enabling DCOM.

The Emphasis of COM is on single-user, multitasking visual desktop. COM objects are seen as subcomponents of an application. COM objects are often embedded in documents. COM objects are not limited to subcomponent status, but the COM model clearly supports it.

DCOM works on top of Open Group Distributed Computing Environment Remote Procedure Call (DCE RPC). The COM interfaces are defined with similar languages as

the RPC Interface Definition Language, which is either Microsoft IDL (MIDL) or Object Definition Language (ODL), even if they are not distributed over network.

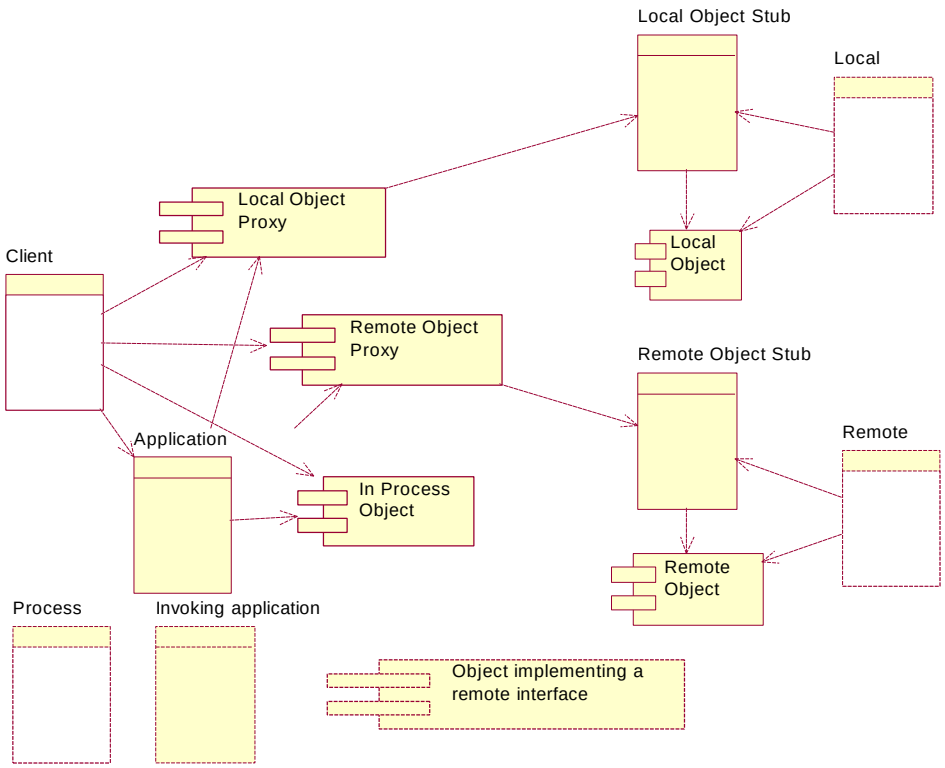
The COM technology sprawls in different areas and contains such separate and overlapping things as COM, DCOM, OLE, Automation, COM+, ActiveX and Microsoft Transaction Server (MTS).

3.5.1 Component Object Model

COM is a standard for binary files. The Microsoft Interface Definition Files (MIDL) have a standard way of compiling them into COM components, but they are not a part of the COM standard. As COM is purely a standard for component interoperability, it is language and platform neutral.

The COM related technologies are built on top of COM. COM is the infrastructure and related technologies are the tools to build COM applications and extensions to standardise application building.

COM uses Distributed Computing Environment Remote Procedure Call (DCE RPC) as a communication protocol. COM uses the proxy pattern [Gamma et. al. 1994]. COM is the component that takes care of finding the skeleton for the client and COM has the security needed in these operations. COM offers an interface to the client where it exposes its functionality. Again there is a mapping from MIDL to these COM interfaces, although MIDL is not a part of COM. COM also has a built in persistent storage functionality.



10. COM Communication

The COM objects always need a proxy stub and a proxy skeleton. COM also needs a registry in which object libraries or executables are found. The COM implementation is platform dependent and so is registry handling.

Class Ids and Interface Ids identify COM objects; both are generated with the same system as RPC Globally Unique Identifiers (GUID). COM does not allow exception handling and has a strange way of taking care of inheritance – which is handled in detail at COM/CORBA interoperability section where CORBA can be used for comparison.

3.5.2 Distributed COM

When COM was first published it simply replaced the Windows messaging system for embedded objects, drag and drop functionality and such. The DCOM was simply an extension to COM, which allowed discovery of objects over network space. COM allowed object registry to point into a library or an executable; DCOM allowed it also to point into another computer.

The Service Control Manager (SCM) is a part of COM which takes care of object discovery. When a COM object is requested, the SCM is launched. When the SCM finds a network address from the registry it launches a SCM in that address and repeats the request it got from a client. The remote SCM looks up the object and invokes it as from the local computer.

The client does not have to know if the object is at a local address or if it is remote, it does not matter for the server either. Therefore COM servers can be relocated into a remote computer without any changes. DCOM was more like a patch to COM than a separate technology, which can be seen from the literature too, which usually speaks only about COM.

3.5.3 Automation

The Automation is a dynamic invocation wrapper on top of COM. The automation wraps around the Dispatch COM interface which allows any request to be processed. The automation removes the need of proxy stub generation during client building.

Microsoft tools support automation better than just plain COM. It means that a COM server and a COM client are more easily created as automation than plain COM. The client building is definitely easier as the client needs only a name to the COM registry and some kind of interface definition.

Usually interface definition is acquired from a type library (TLB). Type libraries are usually generated from IDL files. The advantage of using generated information instead of original information is that generated information has been checked. The process can be compared into Java byte code generation.

The automation interfaces are generated with Object Definition Language (ODL). An automation object, which supports only automation, is called a dispatch interface and an automation object, which also supports COM, is called a dual interface.

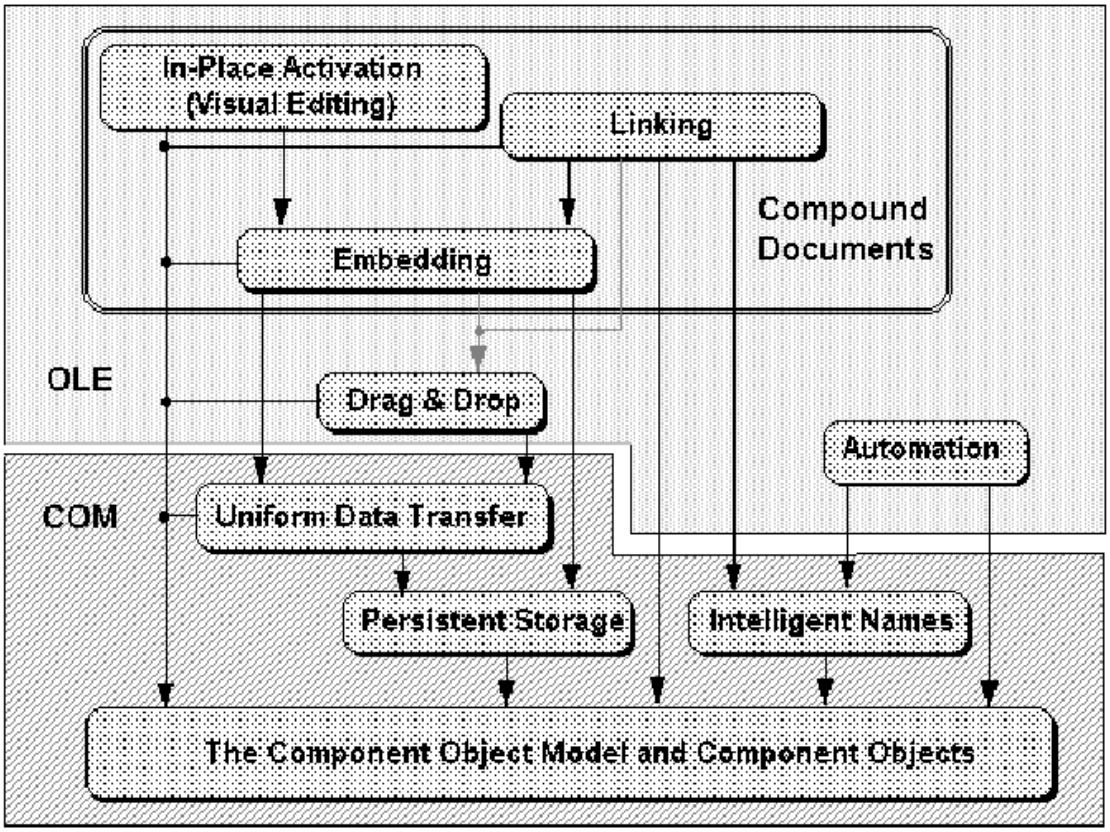
3.5.4 ActiveX

ActiveX is a graphical user interface component technology. ActiveX defines how an object should define its interfaces, so it could be easily used in graphical builder tools and web pages. Specifically, ActiveX extends OLE user interface components to Internet use.

ActiveX is a sister technology to Java Beans. Both technologies present a method for querying interfaces. Unlike Java Beans, ActiveX components can be written with any language supported by Microsoft. And unlike Java Beans, ActiveX is usually seen only in Windows. ActiveX components are used as shared libraries. An example of an ActiveX component is a button: when it is pressed in a panel, an event is sent to the listener. As Java Beans, ActiveX components have little to do with distribution.

3.5.5 Object Linking and Embedding

Object Linking and Embedding (OLE) are the drag and drop mechanisms and other gadgets needed in a desktop environment. Automation was originally called OLE Automation and ActiveX was first called OLE Controls. It is a bit hazy what belongs to OLE and what does not, but generally everything in OLE is built on top of COM.



11. COM Architecture [Microsoft 1995]

Compound documents are documents which have other documents embedded in them. Embedded documents are linked to the source in which they were originally created and can be activated and edited through container document. Drag and drop is a tool to make this document embedding as easy as possible.

The first mission of COM was to serve this structure, which explains its original remote capability haziness. Clearly COM will still serve OLE besides being more common in distributed systems. The Microsoft strategy in the matter seems to be building tools to communicate with COM, so that it can be used from different perspectives without too much developer complexity.

3.5.6 COM+

COM+ [Kirtland 1997] is another wrapper on top of COM. To make it short, COM+ apes CORBA and Remote Method Invocation (RMI). COM+ allows exceptions and gives Global Unique Identifiers (GUID) on the advantage of namespaces. Inheritance model has been copied from Java allowing multiple interface inheritance and single implementation inheritance. Events have been adopted from ActiveX.

As a very interesting point COM+ does not define a unique interface language like RPC, RMI or CORBA. It has simply been stated that for each language a tool that fits in to the environment will generate the COM+ code. In C++ case this does mean a new

interface language which looks like C++ headers. With Java all ordinary Java classes are COM+ classes.

Clients can use Automation and COM+ servers at the same time – although it seems that same syntax can not be used for both. As with Automation, servers are described with compiled information. As TLB file syntax is not diverse enough to support COM+ requirements, the COM+ tools will create a new metadata type file.

3.5.7 Microsoft Transaction Server and Other Services

Microsoft Transaction Server (MTS) is a COM component, which ensures concurrency and atomic operations of other COM components. MTS works between COM components and COM components and databases. As with other Microsoft products, also MTS is supported by different Microsoft environments. MTS requires components to use a transaction protocol, either Microsoft OLE transaction protocol or The X/Open DTP XA standard. It is the job of MTS is to automatically begin and commit transactions.

With MTS Microsoft COM has security, persistency, transaction, graphical component and discovery support. The services listed above are guaranteed but other services may be available through other service vendors.

Microsoft has implementation of COM on Windows and Macintosh and there are implementations on other platforms by other vendors.

COM objects can be implemented with Visual Basic, Visual Java and Visual C++. Other vendors and not just Microsoft also build support for COM, at least as long as the related programming tool is for Windows.

Overall COM has been quite widely *implemented* compared to other middleware *specifications* and has quite a good support.

3.6 Remote Method Invocation and Related Java Tools

Remote Method Invocation [Javasoft 1998] is a Java specific middleware specification. Specialising into one environment gives it strength that lacks from other middlewares. RMI is easy to learn and easy to maintain. For some reason RMI has not really been developed since the basic infrastructure was published leaving it somewhat unprepared for large-scale systems. There are few other Java technologies related in one way or another to RMI or distribution generally. Related technologies are Jini, Enterprise Java Beans and Servlets and they get a piece of their own in this representation.

As RMI has been kept simple, major features are easy to list:

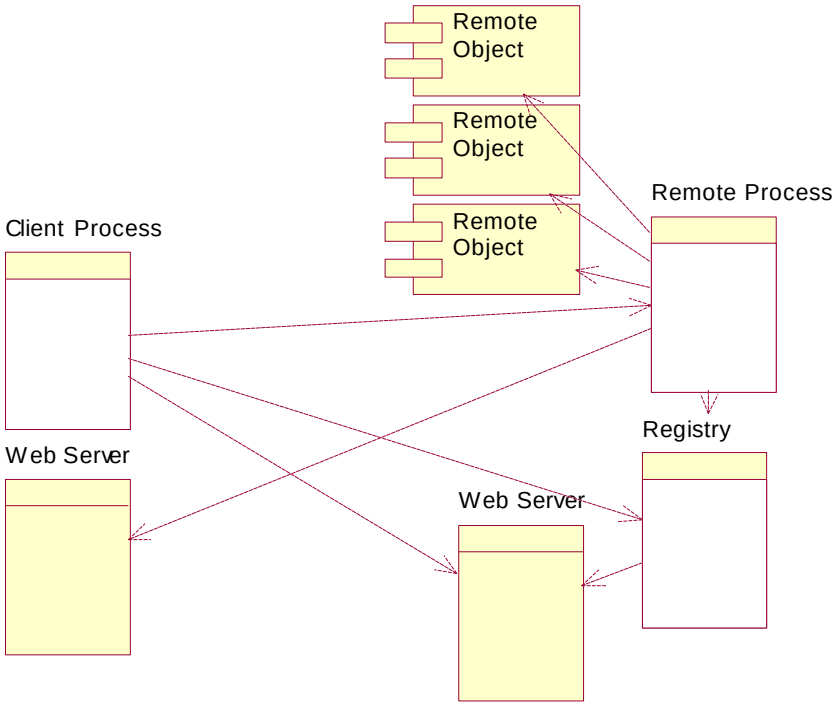
- ability to copy code over remote invocations,
- can be used through Common Gateway Interface [Coar et. al.] scripts,
- can encrypt network traffic,

- includes a naming service and
- remote reference counting and garbage collection.

The RMI specification has been tightly tied to implementation so it can be read almost like a user guide. RMI specification has been fully implemented by Sun in Java Development Kit 1.1 and 1.2.

3.6.1 RMI Architecture

The RMI system does not have to have anything to do with web servers but can work in conjunction with them. In a basic setting the client and the server interact as in the proxy pattern [Gamma et. al. 1994]. Initial object references are found through the RMI registry, which is a naming service. The client communicates with Java Virtual Machine, which can have both remote objects and local objects available.



12. RMI Object Model

Web servers are needed if a direct connection can not be established between components. The RMI specifies how the HTTP protocol and the CGI programs can be used for connection establishment and communication. The communication will therefore use WWW services, which usually are granted access to.

The RMI registry is a naming service. It forms a directory in which object/name pairs can be stored. Registry reserves a port in the host machine and is contacted through a naming interface ‘java.rmi.Naming’.

Client and server objects communicate using proxies. Initial references between proxies are traded through RMI. After the client and the server have established

communication, both ends can easily trade new references between themselves. Proxies are generated from ordinary Java interfaces.

3.6.2 Remote Interfaces

The great thing about RMI is that the developer can do all the work with Java language. Microsoft COM gets near in that the remote interface can be generated with tools.

Ordinary Java interfaces inherited from interface java.rmi.Remote can be compiled into RMI proxies. All methods in a RMI interface must declare an exception called RemoteException that is used by the system.

Remote objects are passed as references in method invocations. Ordinary objects are passed as copies. Objects that are to be copied over RMI must implement an interface named as Serializable. Java can turn Serializable objects into data.

Stubs generated from Remote interfaces are also Serializable. When a client gets an ordinary object over remote call, it gets a copy of the object. When a client gets a remote object over a remote call, it gets a stub of the object. The exceptions are passed as objects like any other objects. The remote interface can define exceptions like any Java interface; also native exceptions can pass through remote calls.

3.6.3 Servlets

Servlets are classes that implement a special Servlet interface. The Servlet interface and extending HTTP interface define methods in which HTTP requests can be submitted to a Java object. The result could be compared to a CGI script invoking a Java object, except that a lot of overhead is taken off.

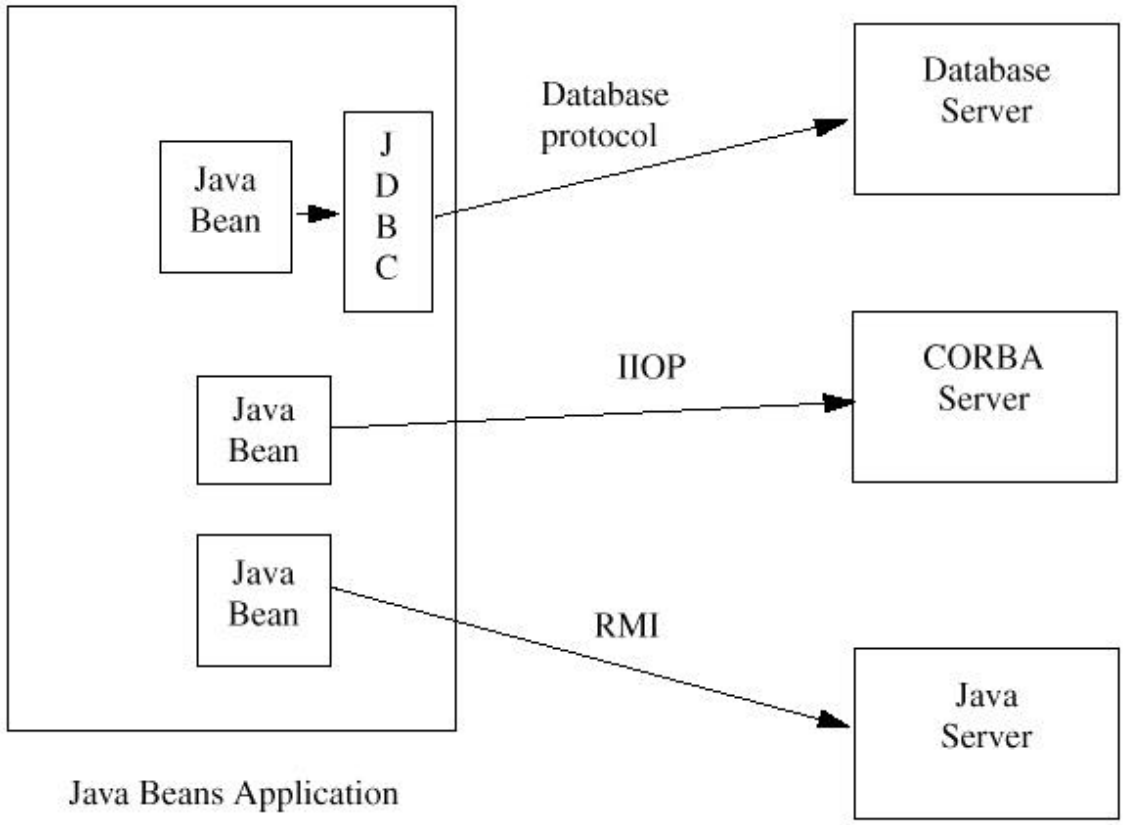
Servlets work as ordinary Java programs. The Servlets have an interface for easy cookie handling and also for alternative session handling. Changing the Uniform Resource Locators does alternative session handling. Servlets have nothing to do with the RMI, except that they could very well be used as an extension in a system that has HTTP interfaces besides RMI.

Microsoft has a similar tool called Active Server Pages (ASP). ASP applications are written with Visual Basic so they have strong connection to the COM model.

Both Servlets and Asps need collaboration of the web server. In my understanding most modern web servers support them though.

3.6.4 Enterprise Java Beans

The Java Beans define how components should be built with Java. When a component implements a Bean interface, the rapid application development tools can be used to attach the component into software. A Bean component can also have complete functionality to be used in a container like a compound document.



13. Java Bean Possibilities [Javasoft 1997 b]

Figure 13 from Java Beans Specification [Javasoft 1997 b] shows how Java Beans can use remote components through different middlewares. Java Beans are actually ordinary Java classes and can implement all necessary functionality in them as appropriate.

What makes Beans so special is that Sun has co-operated with different software vendors to enable interoperability of Beans in different component systems. The Beans specification allows interoperability with different COM models. With Bean interoperability with other component models, the Beans can effectively be used as a bridge from different applications to RMI as well as to other Java based tools.

3.6.5 Jini

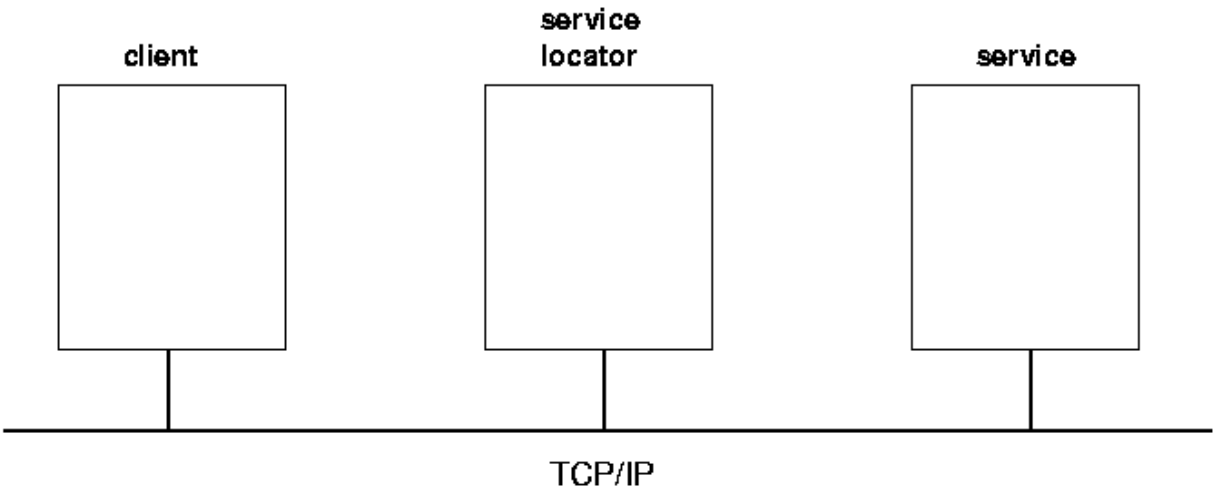
Jini is a middleware working on top of Remote Method Invocation [Javasoft 1998]. The idea of Jini is to provide a network wide ‘plug and play’ environment. Ordinarily middlewares contact each other through a naming service or with a predefined configuration information. A Jini system has a discovery service, which does not need any predefined information.

When distributed systems are built, objects will have to be configured to contact each other. In my experience this configuration information is difficult to keep updated and prone to errors. Jini like technology is therefore quite welcome in my view.

The Jini model is quite complex to the developer, but it seems that Sun is currently developing an easier approach. A problem of the Jini is that it is not transparent to the developer, the cost of hiding the contact information has been an added complexity of the actual middleware.

3.6.6 Jini Component Model

Newmarches component model looks just like any other middleware component model. The components are a client, a service, a service locator and the network. The only implementation of the network is TCP/IP.



14. Jini Component Model [Jan Newmarch 1999]

Jini seemingly works like any other middleware. The service finds a locator and gives an object to the service. The client finds the locator and gets the object.

In the case of an RMI service the object transferred through the locator to the client is a stub. The client uses the stub to access services through the network.

The service object given to the locator by service is not limited to RMI stubs. The object can be a full implementation of the service needed. The object can use some other distribution mechanism besides RMI. In essence, the object contents are highly abstracted. In my view it is a quite significant thing – a next step of abstraction behind distributed objects, behind middlewares.

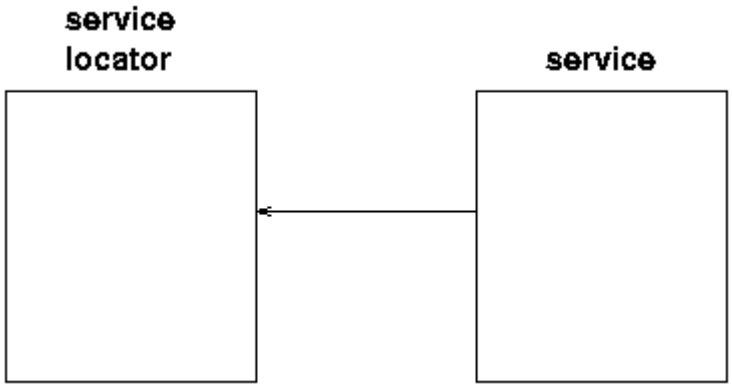
3.6.7 Jini Service Discovery

As the component model shows, the Jini functionality does not differ from other middlewares. The difference between Jini and other middlewares is simply that the Jini

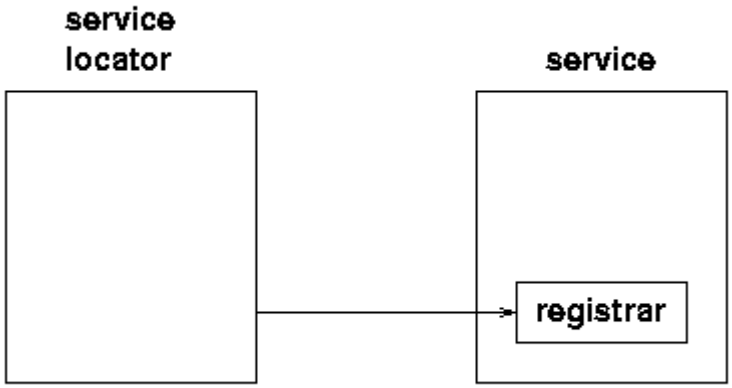
developer and the application end-user are separated from the network by a layer. This layer takes care of contacting the locator, a kind of a naming service, in both the client and the server side.

Ordinary middlewares abstract the communication between service and client to ordinary method calls. Jini also does the abstraction of communication, but it also abstracts the fact that there is a network where the communication is being done.

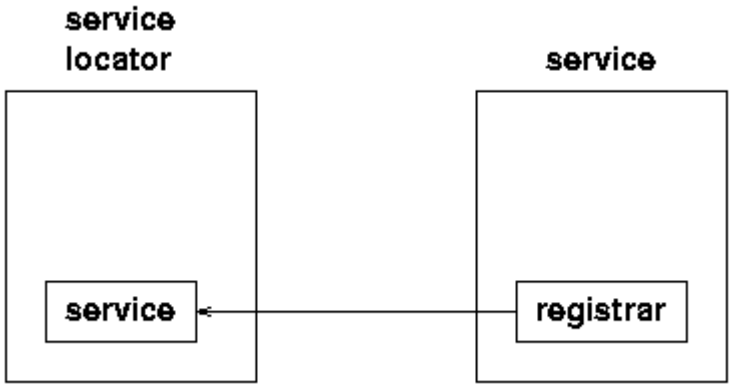
The transparency of network is possible with broadcasting. When a service enters the network, it broadcasts a message. Based on the broadcast message, the service locator knows where to fetch the service. The service locator gives the service an interface that the service uses to push its objects to the locator. The client contacts the service locator in the same manner and pulls required objects from the locator through the given interface. Following figures by Newmarch [Jan Newmarch 1999] illustrate the process.



Service finds the locator using an URL or by broadcasting.



Locator gives the service an access interface.



Service uses the access interface to add its information to the locator.

15. Service registration process.

Broadcasting is done with an UDP packet sent to broadcast host 224.0.1.84 and port 4160. Broadcast spreads from host to host, until it expires. Expire time can be configured, but usually it is set to expire on local network. Besides broadcasting Jini can discover a locator through knowing the exact host – usually used when communicating to another network. Web servers can be Jini enabled. Discovering through a web server is done with HTTP request.

3.6.8 Java Native Interface

Java Native Interface (JNI) [JavaSoft 1997a] is a bridge from Java to platform dependent binary libraries. In Windows these libraries are Dynamic Link Libraries (DLL) and usually different platforms have similar binaries. Sometimes there are different binaries for dynamic and static linking.

JNI implementation is naturally platform dependent. The goal of The JNI project is that it could be used on every platform in which Java is used. JNI can be used to access performance critical code, legacy libraries and platform dependent features.

JNI is naturally mainly C++ oriented and C++ implementations are easily available. JNI automates bridging of Java and C++ in very much the same way as middlewares do. A developer will have to write a skeleton in the native code and the client will have to be aware that it is using native code through invocations are just like against Java code.

4 Bridging Technologies

Bridges implement interoperability between middlewares. One middleware can use another one through a bridge. Typical use would be integrating old components in the new system or using pre-built components, which have been built with different middleware. As there are several different middlewares, bridges generally become useful when components built with different middlewares are needed in the same system.

There are bridges mainly for communication between CORBA and other systems. There are also bridges for different protocols, but they are not the issues here. Most of the commercial bridges are for COM-CORBA. While bridges unite different middlewares, they can also be seen as uniting different operating systems – COM on Windows and CORBA on Unix.

Bridges come in varieties. Bridges can be formed compile time or dynamically. Bridges can allow bi-directional communication, or be just for one way. A bridge can be integrated to a client and a server or be an independent component. A bridge can replace normal component communication that disables communication with the actual middleware or duplicate it, which allows the use of a component simultaneously with two different middlewares.

This chapter deals with general bridging issues first [4.1] – [4.6] and then with two specific cases: COM/CORBA Interoperability [4.7] and RMI over IIOP [4.8].

4.1 Similarities in Middlewares

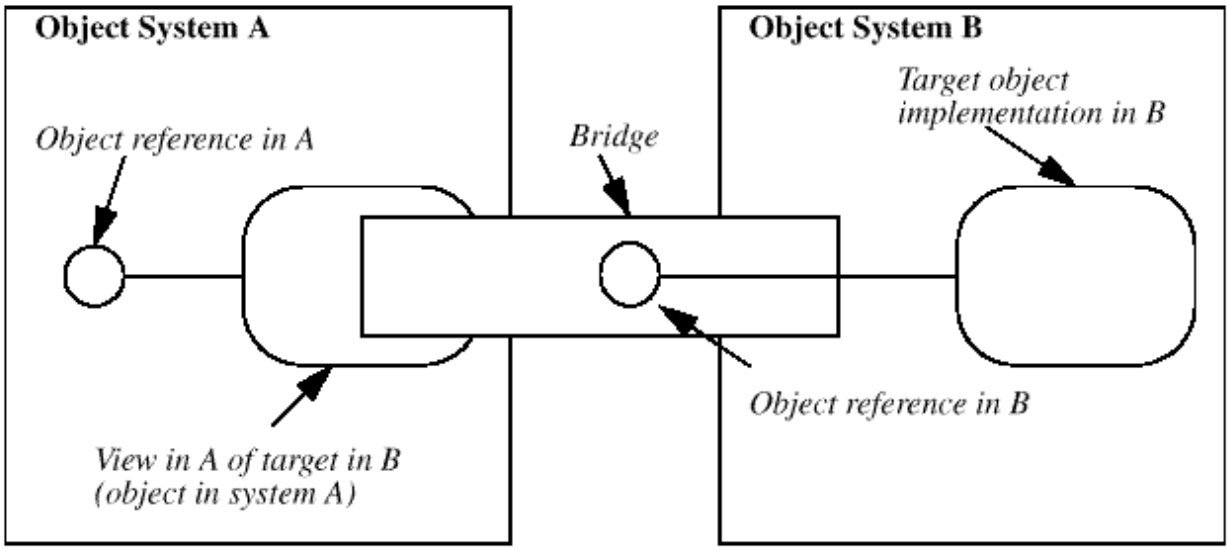
Bridging technologies have emerged quite quickly and from different vendors. Bridges are pretty easy to implement and the reason is that middlewares have a lot in common.

ONC RPC [Internet RFC 1831], DCE RPC [OSF 1997], COM [Microsoft 1995], CORBA [OMG 1999 b] and RMI [Javasoft 1998] all use an interface to define how a component can be used. The interfaces are language independent. All of the middlewares are protocol independent. Most of the middlewares are object oriented. In addition, all interface definition languages used in the middlewares have pretty much the same syntax.

Problems arise in object binding and in any advanced features middlewares might have. As there are two different middlewares used, there is a need for two object binding procedures. Advanced features like callback, exception handling and various object-oriented features may not be supported by the other middleware. When these difficulties have been overcame, there might be surprising complexity in the system and the client might well look like some kind of a hack instead of a piece of object oriented software.

4.2 Interoperability Model

Figure 16 shows basic interoperability model from OMG Interworking Architecture [OMG 1998 e]. The figure shows an object in system B which has been bridged to system A. If an application in system A would want to use the target in system B, it would seem to that application that it is using an object native to the system A. An object in A would have a native reference and a native view (as in figure).



16. Interworking Model

A bridge pretends to be an object implementing the functionality of the object in B in system A. In reality it takes the requests it gets to object system B and invokes the implementation in B, to which it has reserved reference earlier.

The fact that a bridge must reside in two object systems at the same time is somewhat confusing. Figure 16 is also a little misleading if the bridge is integrated into the client and does its job compile time.

4.3 Transparent Interoperability

Transparency in interoperability means that a component interface can be used as if it was implemented with the same middleware as the invoking code. This includes getting a reference to the component, making calls towards the component and handling return values from the component. It should be possible to view and treat the interoperable component as if it was built with the same technology as the client. Transparent interoperability would allow a developer to work with the familiar component model regardless of the component technology used in the target component.

Middlewares do not support identical features. Transparency means that although a developer is using a component that natively throws exceptions, her system wills not if it does not support them. Then information from the exceptions must be conveyed by some other means.

Compromises in data control may cause unusual solutions for the client. If a non-object oriented client will have to reflect an object oriented server in its calls, the result may be very different from the usual arrangement, although it could be syntactically correct. Syntactical transparency may therefore cause unconventional solutions. And although transparency is achieved, the gain may not be worth it.

4.4 Two-way Interoperability

Two-way interoperability means that the technology, which enables interoperability, works both ways with two different component technologies. Obviously it really does not matter if we have to use two one way technologies or one two-way technology, unless calls made in-between are somehow related.

Two-way interoperability becomes important when the target component expects to get a reference to the calling component, and make calls back to the invoking component. Call-back references could be required for instance if the component has a lengthy task and it must inform the invoking component when it is ready or if components are equal parts of the system needing each other.

In two-way interoperability solutions the interoperability solution must be aware of all related components, so that it may pass calls to any direction. In theory, if there were

such a component technology, which would not allow a client to be in the same package with the server, two-way interoperability would not be possible.

4.5 Static Bridge

Static bridges are built before any processes are started. Static bridges are usually built before the client and then the client will be built to use the bridge.

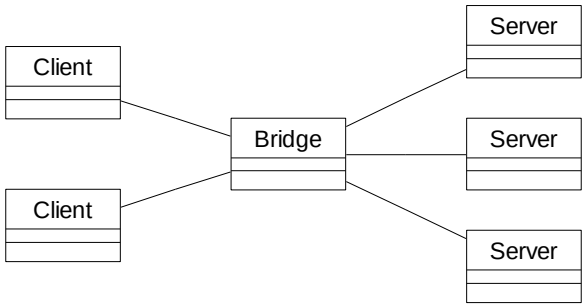
The simplest bridge is the one on the server side, produced during interface compilation. Instead of compiling for instance a RPC skeleton, bridge compiles a CORBA skeleton from the RPC interface definition. The object will be unable to serve RPC clients, unless compiled stub can serve both RPC and CORBA clients. If all servers can be compiled this way, then the other middleware can be completely discarded. If a server can not be recompiled, this option can not be used. Great advantage is high performance – a bridged server can have response times identical to that of an ordinary server. Another advantage is ease of use after rebuild – no additional components are needed. The idea of bridging is that the client can be developed with the middleware of choice, therefore there would be no point in doing the bridging on the client side.

Another kind of a bridge is a component that declares the same interface as the target component with another middleware interface language. The bridge does not implement any service logic; instead it is also a client to the target component and forwards the calls to the target. In straightforward cases this kind of bridge is easy to build manually. Problems are doubled network traffic and maintenance – each time a target component is needed, the bridge will have to be started first.

Third way to do bridging is to extend the server to serve several middlewares. This will not necessarily be much trouble if the service logic is separated from the interface implementations or if extended interfaces can forward calls to old interfaces. The maintenance cost is a problem, as when the original interface changes, extension changes always too.

4.6 Dynamic Bridge

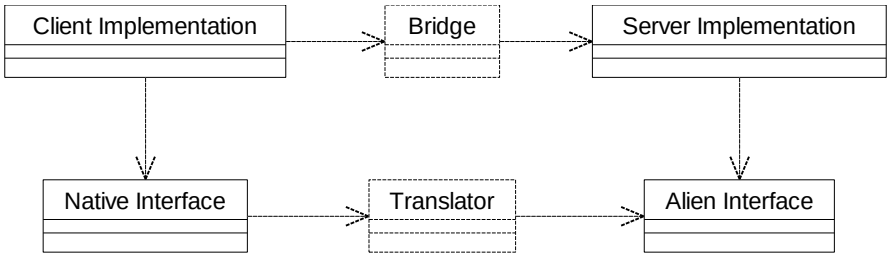
The point of using a dynamic bridge is that it can present itself as any server. Clients can be developed as single middleware applications, servers can be developed as single middleware applications and the bridge does not have to be developed.



17. Dynamic Bridge Relationships

Clients and servers can also be dynamic. A client could use the networking protocol to do an imaginary call not related to any interface. A server could handle any call regardless of what interfaces it implements. In the case where a client and a bridge are dynamic, all that such a bridge needs is the location of the servers and a standard way to map requests from a middleware to another.

It is quite safe to assume that some clients will be static. Static clients need compile time information about the server interface. The bridge needs to implement a tool that can map interfaces from one middleware to another.



18. Static Client Dependencies

A dynamic bridge itself does not have to be aware of where the client gets the information to call the server. A translator tool will have to use a mechanism that is synchronised with the bridge, so that the meaning of requests will not be lost in the bridge. Therefore dynamic bridge needs a standard mechanism for translating one middleware interface to another middleware interface both compile time and runtime.

4.7 COM/CORBA Interoperability

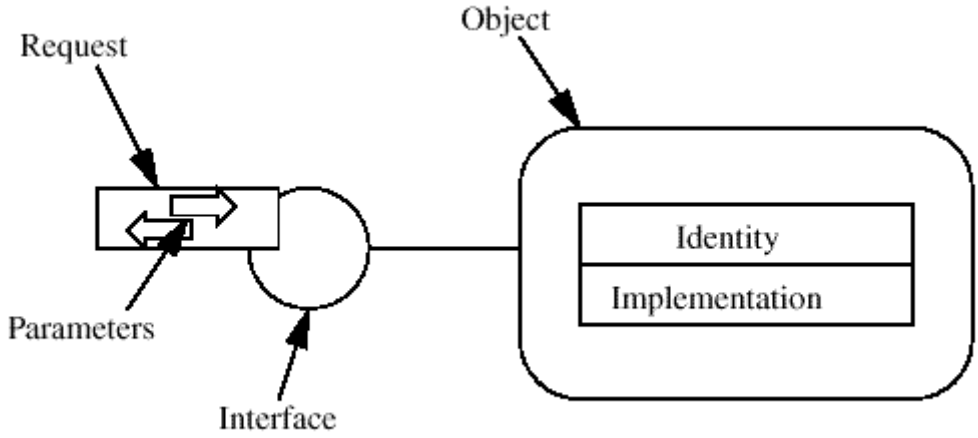
COM and CORBA are seemingly dominant middleware architectures of the day. COM/CORBA interoperability has also been properly approached in the form of interoperability specification belonging to the CORBA [OMG 1999 b] specification. This specification has been developed in co-operation with Microsoft and can be considered as the standard for COM-CORBA interoperability.

COM/CORBA interoperability specification separates COM/CORBA interoperability and Automation/CORBA interoperability. COM and Automation have

incompatible types and COM does not allow dynamic invocations on objects as Automation Dispatch interface does.

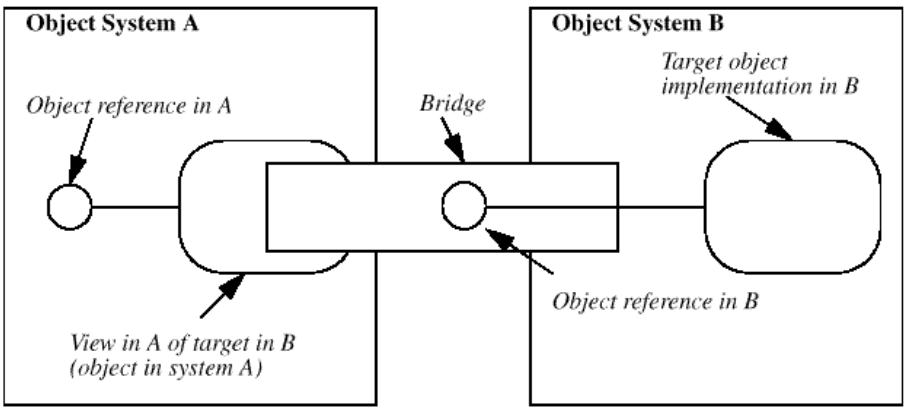
4.7.1 Interworking Architecture

The goal of the COM/CORBA bridges is to allow COM clients to use CORBA objects as if they were COM objects and vice versa. A bridge should be bi-directional, so that a COM object could be simultaneously a client to a CORBA object that is a client to that COM server.



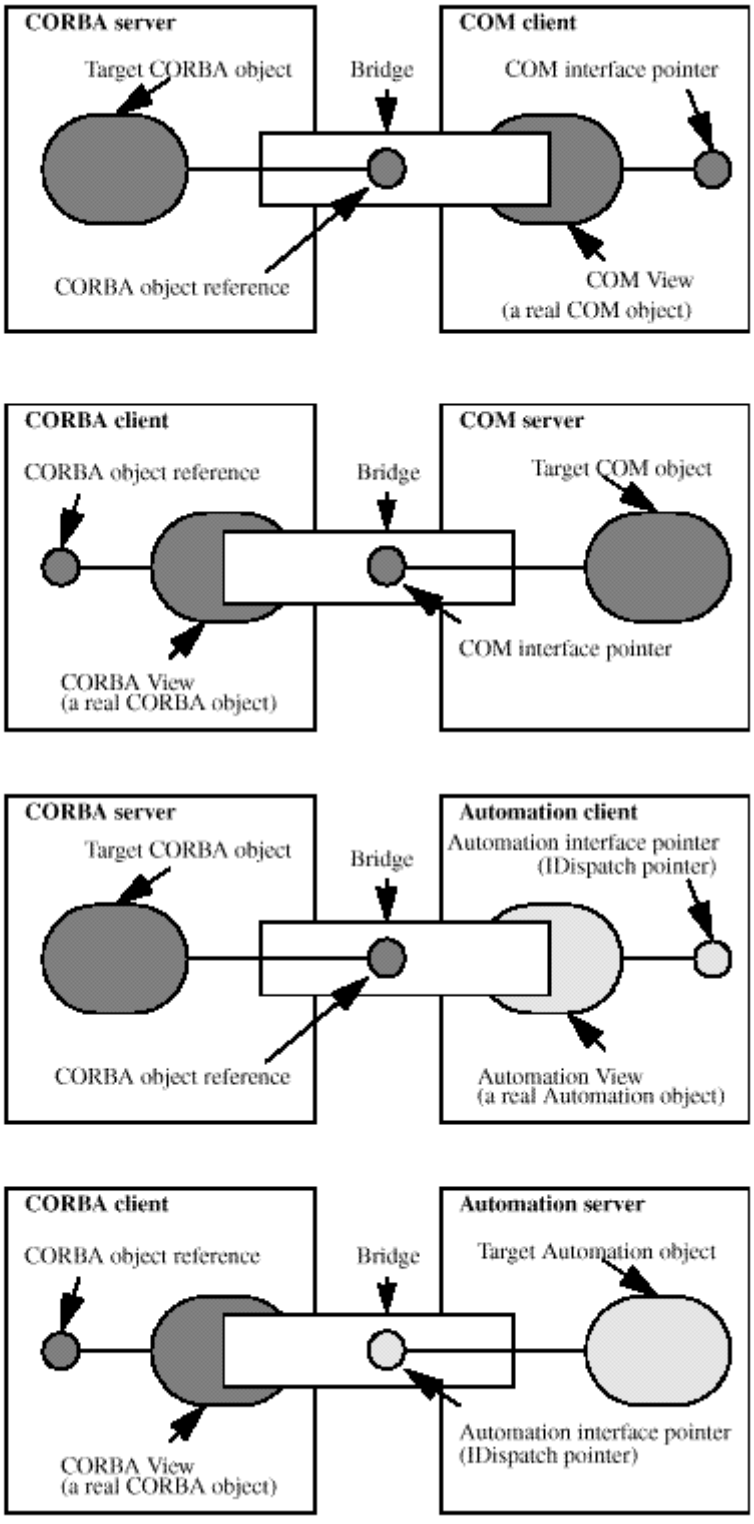
19. COM and CORBA Object Abstraction [OMG 1999 e]

On the high level COM and CORBA deal with objects on a very similar manner. Both separate interface from the implementation and both have quite similar object models. Object discovery models differ, but so do discovery models usually inside CORBA implementations.



20. Interworking Model [OMG 1999 e]

Figure 21 shows how this general Interworking model, (also presented in chapter Distribution Technologies), maps into COM/CORBA interworking architecture. The figure shows four different cases for the generic model.



21. Interworking Mapping [OMG 1999 e]

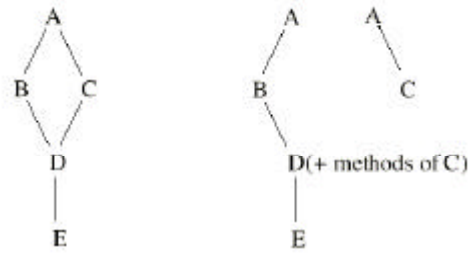
4.7.2 Mapping Issues

COM and CORBA both use interfaces. These interfaces will have to be mapped into the other object model and then implemented. In specification level the mapping only needs to be done but on interface level, implementation details can be left to vendor.

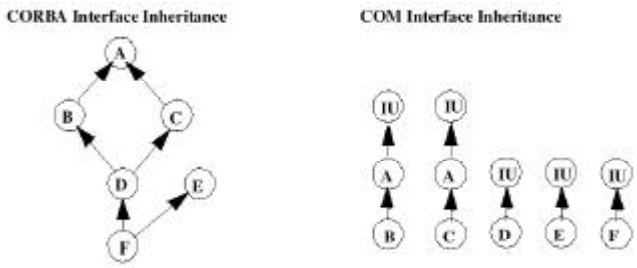
CORBA object references map into COM interface pointers. Most of the types, primitives and constructed types (structs, enums…) map closely. With CORBA/Automation there are some types which do not match. CORBA/Automation can work dynamically as both have dynamic invocation possibilities.

Each CORBA interface must be mapped into two separate interfaces. Automation and COM will have separate interfaces. There are two mapping specifications, one for COM/CORBA and one for Automation/CORBA. Both specifications are, of course, bi-directional.

CORBA allows multiple inheritance. COM does not allow multiple inheritance and neither does it allow implementation inheritance. CORBA multiple inheritance is mapped into COM single interface inheritance. COM interface aggregation, (COM version of multiple inheritance), is mapped into CORBA multiple inheritance. Automation does not allow multiple inheritance so CORBA interfaces must be mapped into multiple strands of single inheritance.



22. CORBA Inheritance Mapping to Automation



23. CORBA Inheritance Mapping to COM

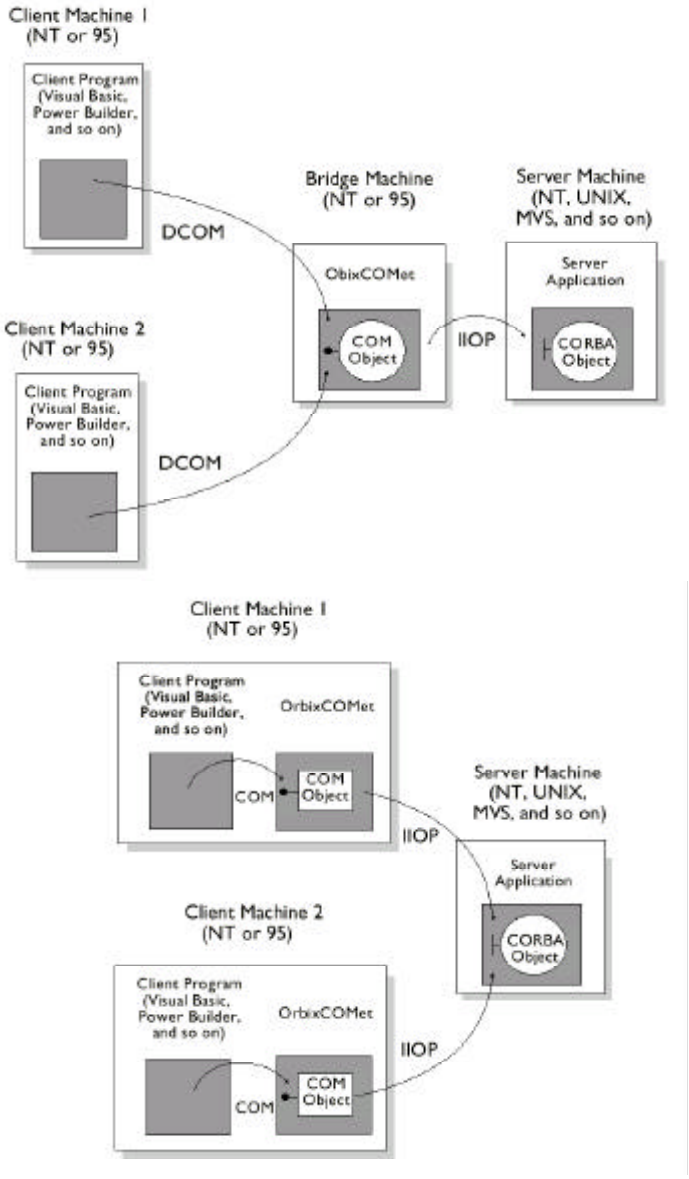
CORBA and COM identity must be matched. CORBA uses human readable names where COM uses unique identifiers. A CORBA object must have unique identifiers when used from COM.

COM does not have exception handling. CORBA exceptions are mapped into COM error codes and COM error codes are mapped into CORBA exceptions. CORBA exception additional data should still be available, so if it is accessed, the procedure will vary little from ordinary COM applications.

All interfaces should map back to the original interface when the specification is used in the other way.

4.7.3 Use Case

OrbixCOMet is a bridge implementing the interworking specification. OrbixCOMet is made by IONA Technologies, which is known, for their CORBA implementation Orbix. IONA is also one of the participants in OMG. OrbixCOMet matches original OMG interworking specification.



24. OrbixCOMet Bridge [IONA 1998]

OrbixCOMet is an application and a set of application tools. Before the client and the server are started, Comet must be started with appropriate configuration information, which it will use to find CORBA and COM systems. Tools are used to generate static mapping, if needed.

OrbixCOMet has been implemented only on Windows platform. This means that all applications are Windows dependable. Even if the developer uses both COM and CORBA on Unix platform, the Windows platform is needed in between.

OrbixCOMet is a dynamic bridge, allowing mapping between Automation and CORBA without compile time knowledge of either. CORBA interface information though must be stored in the CORBA Interface Repository (IFR). COM does not allow dynamic invocations, so COM->CORBA bridging always needs static stubs.

Mapping during runtime degrades system performance. If the bridge is on a separate machine from the client and the server, it also would naturally double network traffic.

4.8 Java Remote Method Invocation over Internet Inter-Orb Protocol

Remote Method Invocation [Javasoft 1998] over Internet Inter-Orb Protocol [OMG 1999 b] has been made by IBM and Sun as an Object Management Group (OMG) standard. It is defined by two documents, 'Java Language to IDL Mapping' [OMG 1999 d] and 'Objects by Value' [OMG 1999 a]. Objects by Value specification defines how to acquire states, actual values, of objects with CORBA.

The problems with RMI and CORBA interoperability are quite like ones with the COM [Microsoft 1995] and CORBA interoperability [OMG 1999 b] – how to express another languages powerful syntax on the other. Where COM could not match CORBA, CORBA can not match RMI. Mainly RMI allows more flexible exception handling and allows objects to be transferred, or copied, over remote invocations. 'Objects by Value' adds object transfer to CORBA and 'Java Language to IDL Mapping' takes care of the rest of the details.

The point in RMI over IIOP is that the RMI servers could be invoked as before with RMI clients. In addition with some extra work those RMI servers could be invoked with Java clients using CORBA and by another language clients using CORBA.

4.8.1 Objects by Value

Objects by Value [OMG 1999 a] specification assumes that it is reasonable that lightweight objects are transferred across remote calls, that it is not enough to get references to objects. An obvious example of a lightweight object is a string. Besides that a string object has the raw data, there is information on how that data should be handled. When a CORBA string is taken over in an invocation, only raw data is

transmitted, as both ends already know what kind of functionality a string has. The case with other lightweight objects is the same; only object state is transmitted, object functionality already exists on the other end or it is transmitted with some other mechanism.

Objects by Value (OV) is not Java specific like Java Language to IDL Mapping. Actually OV deals only with Java and C++ but it is in principle a general specification. OV extends the CORBA specification [OMG 1999 b].

The specification is not supposed to support transfer of complex and massive objects. It would naturally be more difficult to transfer object hierarchies for instance and even more difficult to make a general specification for that. So the specification is not like Java remote object transfer ported to other platforms.

4.8.2 Java Language to IDL Mapping

The Java Language to IDL Mapping (JLIM) [OMG 1999 d] specifies what the name says. Usually OMG specifications show how to implement IDL on a specific language, but JLIM shows how to make IDL from Java remote interfaces, Java exceptions and serializable objects.

JLIM does not map the whole Java language but defines a subset of Java called RMI/IDL that is mapped. Subset definition includes:

- one of the Java primitive types (integer, String and such),
- a conforming remote interface (RMI interface),
- a conforming value type (java.io.Serializable or java.io.Externalizable),
- an array of conforming RMI/IDL types,
- a conforming exception type,
- a conforming CORBA object reference type and
- a conforming IDL entity type (derived from CORBA types).

So basically JLIM maps RMI to CORBA so that RMI can also transfer CORBA references. So it is not a subset of RMI, but more like RMI appended with CORBA.

5 MUD in Eight Hours

Each technology presented in this book has its weaknesses and strengths. Weaknesses are not always relevant – usually weaknesses are present when the tool is used to do something it is not supposed to. Better way in my opinion is to present a technology through its strengths. Good way to present strength in an understandable level is to show what the technology or the tool does cleanly and neatly.

This chapter has examples of distributed applications. I have chosen tools for the examples based on my own interest. Applications are supposed to present strengths of these tools. Technologies, or tools, are in the following order: Transmission Control Protocol (TCP), Common Object Request Broker Architecture (CORBA) and Component Object Model (COM)-CORBA Bridge.

A TCP application is the simplest example of persistent client/server relationship that has any use I could think of. The CORBA example uses Object Management Groups Interface Definitions Languages power to the fullest, defining a multiplayer game in just a dozen lines. The COM-CORBA bridge shows how the CORBA game is used from an Automation Visual Basic client.

5.1 Transmission Control Protocol Example

TCP transfers bytes (eight bit sequences) as a stream usually using Internet Protocol (IP) packets underneath. TCP is connection oriented. As long as the TCP connection stays client can expect services and the server will know who are connected to it. Things got to be simple, or we end up building layers on top of layers. On the other hand there is lot of power in TCP with little work and performance stays high compared to more complex technologies with more overhead.

If I want to make something simple and that's easy for service provider to set up Ill use low level tools. This example's like that, it runs from a single command line and its so simple it fits on two pages.

This example is a centralised log service. Clients from all around the network can write their log to a single common location. As a client activates the service a line is automatically written to the log and when the connection is broken another line is written. Therefore a log could be used to determine when a client has crashed – on the other hand as long as the connection has not been broken we know that the client is alive. This is a very useful service in a large system where processes have to be monitored. It could have more features of course, but then it might not be so simple anymore.

Network communication could be hidden behind a proxy [Gamma et. al. 1994] pattern but we would lose all meaning on using TCP directly, as there are tools, (middlewares), to automate proxy generation. As we are not using middlewares that would take care of threading, it will have to be taken care by ourselves. The log server actually writes to the screen and in our advantage writing to the screen is thread safe in Java so threading does not become too complex. Each connection has its own thread and input from the clients is read a line at a time. Characters are expected and each end of line character is a marker for a log write.

LogServer.java

```
package end7.example.tcp;


import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.BufferedReader;
import java.io.FileOutputStream;
import java.io.DataOutputStream;
import java.net.ServerSocket;
import java.net.Socket;
import java.net.InetAddress;
```

```
import java.util.Date;

public class LogServer
implements Runnable{
  public static ServerSocket ss;
  public static DataOutputStream dos;

  BufferedReader bufferedReader;
  Thread thread;
  String name;
  Date date;

  public LogServer(){
    try{
      Socket s = ss.accept();
      InputStream is = s.getInputStream();
      InputStreamReader isr = new InputStreamReader(is);
      bufferedReader = new BufferedReader(isr);
      InetAddress ia = s.getInetAddress();
      name = ia.getHostName() + " " + bufferedReader.readLine();
      write("new client");
      thread = new Thread(this);
      thread.start();
    }catch(Exception e){
      e.printStackTrace();
    }
  }

  public static void main(String[] args){
    try{
      int port = Integer.parseInt(args[0]);
      ss = new ServerSocket(port);
      FileOutputStream fos = new FileOutputStream(args[1]);
      dos = new DataOutputStream(fos);
      while(true){
        new LogServer();
      }
    }catch(Exception e){
      e.printStackTrace();
```

```
    }
  }

  public void run(){
    try{
      while(true){
        write(bufferedReader.readLine());
      }
    }catch(Exception e){
      write("client leaving");
    }
  }

  public void write(String text){
    try{
      date = new Date(System.currentTimeMillis());
      String message = date.toString() + " " + name + "\n" + text;
      dos.writeBytes(message);
      System.out.println(message);
    }catch(Exception e){
      e.printStackTrace();
    }
  }
}
```

Class has four functions: constructor, main, run and write. Main function loops waiting for new connections. Constructors wait for connections and when connection is established, object construction finishes. Each object in turn loops in the run function, which is started on each object by the thread. All threads write to screen and log file using function write.

Client looks like the server. This is just an example how the server could be used, a test client to see that the server works. Client simply initialises standard TCP sockets and writes few messages. Most of the code is error handling, which is forced by Java.

LogClient.java

```
package end7.example.tcp;

import java.io.OutputStream;
import java.io.DataOutputStream;
import java.net.Socket;
```

```
public class LogClient{
  DataOutputStream dos;

  private LogClient(){
  }

  public LogClient(String name, String host, int port){
    try{
      Socket s = new Socket(host, port);
      OutputStream os = s.getOutputStream();
      dos = new DataOutputStream(os);
      write(name);
    }catch(Exception e){
      e.printStackTrace();
    }
  }

  public static void main(String[] args){
    try{
      System.out.println("Testing");
      int port = Integer.parseInt(args[2]);
      LogClient lc = new LogClient(args[0], args[1], port);
      Thread.sleep(5 * 1000);
      lc.write("hello world");
      Thread.sleep(5 * 1000);
    }catch(Exception e){
      e.printStackTrace();
    }
  }

  public void write(String text){
    try{
      dos.writeBytes(text + "\n");
      dos.flush();
    }catch(Exception e){
      e.printStackTrace();
    }
  }
}
```

5.2 CORBA Example

One day working with an IDL it occurred to me that it could be easy to implement a MUD with CORBA. Half a year after the idea I gave it a try and found myself with a playable game eight hours later. I made that version with RMI, but it was no different from this one below implemented with CORBA. To make the example more complete, I included a COM/CORBA bridge and a Visual Basic client.

The heart of the basic application is the IDL file. For me it was also the technical specification. IDL is so close to C++ and Java that it should be easy to understand with little study if either programming language is familiar.

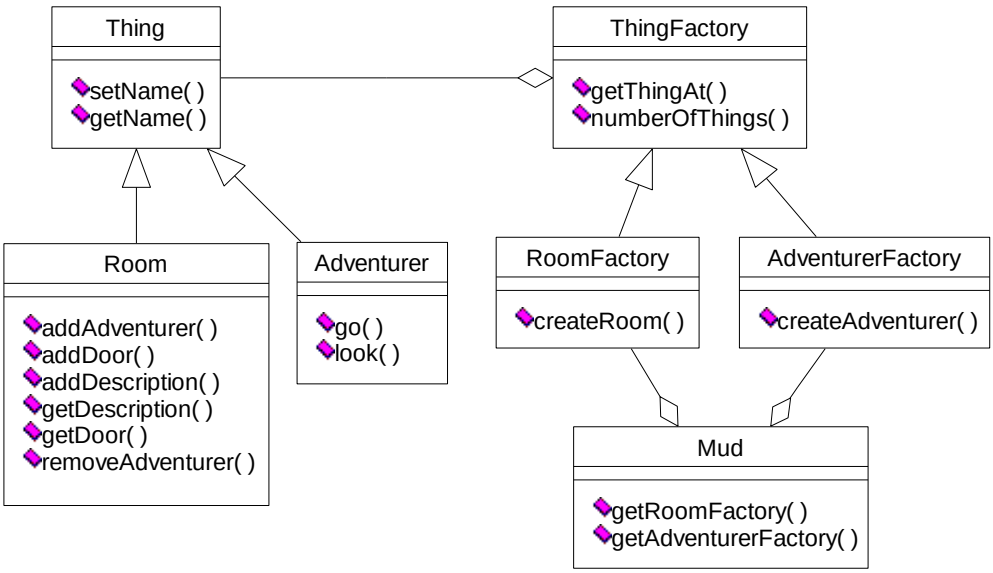
Mud.idl

```
module end7{
   module example{
         module corba{
               interface Thing{
                     string getName();
                     void setName(in string name);
               };
               interface Adventurer : Thing{
                     void go(in string room);
                     string look();
               };
               interface Room : Thing{
                     void addAdventurer(in Adventurer adventurer);
                     void addDoor(in Room room, in string description);
                     void addDescription(in string description);
                     string getDescription();
                     Room getDoor(in string description);
                     void removeAdventurer(in Adventurer adventurer);
               };
               interface ThingFactory{
                     Thing getThingAt(in long index);
                     long numberOfThings();
               };
               interface AdventurerFactory:ThingFactory{
                     Adventurer createAdventurer();
               };
               interface RoomFactory:ThingFactory{
```

```
                Room createRoom();
            };
            interface Mud{
                AdventurerFactory getAdventurerFactory();
                RoomFactory getRoomFactory();
            };
        };
    };
};
```

MUD contains rooms and adventurers. There is separate room factory and adventurer factory to divide code into smaller units. Common functionality is in the thing factory, which is abstract class. Also the common functionality of rooms and adventurers is in similar in the thing class.

Looking the IDL you should realise that two kinds of clients possible. One kind of client could have access to room factory and could create and modify rooms. Another kind of client would have access to adventurer factory and could create adventurer, which in turn would adventure in the rooms. Figure 25 shows the IDL in UML notation.



25. MUD class hierarchy

Next step from IDL is to use a code generator. Code generator creates proxies for each interface. I used JDK idl2java generator, which has decent documentation on how it is used. Each interface results as four files, so its better to generate them to a separate directory. Files are client stub, server skeleton, 'helper' to be used in both sides and an interface. Now we are ready to start the actual coding.

MudImpl.java

```
package end7.example.corba;
```

```
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;

/**
```

- Mud ‘controller’ class implementation

```
 */
public class MudImpl extends _MudImplBase{
  public AdventurerFactoryImpl afi;
  public RoomFactoryImpl rfi;

  public MudImpl(){
    afi = new AdventurerFactoryImpl();
    rfi = new RoomFactoryImpl();
    afi.rfi = rfi;
  }

  public AdventurerFactory getAdventurerFactory(){
    return afi;
  }

  public RoomFactory getRoomFactory(){
    return rfi;
  }

  /**
```

- Takes standard java ORB arguments

```
   */

  public static void main(String args[]){
    try{
      W.init();
      ORB orb = ORB.init(args, null);
      MudImpl mi = new MudImpl();
      orb.connect(mi);
      org.omg.CORBA.Object ns = orb.resolve_initial_references(“NameService”);
      NamingContext ncontext = NamingContextHelper.narrow(ns);
      NameComponent ncomponent = new NameComponent(“Mud”, “”);
      NameComponent ncpath[] = {ncomponent};
```

```
            ncontext.rebind(ncpath, mi);
            W.s(orb.object_to_string(mi));
            java.lang.Object s = new java.lang.Object();
            synchronized (s) {
              s.wait();
            }
          }catch(Exception e){
            e.printStackTrace();
          }
        }
      }
```

This is the main object. Following the code is started from the main function.

1. `W.init();` - initialising my own debug routines.
2. `ORB orb = ORB.init(args, null);` - Initialising ORB, arguments are taken directly from command line, usually includes just the port to be used.
3. `MudImpl mi = new MudImpl();` - Main object is actually created, so we jump to the constructor.
4. `afi = new AdventurerFactoryImpl();` - class that takes care of adventurers is created.
5. `rfi = new RoomFactoryImpl();` - class that takes care of rooms is created.
6. `afi.rfi = rfi;` - adventurer factory happens to need a reference to the room factory. And then back to the main function.
7. `orb.connect(mi);` - MUD is made available. After this it can be invoked.
8. `org.omg.CORBA.Object ns = orb.resolve_initial_references("NameService");` - Name Service is a standard way for making objects available.
9. `NamingContext ncontext = NamingContextHelper.narrow(ns);` - Example of helper class use. Abstract object is cast into a usable one.
10. `NameComponent ncomponent = new NameComponent("Mud", "");` - Entry which can be used in Name Service is created.
11. `NameComponent ncpath[] = {ncomponent};` - Name Service is built like a directory, so the entry must be stored in a path too.
12. `ncontext.rebind(ncpath, mi);` - Path and object are bound together.
13. `W.s(orb.object_to_string(mi));` - Textual representation of the MUD object is written to screen, this is needed later in bridging.
14. Thread goes to sleep. New threads will be initialised to serve as needed by the ORB implementation.

Package is derived from the IDL file, terminology simply changes from modules to package. – `package end7.example.corba;`

`import org.omg.CosNaming.*;` - This library has the naming service.

`Import org.omg.CosNaming.NamingContextPackage.*;` - This library has the entry code.

Import org.omg.CORBA.*; - Basic CORBA functionality.

Class inherits from the generated class, generated classes are named _<interface name>ImplBase.

Get functions explain themselves.

Rest of the server code is included at the end. There is no more new functionality that is involved with CORBA. As Mud, other server classes also derive from generated files, which in turn inherit from generated interfaces. Its time to turn attention to client side. As mentioned before there are two kinds of clients, mud's room controller is much simpler so it makes a better introductory example.

MudMaker.java

```
package end7.example.corba;

import org.omg.CosNaming.*;
import org.omg.CORBA.*;

public class MudMaker {


  public static void main(String args[]){
    try{
      W.init();

      ORB orb = ORB.init(args, null);
      org.omg.CORBA.Object ns = orb.resolve_initial_references("NameService");
      NamingContext ncontext = NamingContextHelper.narrow(ns);
      NameComponent ncomponent = new NameComponent("Mud", "");
      NameComponent ncpath[] = {ncomponent};
      Mud m = MudHelper.narrow(ncontext.resolve(ncpath));
      RoomFactory rf = m.getRoomFactory();

      Room white = rf.createRoom();
      Room black = rf.createRoom();
      Room yellow = rf.createRoom();
      Room red = rf.createRoom();
      Room blue = rf.createRoom();

      white.setName("White");
      black.setName("Black");
      yellow.setName("Yellow");
```

```
        red.setName("Red");
        blue.setName("Blue");

        white.addDescription("A white room with black curtains");
        black.addDescription("A black room with white curtains");
        yellow.addDescription("A room with yellow spots on the walls");
        red.addDescription("Blood red room");
        blue.addDescription("Room with glass walls the blue sea seen through");

        white.addDoor(black, "curtains");
        black.addDoor(white, "curtains");

        black.addDoor(yellow, "golden");
        yellow.addDoor(black, "golden");

        yellow.addDoor(red, "bronze");
        red.addDoor(yellow, "bronze");

        red.addDoor(blue, "stairs up");
        blue.addDoor(red, "stairs down");

        blue.addDoor(white, "trapdoor");

        W.s("MUD created, press enter");

        W.input.readLine();

      }catch (Exception e) {
     e.printStackTrace();
      }
    }
  }
```

Most of the code above is about making the five rooms and doors between them. All doors except the trapdoor from blue to white are two-way. That is why doors are made in pairs. But to the CORBA code.

1. `ORB orb = ORB.init(args, null);` - Client must initialise CORBA as the server.
2. `org.omg.CORBA.Object ns = orb.resolve_initial_references("NameService");` - Obtaining the naming service to which server stored its reference.
3. `NamingContext ncontext = NamingContextHelper.narrow(ns);` - Object is cast to usable form.
4. `NameComponent ncomponent = new NameComponent("Mud", "");` - Creating an component which will be recognised in Name Service.

5. NameComponent ncpath[] = {ncomponent}; - Putting the component into a directory representation.
6. Mud m = MudHelper.narrow(ncontext.resolve(ncpath)); - Reference is opened and cast into usable object.
7. RoomFactory rf = m.getRoomFactory(); - Function implemented in MudImpl, (shown above), is invoked.

The first call to the code has been made and the following calls are made in similar fashion. As MudImpl object created RoomFactory object, RoomFactory object creates Room objects when requested. Note how references are used to make the doors. It does not matter if those references were from the same server or not, so it would be easy to expand this game server on several different computers.

Here is rest of the room related server code, which is being invoked from the client.

RoomFactoryImpl.java

```
package end7.example.corba;


import java.util.Vector;


/**
```

- Implementation of 'Room creator and controller'

```
 */
public class RoomFactoryImpl extends _RoomFactoryImplBase{
  public Vector things;

  public RoomFactoryImpl(){
    things = new Vector();
  }

  public Room createRoom(){
    RoomImpl a = new RoomImpl();
    things.add(a);
    W.d("New room created");
    return a;
  }

  public Thing getThingAt(int index){
    return (Thing) things.get(index);
  }

  public int numberOfThings(){
    return things.size();
  }
```

```
}
```

RoomImpl.java

```
package end7.example.corba;

import java.util.Hashtable;
import java.util.Vector;
import java.util.Enumeration;

public class RoomImpl extends _RoomImplBase{
  public Hashtable doors;
  public String description;
  public Vector adventurers;
  public String name;

  public RoomImpl(){
    doors = new Hashtable();
    adventurers = new Vector();
  }
  public void addAdventurer(Adventurer adventurer){
    adventurers.add(adventurer);
  }
  public void addDoor(Room room, String description){
    doors.put(description, room);
    W.d("Door made from " + name + " to " + room.getName());
  }
  public void addDescription(String description){
    if(this.description == null){
      this.description = description;
    }else{
      this.description = this.description + "\n" + description;
    }
  }
  public String getDescription(){
    String s = new String("Room name:\n" + name + "\n");
    s = s + "Room description:\n" + description + "\n";
    s = s + "Doors:\n";
    Enumeration keys = doors.keys();
    while(keys.hasMoreElements()){
      s = s + (String) keys.nextElement() + "\n";
```

```
        }
        // There could be some checking so that adventurer
        // requesting data would not be printed, but…
        s = s + "Adventurers:\n";
        Enumeration adv = adventurers.elements();
        while(adv.hasMoreElements()){
          s = s + ((Adventurer) adv.nextElement()).getName() + "\n";
        }
        return s;
      }
      public Room getDoor(String description){
        return (Room) doors.get(description);
      }
      public String getName(){
        return name;
      }
      public void removeAdventurer(Adventurer adventurer){
        adventurers.remove(adventurer);
      }
      public void setName(String name){
        this.name = name;
      }
}
```

Now the 'administration' side of the game has been presented. Player side will connect to the game, as did the room generator, only it will connect to the adventurer factory instead of room factory. Rooms are kind of adventurer containers, and the go command in the adventurer interface, (seen in the IDL), will cause the adventurer to be moved from one container to another, simulating walking through the doors.

Rest of the code does not offer much on the educational side, as the main features have been shown already. There are the adventurer factory and adventurer still to be presented.

AdventurerFactoryImpl.java

```
package end7.example.corba;

import java.util.Vector;

/**
```

- Implementation for 'Adventurer creator and controller'

```
 */
public class AdventurerFactoryImpl extends _AdventurerFactoryImplBase{
```

```
  public RoomFactoryImpl rfi;
  public Vector things;

  public AdventurerFactoryImpl(){
    things = new Vector();
  }

  public Adventurer createAdventurer(){
    W.d("New adventurer");
    AdventurerImpl a = new AdventurerImpl();
    things.add(a);
    // adventurers start at the first created room
    a.residence = (Room) rfi.getThingAt(0);
    return a;
  }

  public Thing getThingAt(int index){
    return (Thing) things.get(index);
  }

  public int numberOfThings(){
    return things.size();
  }
}
```

AdventurerImpl.java

```
package end7.example.corba;

public class AdventurerImpl extends _AdventurerImplBase{
  public Room residence;
  public String name;

  public String getName(){
    return name;
  }

  public void go(String room){
    Room r = residence.getDoor(room);
    // there is no spectacular error handling
    if(r == null){
```

```
      return;
    }
    residence.removeAdventurer(this);
    residence = r;
    residence.addAdventurer(this);
  }

  public String look(){
    String s = residence.getDescription();
    return s;
  }

  public void setName(String name){
    this.name = name;
  }
```

It does not get any easier than this. A nice multiplayer game in a few pages of code. Of course there could be a few more features besides moving and looking in the adventurer interface…

We are still missing the client that is to be used for playing. I have made two clients, one with Java and one with Visual Basic. Visual Basic client is handled in the next chapter and here is the command line Java client.

MudClient.java

```
package end7.example.corba;

import org.omg.CosNaming.*;
import org.omg.CORBA.*;
import java.util.StringTokenizer;

public class MudClient {

  public static String help(){
    return "Commands:\n" +
        "help\n" +
        "go 'place'\n" +
        "look\n" +
        "quit";
  }

  public static void main(String args[]){
```

```
try{
  W.init();

  ORB orb = ORB.init(args, null);
  org.omg.CORBA.Object ns = orb.resolve_initial_references("NameService");
  NamingContext ncontext = NamingContextHelper.narrow(ns);
  NameComponent ncomponent = new NameComponent("Mud", "");
  NameComponent ncpath[] = {ncomponent};
  Mud m = MudHelper.narrow(ncontext.resolve(ncpath));
  AdventurerFactory af = m.getAdventurerFactory();

  Adventurer a = af.createAdventurer();
  W.s("Give name for the adventurer");
  a.setName(W.input.readLine());
  W.s(help() + "\n" + a.look());

  while(true){
    String c = W.input.readLine();
    if(c.equalsIgnoreCase("look")){
      W.s(a.look());
      continue;
    }
    if(c.equalsIgnoreCase("quit")){
      break;
    }
    if(c.equalsIgnoreCase("help")){
      W.s(help());
      continue;
    }
    StringTokenizer st = new StringTokenizer©;
    if(st.nextToken().equals("go")){
      String s = "";
      while(st.hasMoreTokens()){
        s = s + st.nextToken();
        if(st.hasMoreTokens()){
          s = s + " ";
        }
      }
      a.go(s);
```

```
            W.s(a.look());
            continue;
          }
          W.s("Unknown command:" + c);
        }
      }catch (Exception e) {
        e.printStackTrace();
      }
    }
  }
```

5.3 The COM-CORBA Bridge

What we need is the COM client to speak to the Mud server so that it understands. One obvious way is to use Microsoft Java development environment, in which COM has been integrated, and another is to use products based on OMG interoperability paper. With Microsoft development environment we would not be using CORBA either though.

Orbix COMet from IONA Technologies is a dynamic bridge on Windows.
Client communicates to IONA bridge and it communicates onwards with IIOP. Therefore we are platform and language independent.

Making the bridge is a process of using different tools in COMet and in CORBA. No changes are necessary to the server, but we do need the original IDL file. IDL is loaded to a standard IFR, and the bridge is instructed to download the information from the IFR. IFR is a standard runtime way to handle IDL information, so it is a way to avoid static methods.

Container in the bridge, which holds the same information as the IDL, is called type store. If the client would use COM dynamic dispatch, it could be written now. It is easier though to use static information during coding so we generate a TLB from the type store. TLB is referenced in the Visual Basic and the code is written. So once again:

- IDL to IFR
- Type store loads IFR data
- TLB is generated by the type store
- TLB is referenced in the Visual Basic
- Code is written:

game.frm

```
Dim cf As Object
Dim a As Diend7_example_corba_Adventurer
Dim af As Diend7_example_corba_AdventurerFactory
Dim m As Diend7_example_corba_Mud
```

```
Private Sub Form_Load()
   On Error GoTo errorHandling
   Set cf = CreateObject("CORBA.Factory")
   Set m = cf.GetObject("end7::example::corba::Mud:" + Connection.IOR.Text)
   Set af = m.getAdventurerFactory()
   Set a = af.createAdventurer()
   Description.Text = a.look()
   AdvName.Text = a.getName()
   Exit Sub
errorHandling:
   MsgBox (Err.Description + " " + Err.HelpContext + " " + Err.Source)
End Sub

Private Sub Go_Click()
   a.Go (Room.Text)
   Description.Text = a.look()
End Sub

Private Sub SetName_Click()
   a.SetName (AdvName.Text)
End Sub
```

This is just the code under the form and the buttons. Establishing connection is done as form is loaded, after that object is used as any COM object.

`Set cf = CreateObject("CORBA.Factory")` – This is the bridge COM side object.

`Set m = cf.GetObject("end7::example::corba::Mud:" + Connection.IOR.Text)` – We request the bridge for an initialised COM Mud proxy. We use IOR to make sure we get the right reference.

`Set af = m.getAdventurerFactory()` – And now we are already making calls to the actual game.

From the code it is obvious that there are two buttons and two text fields in the user interface: go button, set name button, room text field and adventurer name text field.

Form load function takes information from a previous form 'Connection', which has text field for Initial Object Reference [IOR].

Bridging seems simple, but it is much more difficult than working just with one middleware as there are three layers to be set up and run: server middleware, bridge middleware and client middleware. At least for me the time required to set up environment raised exponentially with each layer.

CORBA	Setting up ORB and server
Bridge	Setting up IFR and bridge, translating information from CORBA to bridge to COM
COM	Setting up client

26. Work Needed with Each Layer

6 Conclusions

My first encounter with middlewares was with CORBA in my previous job. It was an interface to telecommunications product. Customers could use the interface to make clients to suit their own needs. Then I dealt with DCE RPC, which was static mapping from the CORBA interface. One of the customers had RPC programmers so they wanted a RPC interface. Later I started to study possibilities of making COM clients to our system, mapping CORBA to COM. It was the original idea for this book.

In my current job we have three separate distribution technologies we use. One is TCP like telecommunications protocol. Another one is an HTTP like protocol to an old system. Third one is just plain CORBA.

While there has been variety with the technologies I have had to deal with, examples in this book show some variety too. For an example Microsoft COM is a very nice tool with a lot of features and it has surely been tested by millions of users in action. But facing the reality – who is going to use COM on Linux platforms? If you want to make something like my log service, why double the administration and development time by using a middleware? And other way around – other MUD's have been done on top of TCP – I wonder if the developers of those applications had something to run after a day.

Sometimes you just might have to put up a CGI gateway because your network operator will not open ports or what ever. Maybe the sales are done on based if the customer can make her own Visual Basic clients? Maybe dynamic bridging just is not fast enough and one has to switch to static ones. It is hard to survive with a single tool in different situations, even impossible.

Making my point, this view I have taken to look distribution technologies with wide perspective has value in the same way as investigating one technology in depth. Where one technology could be investigated thoroughly to find solutions from it to all problems, the whole field can be looked to find the tools that have apparent solutions to the problems.

Another point I have made is that I think that middlewares can be used together. Middlewares do not only work together with the use of bridges, but as they have different uses co-operation with different middlewares could be preferred.

It is clear to me that there is advantage in using these tools in combinations. Thinking that a project will be done using just C++ and CORBA is a little narrow-minded. Depending on the complexity of the project a wider diversity of tools might be needed. Also if the developer group has special skills with some technologies, it might be ideal to try to build the project with those tools.

There is probably someone who is ready to bet her credibility on saying that one of these tools is over the others. She is probably a sales person with heavy interest on selling the product. She could as well say that you should do all the work with a hammer

and forget the screwdriver and rest of the toolbox. I am quite content saying that if you have a job, the tools presented in this book are a good way to start.

References

[Bowen 1998] Ted Smalley Bowen, "*Microsoft Makes Interoperability Strides*", *InfoWorld*, Vol. 20, Issue 5, 2 February 1998.

[Brando 1995] Thomas J. Brando, *Comparing DCE and CORBA*
http://www.mitre.org/research/domis/reports/DCEvCORBA.html).

[Coar et. al.] Coar, K. and D. Robinson, *The WWW Common Gateway Interface Version 1.1* (CGI/1.1), Work in Progress).

[Distinct 1999] *Distinct® ONC RPC/XDR Toolkit for Java™,* 16.2.1999 Distinct Corporation.
http://www.distinct.com).

[Eng 1999] Ben Eng, *Matrix of CORBA services,*
Appendix A,
http://www.vex.net/~ben/corba/cosmatrix.html
15.12.1999).

[Foley 1997] Mary Jo Foley, *Microsoft Extends it's Object Model Beyond Windows*
http://www.techweb.com/se/directlink.cgi?CRN19970825S0045).

[Gamma et. al. 1994] E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley, 1994.

[Internet RFC] *Internet Request For Comments*:
http://www.faqs.org/rfcs/
01.04.1999).

[IONA 1995] *Orbix 2 Programming Guide*, IONA Technologies, 1995).

[IONA 1998 a] *OrbixCOMet Programmers Guide*, IONA Technologies PLC, 1998).

[IONA 1998 b] *Orbix 2*, Iona Technologies PLC, 1998).

[Javasoft 1997a] *Java Native Interface Specification 1.1*,

http://www.javasoft.com/products/jdk/1.2/docs/guide/jni/spec/jniTOC.doc.html 21.03.1999, 1997).

[Javasoft 1997b] *Java Beans Specification*, http://www.javasoft.com/beans/docs/spec.html, 26.06.2000).

[Javasoft 1998] *JavaTM Remote Method Invocation Specification*, http://www.javasoft.com/products/jdk/1.2/docs/guide/rmi/spec/rmiTOC.doc.html 21.03.1999, 1998).

[Javasoft 1999a] *IDL to Java compiler, Sun Microsystems*, http://developer.java.sun.com/servlet/SessionServlet?url=http://developer.java.sun.com/developer/earlyAccess/jdk12/idltojava.html 15.12.1999, 1999).

[Javasoft 1999b] *Java 2 Development Kit*, Sun Microsystems, www.javasoft.com 14.01.1999, 1999).

[Kain 1998] Kain, J. Bradford, *Software Components as Application Building Blocks*, www.quoininc.com/quoininc/ComponentsABB.html 21.03.1999, 1998).

[Kirtland 1997] Kirtland Mary, *The COM+ Programming Makes it Easy to Write Components in Any Language*, *Microsoft Systems Journal*, December 1997.

[Kruchten] Philippe Kruchten, *A Rational Development Process*: Rational Software Corp. Vancouver, B.C., Canada. http://www.rational.com/sitewide/support/whitepapers/dynamic.jtmpl?doc_key=334 01.04.1999).

[Microsoft 1995] *The Component Object Model Specification*, Microsoft Corporation, Redmond WA, 1995).

[Mowbray & Malveau, 1997] Mowbray, Thomas J.; Malveau, Raphael C., *CORBA Design Patterns. John Wiley & Sons* 1997).

[Newmarch 1999] Jan Newmarch, *Jan Newmarch's Guide to JINI Technologies* http://pandonia.canberra.edu.au/java/jini/tutorial/Jini.xml,

23.12.1999).

[OMG 1998 a] *Mapping: OLE Automation and CORBA*, OMG document 98-02-22).

[OMG 1998 b] *Mapping: COM and CORBA*, OMG document 98-02-21).

[OMG 1998 c] *CORBA*, OMG document 98-12-09).

[OMG 1998 d] *Java Language to IDL mapping*, OMG document 99-03-09).

[OMG 1998 e] *Interworking Architecture*, OMG document 98-02-20).

[OMG 1999 a] *Objects by Value*, OMG document 98-01-18).

[OMG 1999 b] Object Management Group, *Common Object Request Broker: Architecture and Specification*. Object Management Group electronic publishing, revision 2.3: June 1999.

[OMG 1999 c] *OMG Unified Modeling Language Specification 1.3* (draft), OMG, 1999).

[OSF 1996] *DCE 1.2 Contents Overview*,
OSF RFC 63.3, 1999).

[OSF 1997] *DCE 1.1: Remote Procedure Call*,
http://www.opengroup.org/onlinepubs/009629399/toc.htm
06.01.2000).

[Rumbaugh, 1996] Rumbaugh, James, OMT Insights. *SIGS* 1996).

[Tallman & Kain] Tallman, Owen. ; Kain, Bradford., *COM versus CORBA: A Decision Framework*. Distributed Computing, September-December, 1998).

[Reaz Hogue 1999] *CORBA 3*, IDG Books, 1999).

[Sun 1999] *Jini Specifications, 1.0.1*, Sun Microsystems, 1999).