# CORBA: A middleware for an heterogeneous cooperative system

Alberto Abelló
Dept. Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya

May 21, 1999

**Abstract**

Two kinds of heterogeneities interfere with the integration of different information sources, those in systems and those in semantics. They generate different problems and require different solutions. This paper tries to separate them by proposing the usage of a distinct tool for each one (i.e. CORBA and BLOOM respectively), and analizing how they could collaborate. CORBA offers lots of ways to deal with distributed objects and their potential needs, while BLOOM takes care of the semantic heterogeneities. Therefore, it seems promising to handle the system heterogeneities by wrapping the components of the BLOOM execution architecture into CORBA objects.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 Needs of integration

Companies are faced with rapidly evolving technology which offers a wealth of new opportunities for those companies that can move swiftly and evolve quickly. The faster a company adapts its structure to the ever changing market, the more it rises its benefits.

Within this frame, information had become a cornerstone in business. Everybody needs to know what is happening in order to react. This means an enterprise needs to take advantage of its information systems in order to gain competitivity. Probably, the needed information is already there, you just have to find and use it in the proper way. This model of business requires new and creative uses of computer technology.

That needed information should be obtained at the lower cost. Nowadays, while the cost of hardware is decreasing, the cost of software is increasing. This makes people think about reusing the existing applications instead of just throwing them away and doing everything from scratch. Reusing seems to be a solution.

However, those already existing applications were not exactly thought as interoperating between them. The problem is even worst if the applications are running on a network of different machines with different operating systems. Moreover, every application could have been done using a different programming language, as well. Diversity in hardware and software is a fact of life. As explained in [Vin97], heterogeneity is the result of several factors: different people across an enterprise often choose different solutions to similar problems; consumers tend to buy the systems that best fulfill their requirements at the most reasonable price, regardless of who makes them; over time, purchasing decisions accumulate, and already-purchased systems may be too critical or too costly to replace. You need tools to harmonize the pieces of this puzzle, to solve those heterogeneities.

## 1.2 A distributed cooperative system architecture

In order to make all the applications in a company or group of companies work together, a really good glue is needed. Since it will be for sure a distributed and heterogeneous set of systems, a "supersystem" hidding its actual complexity will have to be built on top of the existing applications. This proposed extra layer is called "Federation" in [SL90] and presents the set of information systems as a whole (to the federated users at least).

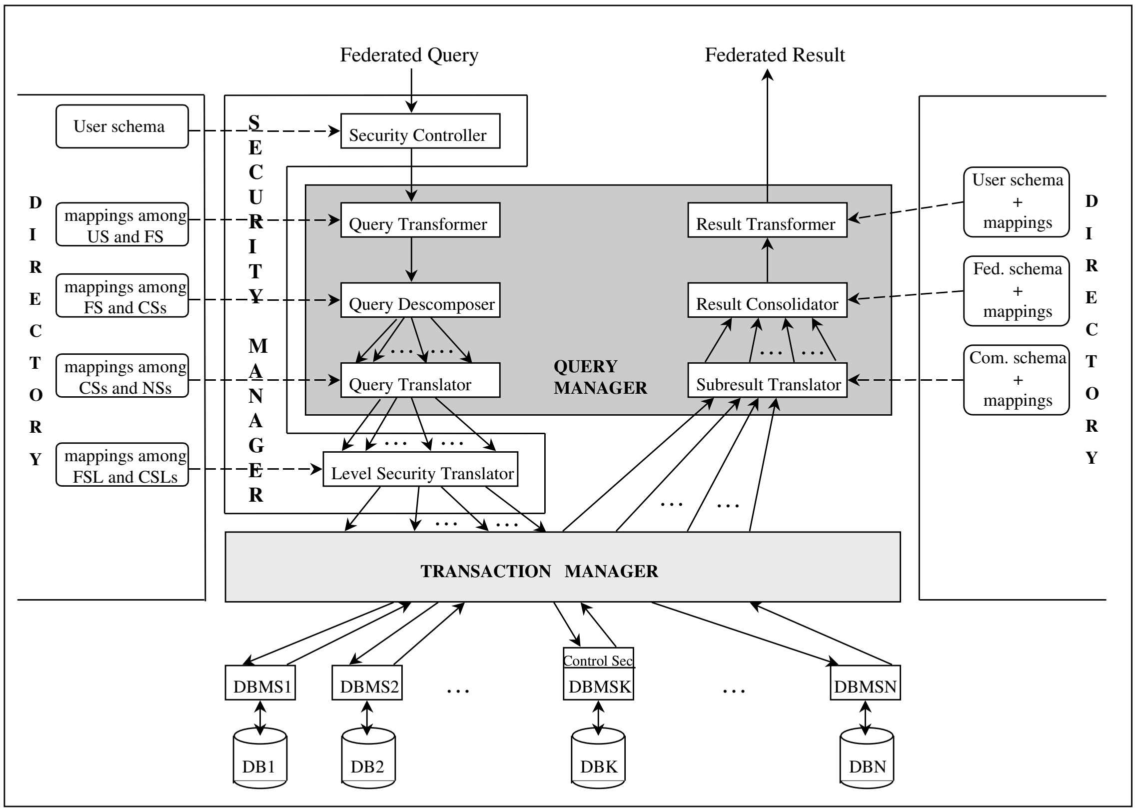


Figure 1.1: Reference execution architecture [ROSC97]

The last end of this paper is finding the way to build the federation by means of the architecture described in [ROSC97]. Figure 1.1 describes the process followed by a given query in that Federated DataBase Management System (FDBMS).

This architecture allows two kinds of queries: those coming from federal users and those that the local users send to their local databases. We will focus in the former since the local queries are not an extra problem because the federation preserves the autonomy of the component databases.

Firstly, the *Federated Query* arrives to the *Security Controller* which checks the *User Schema* to see if it is authorized at its security level. If the query is authorized, it passes to the *Query Transformer* to translate it from the *User Schema* (expressed in the model chosen by the user) to the *Federated Schema* (expressed in the canonical model, let's say BLOOM). Once the query is expressed in terms of the Federated Schema, it is decomposed by the *Query Decomposer* into subqueries, each one of them accessing exactly one of the *Component DataBases* (CDBs). Before sending the subqueries to the CDBs, they should be expressed in terms of the respective *Local Schema*. This translation is done by the *Query Translator*. Finally, the *Level Security Translator* changes the federated security level tags by the given component security levels tags.

The *Transaction Manager* assumes the responsibility for executing the subqueries on each CDB and getting the results from them, following the rules in a given transaction model.

Once the *Transaction Manager* has got the subresults, these must be stuck together to build the result of the original *Federated Query*. This means, more or less, undo the steps to decompose the query, but skiping the security controls. Firstly, the subresults have to be translated from the given local model to the canonical data model (done by the *Subresult Translator*). The subresults expressed in the canonical model are passed to the *Result Consolidator* that obtains the final result. This is not enough since the result coming out from the *Result Consolidator* must be expressed in terms of the federated user model. The *Result Transformer* does that and, finally, obtains the *Federated Result*.

When you have this complex architecture, the problem is how to build it. It could be the case that all the databases are running on a single mainframe, what would make it much more easy. However, that assumption is not realistic. The different CDBs will be wide through a, maybe, huge network, running on different machines and with full autonomy. It could even be the case that some of those machines hosting the CDBs, are neither accepting any modification nor any extra piece of software to be connected to the federation. Those CDBs could be migrated from one machine to another, too. New CDBs could join the federation as well as some DBs could leave it.

Summing up, we need a really flexible system that allow us to tie together all those components and modules while let us scatter and modify them if necesary. Dealing with heterogeneity is rarely easy.

## 1.3 Distributed object systems

Heterogeneity is not a bad thing if you know how to handle it. It enables us to use the best combination of hardware and software components for each portion of the enterprise. It is just a problem of having a standard protocol for interoperability and portability between the components.

Probably, the most important proposed solution is the distributed object systems. The well-known advantages of objects are of use here. It is, roughtly, a client-server architecture with code encapsulated into independent objects which can be placed anywhere in a network. As defined in [OMG95], an object system is a collection of objects that isolates the requestors of services (clients) from the providers of services (servers) by a well-defined encapsulating interface. As components, those isolated objects can be used as building blocks for larger applications (just bigger objects).

Object-Oriented software is easier to modify than software written using other techniques. Some interfaces or implementations could be changed without requiring global changes. This is really important at the enterprise level, where changes are prohibitively expensive to make.

Some different attempts had been shipped to the market. Of course, Microsoft proposed its Component Object Model (COM) and Object Linking and Embedding (OLE). Independently of its advantages, it is a proprietary solution and, definitely, that is not good for an heterogeneous and flexible system. It would go against one of the main requirements of a federation: no assumptions about the CDBs.

Another distributed object system is Remote Method Invocation (RMI), a Java solution [Sri97]. Despite it is a proprietary system as well (Sun Microsystems), Java is intended to run everywhere. Besides, it does not try to compete against anybody but, rather, fill holes left by others.

RMI lets us create a Java object on a given machine and communicate with it as you normally would, because it is no more than an extension of Java itself. Its pros are its cons: both ends (i.e. client and server) must be written in Java and this forces us to re-implement our legacy applications, which is exactly what we were trying to avoid. Another problem with RMI is that objects are not automatically started upon invocation. The RMI registry must be started by hand as well as every server object. Moreover, it locks you into a Java only solution. Take into account Java is interpreted rather than compiled and it means slowness.

Don't worry, the perfect solution is already here. Its name is CORBA, which stands for Common Object Request Broker Architecture. It is an open solution controlled by the members of the Object Management Group (OMG). Almost all (either small or big) computer related companies are associated to this organization (i.e. Digital, Object Design, IONA Technologies, Sun Microsystems, Hewlett-Packard,

... but Microsoft is not).

As explained in [Bak97], CORBA has two aims. Firstly, it makes easier to implement new applications that must place components on differents hosts on the network, or use different programming languages. Secondly, it encourages the writing of open applications, which might be used as components of larger systems. Each application is made up of components, and integration is supported by allowing other applications to communicate directly with these components.

CORBA is not an implementation standard but a communication protocol. It says how the object interface must be, rather than how it should be implemented. It means you just need to know what you will give to an object and what you will get from the object in return. Therefore, if some objects share the same interface, they are interchangeable. Those interfaces are defined using the Interface Definition Language (IDL).

The CORBA standard does not say anything about the implementation of the objects. They could be implemented even in a non-Object Oriented language (e.g. C). In order to implement an object in a given language, you just need a correspondence between IDL and the language. Up to now, correspondences had been defined with C, C++, Smalltalk and Java (that's it, the portability of Java and the acceptance of CORBA can be mixed together). Correspondences with other languages will be defined in the future.

CORBA does not exclude the use of other solutions (i.e. COM/OLE, Java/RMI, etc.), rather it makes easy to use them. It is a big umbrella which shelters everything and everybody. As a consensuated solution, it has the potential of subsuming every other form of existing client/server middleware. For instance, CORBA integration with OLE and Java is described in [Bak97], [OH98] and [KS98].

The aim of this paper is not presenting and comparing different distributed object architectures but finding a promising way to build a federated architecture. To this purpose, it seems much better not to be tied to a proprietary solution. Therefore, we will focus on CORBA because its broad acceptance and its standard role.

## 1.4 Outline of the paper

1. Introduction: Gives a brief overview of the problem, and introduces the BLOOM execution architecture as well as some distributed object systems.
2. Object Management Group solution: Explains the OMG history and its proposed architecture (OMA). The OMA is broken into small pieces, step by step. Firstly, the general architecture is introduced, afterwards CORBA is explained, the different services listed, finishing by giving

concrete examples of the connection to the CORBA system, and a brief overview of some ORB implementations.

3. CORBA in heterogeneous DBMSs: Presents how CORBA is (or could be) used in building three different FDBMSs (i.e. HEROS from Brazil, MIND from Turkey, and finally our beloved BLOOM).
4. CORBA in the persistence of data: Analizes the different ways to get data persistence in a CORBA system, and tries to apply them to the BLOOM execution architecture.
5. Conclusions: contains a reflection on the "political" problems of CORBA, and on how BLOOM and CORBA match.

In the last pages you can find a glossary (with the acronyms, and main terms used), the bibliography, and the IDL CORBA interfaces of a couple of interesting services.

## 1.5 Acknowledgments

This work was partially supported by a grant of the Comissionat per a Universitats i Recerca de la Generalitat de Catalunya and the Spanish PRONTIC programme (under project TIC96-0903).

# Chapter 2

# Object Management Group solution

## 2.1 History

Due to the increasing amount of programming interfaces and packages in the market and the lack of a standards to facilitate the integration of systems in a distributed heterogeneous environment, the Object Management Group (OMG) was founded in April 1989. It is a consortium of computer-involved companies, located in Framingham (Massachusetts, USA), currently supported by over 700 members, including information system vendors, software developers and users. Those members range from giants of hardware and software industry to tiny companies barely starting up. At present, it has become the largest information technology consortium in the world.

As can be read in [OMG95], OMG's objectives are to foster the growth of object technology and influence its direction by establishing the Object Management Architecture (OMA). The OMA provides the conceptual infrastructure upon which all OMG specializations are based. Primary goals are the reusability, portability, and interoperability of object-based software in distributed, heterogeneous environments.

The OMG does not actually produce any software, only specifications. In order to agree on the standard protocols and interfaces, the OMG issues Requests For Proposals (RFPs) and Requests For Information (RFIs), asking for detailed specifications of extensions to the standard. As a response to those requests, each member (or subset of members) can submit its proposal. Each submission must specify how the standard should be extended and how the extension is going to be used (defining neutral IDL interfaces). It must not define the implementation of the given extension.

A period of time (around a year) begins for each proposer to complete the submission, or merge it with the proposals from others. During that period of time, the proposals use to be merged into a single proposal, which is then voted by the OMG members. If that is not the case, and there are more than one final proposal, one is selected by successive vote of the Task Force and the Technical Committee. Finally, the OMG Board of Directors declares the successful proposal to be the official specification.

The standardization process encourages the participation of everybody and drives them towards consensus. Although, at the beginning of it, every company works individually or in small groups, a single agreed proposal is the most common ending. For example, the original CORBA specification was a consensus submission from six companies, and the first set of CORBA services was cosubmitted by eleven. The "Common" in CORBA stands for two API proposals: one coming from HyperDesk and Digital, based on a dynamic API; and another based on a static API coming from Sun and Hewlett-Packard. That is why CORBA supports dynamic as well as static method invocation, both were mixed together into a single response.

Once a standard has been agreed, companies that submit the successful proposal must ship a commercial implementation within a year. At the same time or later on, it can be implemented by any company (OMG member or not), without paying anything to the OMG. The OMG itself does not implement any of its standards, and it stays neutral between the different vendors providing implementations. However, it has committees to maintain and change the specification if needed. That does not mean the standard changes every day. It is not subject to frequent changes.

As can be inferred from this process and who is in it, the OMG adopts specifications based on commercially available object technology. They try to be stuck to realities and get implementations in the market as soon as possible. A full example of the process is found in [Ses96].

An *Object Management Architecture Guide* was published in 1990. The first CORBA specification was adopted by the OMG in October 1991, and its first full implementation was released in July 1993. CORBA 2.0 was adopted in December 1994. During 1996, the Internet Inter-ORB Protocol (IIOP) was adopted to allow the communication between different implementations of the ORB (possibly running on different machines). All of it is rapidly growing up.

## 2.2 Bill of purposes

- Some desirable general properties of the object system we are talking about are listed in [Bak97]:
  - Objects must be simple to create.
  - Objects must be of any size.
  - Access to objects must be efficient.

- It should allow you to invoke methods on server objects using your high-level language of choice.
- An object must be accessible from any programming language (either OO or not).
- Many programming languages must be supported.
- Objects must be accessible across a network.
- It should provide local/remote transparency.
- An object must be accessible from any operating system, not just the one that the object itself is running on.
- Objects must be supported by the current commercial operating systems.
- Objects must be accessible from Web-based clients.
- The message details must be hidden.
- It should provide polymorphic messages (they should be pointed to a concrete object).
- An interface must be simple to implement.
- Easy encapsulation of existing applications must be provided (by means of the separation of object definition and implementation).
- The system must interact with other object-oriented systems (e.g. ODBMS).
- The system must be able to work with non-object-oriented systems (e.g. RDBMS).
- Invocations must be secure.
- It must be easy to give objects high-level symbolic names.
- It must be possible to find an object by giving a description of it.
- Distributed transactions must be supported.
- It must be possible to obtain type information at runtime, the system should be self-describing.
- It must allow both, static and dynamic method invocations.
- The system must have wide industrial acceptance.
- The system must be based on a non-proprietary standard.
- It must be available from multiple suppliers.

• A much more pragmatic list singing the praises of the OMA is in [Sie96]:

  - The object-oriented paradigm meshes with software "best practice" from the start of the development cycle to the end, when the objects are deployed in a distributed object environment.
  - It maximizes programmer productivity: provides a sophisticated base, with transparent distribution and easy access to components.
  - Developers create or assemble application objects, taking advantage of every component.

– Programmers can build new objects by making incremental modifications to existing ones without having to recode the parts that already work.

– Helps code reuse in new or dynamically reconfigured applications.

– The system lets you take advantage of all the tools you have bought, from hardware to development software. Give them an interface and a thin layer of wrapper code, and legacy applications come into the environment on an equal basis with your new software components.

– You can mix and match tools within a project, using any for a given kind of component (i.e. GUI) and another one for the rest (i.e. business code).

## 2.3 General architecture

The task of the OMG was huge and hard: they where trying to put the software industry on the right track. If they wanted to get the support of the vast majority of the companies in the market, they needed a general architecture where everybody fit and which clearly mark the route to follow. With the idea of giving a solid and general framework that define the frontiers and interfaces between different producers, they began defining the OMA.

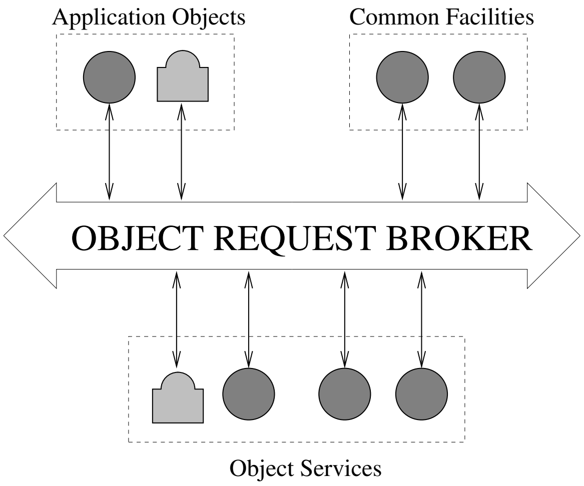


Figure 2.1: Object Management Architecture schema [OMG95]

The OMA provides the conceptual underlying structure, terms and definitions upon which all OMG specifications are based. It is composed of an *Object Model* and a *Reference Model*. The *Object Model* defines an object as an encapsulated entity with a distinct identity whose services can be accessed only through well-defined interfaces. In [OMG95] you can find a description of the different components of the

*Reference Model* (i.e. *Object Services*, *Common Facilities* and *Application Objects*), depicted in figure 2.1:

**Object Services,** a collection of services (interfaces and objects) that support basic functions for using and implementing objects. Services are necessary to construct any distributed application and are always independent of application domains. It includes the lower level and most crucial object interfaces. They are though as being used by other objects. A short description of available services is listed in section 2.5.

**Common Facilities,** a collection of services that many applications may share, but which are not as fundamental as the *Object Services*. They are though as being used directly by the applications. An example of *Common Facility* could be a management system for e-mail. Two kinds of facilities are distinguished:

**Horizontal Facilities,** which can be used by virtually every business (enterprise-wide).

**Vertical Facilities,** which standardizes management of information specialized to particular industry groups (industry-specific).

**Application Objects,** which are products of a single vendor on in-house development group which control their interfaces. *Application Objects* correspond to the traditional notion of applications, so they are not standardized by OMG. Instead, *Application Objects* constitute the uppermost layer of the *Reference Model* and they are though as being used directly by the end-user.

**Object Request Broker** is the core of the model, which enables objects to transparently make and receive requests and responses in a distributed environment. It is the foundation for building applications in heterogeneous environments. It keeps all together, is like a engine that makes work the other parts of the model. Its detailed description is found in section 2.4.

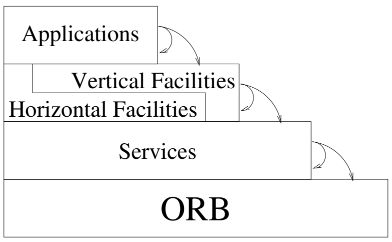


Figure 2.2: OMA components calls

These four or five components could be simply seen as in figure 2.2, where any set of objects requests services mainly through those interfaces in its own level or in the level just below it. That is probably the unique difference between applications, facilities and services: how general, reusable and near to the end-user they are.

The OMA gives a overall view of the proposed architecture and how everything interacts. It does not actually concrete anything, it is pretty general. Each one of its parts needs to be specified in much more detail. The task of the OMG is standardize the underlying object architecture and the interfaces to the object services and facilities. It has been conceived as a core (i.e. CORBA) and a growing number of standards (i.e. CORBA services, and CORBA facilities) that extend the core.

Since all together is just a standard, and no description about its implementation is given at all, each part of it can be provided by a different vendor. Each service, facility or application object in your system could have been implemented by an absolutely different maker. Even it is possible having a different ORB running on each one of your machines.

## 2.4 Broker architecture

Zooming into the OMA, we find the Common Object Request Broker Architecture (CORBA) which describes in much more detail how everything works and the available tools to get the desired results. While the OMA tries to group the different kinds of objects that will exist inside a system attending to its genericity in usage (who calls who), CORBA describes the way those objects are going to work together. It is based on a client-server architecture where clients ask for services and the servers offer perform those services.

CORBA separates the specification of a service offered by a given object from its actual implementation. To avoid breaking encapsulation, distributed OO applications must deal only with object interfaces and should not care whether the object implementations are in the same process or on another machine.

Shipping objects across the network is not advisable. Obviously, it would break object encapsulation by accessing private data, not to mention it would need compiler-specific knowledge about how objects are laid out in memory and that would affect portability. Therefore, the server code and data remain in its machine, and it is the method call and result who travel between client and server.

The architecture proposes a self-describing system where everything is designed as flexible as possible based on the interoperation on an object bus (the Object Request Broker). This even allows intelligent components to dynamically discover each other, they just need access to the bus. It is a star architecture with the ORB sited in the middle and the objects hanging from it. In figure 2.3, you can see the different parts of CORBA and the ORB as defined in [OMG95]. A much shorter overview of the architecture is

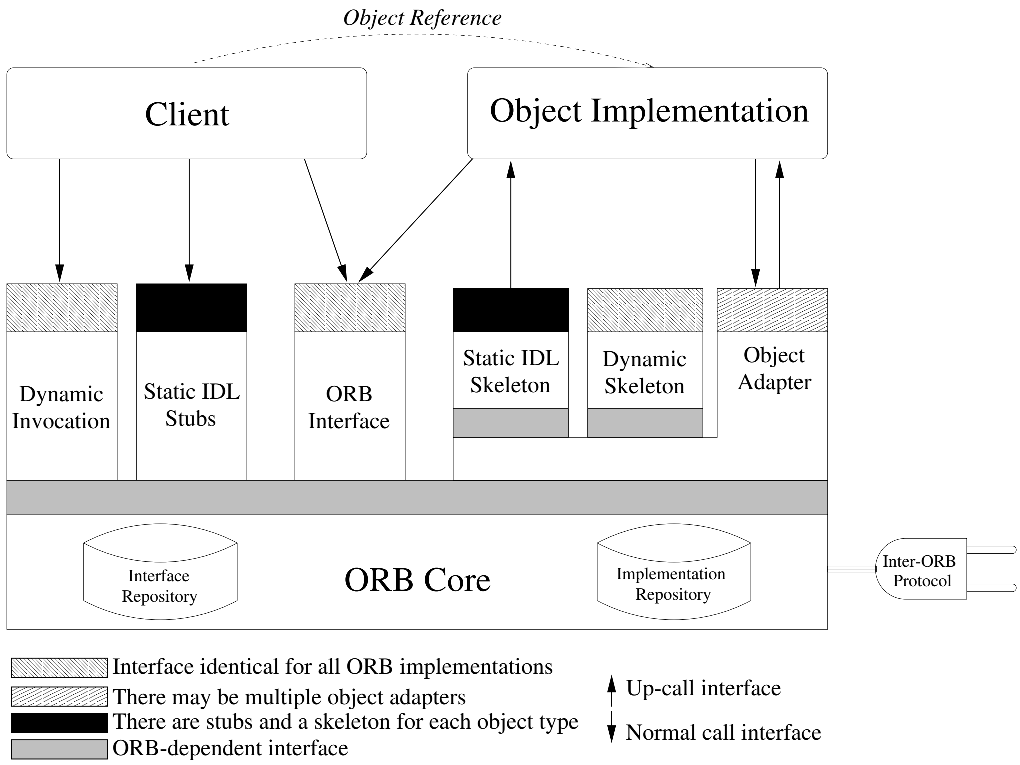


Figure 2.3: Common Object Request Broker Architecture

in [Vin93].

- Client side:

  **Static IDL Stubs,** interface known at compile time that allows invocation of requests to objects. They are like local proxies for a possibly remote server object. Each object should define its specific one and distribute it to its clients to be linked with them. A more detailed explanation is in 2.4.1.

  **Dynamic Invocation Interface,** the interface that allows dynamic creation and invocation of requests to objects. It is the same interface independently of the target object's interface. A more detailed explanation is in 2.4.2.

- Server side:

  **Static IDL Skeleton,** interface known at compile time that allows the ORB to pass a request to a given implementation of an object. It is also used to get the result of the request, if any. Each object defines its specific one and should be linked with it. A more detailed explanation is in 2.4.1.

**Dynamic Skeleton Interface,** the server's side interface that can deliver requests from the ORB to an object implementation that does not have compilation time knowledge of the type of object it is implementing. A more detailed explanation is in 2.4.2.

**Object Adapter,** describes the primary interface that an implementation uses to access ORB functions. Its roles include finding the target object of an invocation, determining what operation to call, and actually making the call on the target object. An ORB could offer multiple object adapters to support any style of object implementation (e.g. Basic Object Adapter, Portable Object Adapter, Object Database Adapter, etc.).

- ORB components:

**ORB Interface** describes the interface to the ORB functions that do not depend on object adapters. These operations are not many, but they are common for all ORBs and object implementations.

**Interface Repository** manages and provides access to a collection of object definitions at runtime. It keeps object interfaces, the methods they support, and the parameters and return values they require. This information is used by the ORB as well as the clients (remember the DII). CORBA defines a standard APIs for looking up the meta-data that defines a server interface.

**Implementation Repository** keeps the runtime relations between objects and implementations (i.e. the classes a server support, the objects that are instantiated, their IDs, etc.). It allows the ORB to locate and activate implementations of objects.

**Inter-ORB Protocol** allows the communication and interaction between different ORBs, even if they are running on different machines.

**Object Reference** identifies an object running on a given ORB. It does not depend on the *Client* nor *Object Implementation* but on the ORB. The ORB uses the references to identify and locate objects so that it can direct requests to them. An *Object Reference* always refers to the same object for which it was created (while it exists) and cannot be modified by the client. However, it can be stored and restored from storage systems. Object References can be made persistent by first asking the ORB to convert them to string. Clients can store these string object references in their own private data files and later retrieve them, ask the ORB to change them back into object references, and use them to make requests. This capability can be used to maintain persistent links between objects and the applications that use them.

The *Client* is who wants to perform an operation on a given object and the *Object Implementation* is the code which actually does what is asked. The implementation defines the data for the object instances and the code for the object's methods. It is important to notice that the *Client* role is relative to a particular object (i.e. the implementation of one object could be the client for others). The *Client* is just the piece of code that makes requests of objects.

To make the request, the *Client* can use the DII or an specific stub. In either case, the *Client* needs an *Object Reference* which identifies the object that will answer. The *Object Implementation* that corresponds to the *Object Reference* receives the request coming from the ORB through either its specific *Static IDL Skeleton* or the generic *Dynamic Skeleton*. CORBA has no special object creation operations, therefore, object references are always obtained by making requests on other objects (i.e. an object factory, the *Naming Service* or the *Object Trade Service*).

The ORB is responsible for finding the corresponding object implementation to a given request, wherever it is sited (it could even establish a connection to a different ORB which control the demanded implementation). The location of the implementation is absolutely transparent to the *Client*, as well as its programming language. Both, programming language and location, are hidden by the ORB. An ORB is plugged to other ORBs by means of a Inter-ORB Protocol. Using that protocol, the ORBs interact between them to give access to the implementations controlled by any of them from any other.

The clients and the implementations could directly interact with the ORB through the *ORB Interface* to access some of its functions. Besides, the implementations can call the *Object Adapter* by using its interface. The *Object Implementation* is closely connected with the *Object Adapter*. Generally, the *Object Implementation* does not depend on the ORB or how the *Client* invokes the object, but on the kind of *Object Adapter* chosen. If the same ORB has different object adapters, the object adapter to be used will be chosen by the *Object Implementation* based on the services it requires. The *Object Adapter* provides:

- Generation and interpretation of object references.
- Method invocation.
- Security of interactions.
- Object activation and deactivation.
- Mapping object references to implementations.
- Registration of implementations.

The *Client* has no knowledge of the implementation of the object, which object adapter is used by the implementation, or which ORB is used to access it. Summing up, by means of all the described mechanisms, the ORB hides to the *client*:

- Location: The client does not need to know where the object implementation is.
- Implementation: It does not know how that implementation works, neither.

- Execution state: It does not matter whether the object is currently activated and ready to accept requests. The ORB transparently starts the object up if necessary before delivering the request to it.
- Communication mechanisms: The *Client* is not aware of the communication mechanisms used by the ORB to deliver the request to the *Object Implementation* and return the response to it.

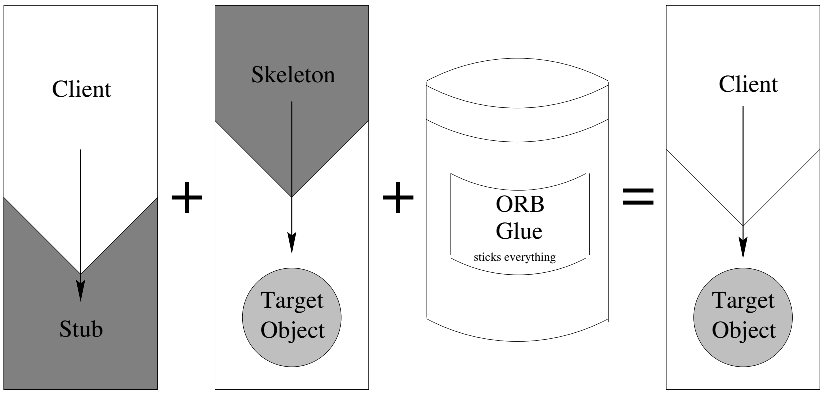


Figure 2.4: ORB role

What really matters is that the ORB must be transparent for the client requesting a service from a given object. The communication work is done by the ORB and neither the client nor the target object need to be aware of the existence of the ORB and the communications problems it is solving. As depicted in figure 2.4, the ORB could be seen as the glue that sticks together the *Client* and the *Target Object* regardless of where they are actually sited.

### 2.4.1 Knowing interfaces at compile time

Every object interface should be defined by means of the Interface Definition Language. The IDL defines the types of objects according to the operations that may be performed on them and the parameters to those operations. From that definition, the specific stub and skeleton for the given object and *Object Adapter* are created.

If that interface is known at compile time, it should be linked at both sides (i.e. *Client* and *Object Implementation*). Once that is done, it does not matter if the implementation of the object is in the same address space or running on a remote machine, if it was made with the same programming language or with a different one. The CORBA objects will look exactly like any other object in the client. You just need to call their methods with the corresponding parameters and get the return values if any.

When a call is done, it goes to the ORB through the stub. The ORB looks its repositories to find an implementation of the object. If that implementation is not running, the ORB automatically starts it up. Finally, the request arrives to the implementation through the skeleton. The implementation performs the request and the result goes the way back to the *Client*.

The obvious advantages of this approach are:

- Is easier to program.
- Provides type checking at compile time.
- Gives "good" performance.
- Clearly shows what the code is doing.

### 2.4.2 Discovering interfaces on the fly

CORBA also contemplates the impossibility of knowing the object interface at compile time (either by the *Client* or the *Object Implementation* itself) and describes how this problem is solved. It is even possible to have one of the sides linked with the needed stub or skeleton and the other not. From the *Client* point of view, it is only matter if itself knowns the stub or should use the DII (it is not important how the request will arrive to the *Object Implementation*). Conversely, from the *Object Implementation* point of view, it is only matter if itself knows the skeleton or should use the DSI (it is not important how the request arrived to the ORB).

While calling a method using a stub is transparent to the client, using the DII, the *Client* must specify the *Object Reference*, the name of the method to be invoked and the list of parameters to pass to that method. The returned value is of type *any*.

That dynamic call is done by means of standard objects always present in the ORB (i.e. *CORBA::ORB*, *CORBA::Object*, *CORBA::Request* and *CORBA:NVList*). Mainly, the methods *CORBA::Object.create_request*, *CORBA::Request.add_arg* and *CORBA::Request.send* are used. The needed information about the desired method to feed those calls (e.g. number of parameters, types, ...) can be retrieved from the *Interface Repository* using *CORBA::Object.get_interface*.

The usefulness of the DII is as clear as daylight. A client could look for the best object offering what it needs and call it without any knowledge at compilation time. However, the benefits of using the DSI are not clear enough. DSI works exactly as DII does: it allows the ORB to call methods on objects without knowing their interfaces. When is that going to be useful? Well, probably it is never going to be useful at all ... for clients. It was added to the CORBA specification to allow some cases found while trying to implement generic ORB bridges, that is, interaction between different ORBs. Of course, this kind of calls are absolutely transparent to the end-user.

Static invocations are easier to program, faster and self-documenting. Dynamic invocations provide maximum flexibility, but they are difficult to program; they are very useful for implementing tools to discover services at runtime.

## 2.5 Services

The OMG planned the development of the OMA as an iterative process. It begun giving the general OMA reference model and making it more specific in the CORBA architecture. Once that has been done, the stress is on the concrete services specifications. The next step will be standardizing facilities and having full CORBA compliant applications.

The CORBA services are simply a collection of system-level services. They try to complete the services directly offered by the ORB itself. Their specification is absolutely independent of their implementation, which allows to easily connect to the bus and use all the services it offers. Using or implementing the interfaces defined by the CORBA services, every ordinary object can be made lockable, persistent, secure, transactional, ... It is not mandatory to use any of this services, you should only use them if it saves time to implement your objects.

The services actually are standard interfaces which guaranty the interoperability between different implementors. They are a contract between the vendor and the customer, so you can get what you want from where you like the most, and use it without any problem at all. This policy implies every service could be implemented by an absolutely different company without any knowledge of each other, and the services would work smoothly together. It means real software reusability.

A list of desirable architectural goals for these services being developed can be found in [OMG94]:

- Scalability.
- Portability.
- Performance.
- Security.
- Precise descriptions.

- Independence and modularity.
- Minimum duplication of functionality.
- No hidden interfaces (allow absolutely implementation swapping).
- Consistency among different object services (they should be able to work together).
- Extensibility of individual object services (inheritance, delegation, ...).
- Extensibility of the set of services (allow adding new services without affecting the already existing ones).
- Configurability (allow different combinations of services, the ones using the others).

### 2.5.1 Up till now

The CORBA services specification is not well established, yet. Some service definitions are more advanced than others. Moreover, since it is iterative, it is probably a never-ending process. The current services specification is described in [OMG98]. Here they are listed ordered by the date their proposals where requested.

#### Life Cycle Service

Defines conventions for creating, deleting, copying and moving objects. They allow to perform life cycle operations on objects in different locations. It is based on a factory model. A factory is simply an object that creates another object returning it on demand. It is not a special object at all.

#### Naming Service

Can be used by the servers to bind a name to any of its objects, relative to a naming context (set of name bindings in which each name is unique) and independent of the name of the server or the host it is running on. To resolve a name is to determine the object associated with the name in a given context. In other words, the naming service associates a human-recognizable name with its concrete ORB-specific object reference.

### Persistent Object Service

Provides a set of common interfaces to the mechanism used for retaining and managing the persistent state of objects on a variety of storage servers (ODBMSs, RDBMSs, flat files, etc.). The object ultimately has the responsibility of managing its state, but can use or delegate to the *Persistence Object Service* for the actual work. Its usage is described more deeply in section 4.1.5, while its interfaces are listed in appendix A.2.

### Event Notification Service

Provides basic capabilities to allow different kinds of message or event delivery over an *event channel*. There can be multiple consumers and multiple suppliers of events connected to that channel. Suppliers can generate events without knowing the identities of the consumers and consumers can receive events without knowing the identities of the suppliers, they just need to know the *event channel*. It supports push and pull event delivery models: consumers can either request events or be notified of events. It also allows components to dynamically register or unregister their interest in specific events.

### Transaction Service

Supports multiple transaction models, including flat and nested models. It includes network interoperability to allow a transaction service interoperate with a cooperative transaction service using different ORBs. It provides two-phase commit protocol between the databases and a coordinator.

### Concurrency Control Service

Enables multiple clients to coordinate their access to shared resources. Coordinating access to a resource means that when multiple, concurrent clients access a single resource, any conflicting actions by the clients are reconciled so that the resource remains in a consistent state. It is regulated with locks associated with single resources and a single client. Several lock modes had been defined to allow flexible conflict resolution.

### Relationship Service

Allows relationships between CORBA objects (that know nothing of each other) to be explicitly represented and traversed. The service defines two new kinds of objects: *relationships* and *roles*. A *role* represents a CORBA object in a relationship. Together with the *Life Cycle Service*, it allows to copy, move, and remove graphs of related objects in some different ways.

### Object Externalization Service

Defines protocols and conventions for externalizing and internalizing objects. Externalizing an object is to record the object state in a stream of data so that it can be internalized into a new object in the same or different process. Its usage is more deeply described in section 4.1.4, while its interfaces are listed in appendix A.1.

### Security Service

Protects individual objects or groups of objects, so that only suitably privileged users can call specified operations on them. This service specifies different security functionalities:

**Identification/Authentication** of principals to verify they are who they claim to be.

**Authorization/Access control** deciding whether a principal can access an object.

**Security auditing** to make users accountable for the security related actions. Auditing mechanisms should be able to identify the user correctly, even after a chain of calls through many objects.

**Security of communication** between objects, which is often over insecure lower layer communications. The *Security Service* provides encryption mechanisms.

**Non-repudiation** provides irrefutable evidence of actions such as proof of origin of data to the recipient, or proof of recipient of data to the sender to protect against subsequent attempts to falsely deny the receiving or sending of the data.

**Administration** of security information.

#### Time Service

Enables the user to obtain the current time together with an error estimate associated with it. It generates time-triggered events based on timers and alarms. It can be used to compute the interval between two events as well as synchronize time in a distributed environment.

#### Object Query Service

Allows users and objects to invoke queries on collections of other objects. It is based on existing standards for query, including SQL-92, and OQL-93, as well as the upcoming SQL3 specification. This service provides an architecture for a nested and federated service that can coordinate multiple, nested query evaluators, and return a single object or a collection of objects. Its usage is more deeply described in section 4.1.6, while its interfaces are listed in appendix A.3.

#### Licensing Service

Provides a mechanism for producers to control the use of their intellectual property and be compensated for its use. A license has three types of attributes that allow producers to apply controls flexibly: time, value mapping, and consumer. Time allows licenses to have start/duration and expiration dates. Value mapping allows producers to implement a licensing scheme according to units, allocation, or consumption. Consumer attributes allow a license to be reserved or assigned for specific entities. It supports charging per session, per node, per instance creation, and per site.

#### Property Service

Provides the ability to dynamically associate named values with objects outside the static IDL-typed system. It defines operations to create and manipulate sets of name-value pairs or name-value-mode tuples. The names are simple IDL *string*s and values are IDL *any*s.

#### Object Trade Service

This service could be seen as a kind of "Yellow Pages" where the clients look for a description of what they want to do. The objects publish the services they are offering and the potential clients can look for whatever they need. It could seem similar to the *Naming Service*, however, the difference here is the client does not even know the name of the desired object or if it exists (the *Naming Service* is more similar to a telephone directory).

#### Object Collections Service

Allows to group objects which, as a group, support some operations and exhibit specific behaviour that are related to the nature of the collection (i.e. set, bag, queue, stack, list, tree, etc.) rather than to the type of objects it contains.

### 2.5.2 To do list

Some other services have not been requested for proposals, yet, but they are in mind. Some of them are listed here in alphabetical order.

#### Archive Service

Copies objects from an active/persistent store to backup store and vice versa. It will use the *Object Externalization Service* to get the internal state of objects.

#### Backup/Restore Service

Provides recovery after a system failure or a user error. It is closely related to the *Transaction Service*. In order to record a given state, it will use either the *Object Externalization Service* or the *Persistent Object Service*.

#### Change Management (Versioning) Service

Supports the identification and consistent evolution of objects including version and configuration management. This service should work with the *Persistent Object Service* to allow persistent objects to evolve from old to new versions.

**Data Interchange Service**

Enables objects to exchange some or all of their associated state. This service should work with *Persistent Object Service* to allow state to be exchanged when one or more of the objects are persistent.

**Internationalization Service**

Extends the *Naming Service* to better support representing and resolving names for some languages and cultures.

**Implementation Repository**

Supports the management of object implementations. The *Persistent Object Service* may depend on this to determine what persistent data an object contains.

**Interface Repository**

Supports runtime access to IDL-specified definitions such as object interfaces and type definitions. The *Persistent Object Service* depends on this to determine if a persistent object supports certain interfaces.

**Logging Service**

Implements the abstract notion of an infinitely long, sequentially-accessible, append-only file. It typically supports multiple log files, where each log file consist of a sequence of log records. It is related to the *Transaction Service* (for undo and redo), the *Change Management Service* (to support recovery) and the *Concurrency Service* (to keep track of locks).

**Recovery Service**

Responsible for keeping record of the changes made to the state of recoverable objects during a transaction and undo the updates if the transaction rolls back.

**Replication Service**

Provides explicit replication of objects in a distributed environment and manages the consistency of replicated copies.

**Startup Service**

Enables requests to automatically start up when an ORB is invoked. It supports bootstrapping and termination of the *Persistent Object Service*.

## 2.6 Using the Object Request Broker

All the beauty described has no-sense without the ORB. It is what makes everything work together, the core of the architecture, the cornerstone without which everything would fall down, the bus where all objects are connected. It makes possible objects interacting without any knowledge of each other.

Most of the CORBA services and facilities could be implemented on the ORB, although a small number either must be integrated into the core or require any extensions to it. Again, the idea is keeping it as simple as possible, and separating different concepts. Therefore, the ORB is clearly separated from the concrete services.

As always in CORBA, it does not specify how the ORB must be implemented but which are its interfaces. This means it could be implemented as different pieces given the desired functionality rather than as one block of concrete. If it offers the required interfaces, it does not matter its implementation. There may be multiple ORB implementations which have different representations for object references and different means of performing invocations. Some possible ORB implementations proposed in [OMG95] are:

**Client- and Implementation-resident ORB,** some routines are linked with the clients and the implementations to allow the communication between them.

**Server-based ORB,** all clients and implementations can communicate with one or more servers whose job is to reroute requests from clients to implementations.

**System-based ORB,** the ORB could be provided as a basic service of the underlying operating system.

**Library-based ORB,** the implementations might actually be in a library. Therefore, the stubs would be the actual methods.

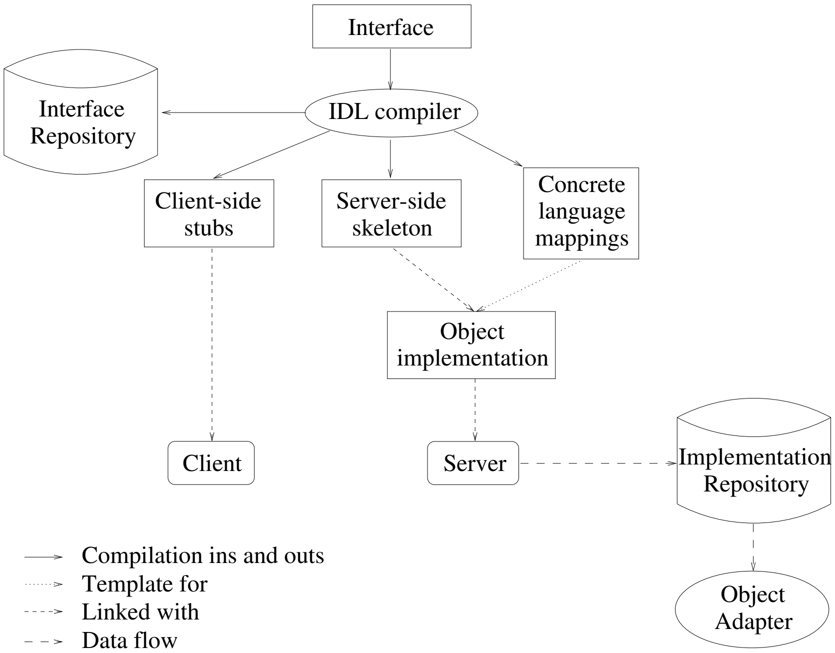


Figure 2.5: Code generation process

Independently of how the ORB is implemented, the process to get an object implementation accessible to its potential clients is described in figure 2.5.

### 2.6.1 The Interface

The interfaces are the joint between the ORB and the objects. They are a contract between clients and implementations, what the client can ask and what the implementation have to provide. At the same time, based on these contracts, the ORB puts the client into touch with the requested object. They let the communications infrastructure know the format of all messages the object will receive and send. The contracts are written using the *Interface Definition Language*, which is purely descriptive (it does not say how but what).

The valid requests a client can make on a given object are defined by the interface or interfaces the object supports. An interface contains some number of operations, any of which can be requested by the client. In the interface definition, we fix the functionality of an object. However, we do not fix:

- The programming language used to implement it.

- The platform it will run on.
- The ORB it will connect to.
- Whether it will run local to its clients or remotely.
- The network hardware or protocol it will use, if remote.
- Security aspects.

Both, client stubs and server skeletons, are generated by the IDL compiler, starting from the interface definition. IDL provides operating system and programming language independent way to define interfaces to all the services and components (a language mapping of IDL to the given programming language should have been defined). You just need the proper IDL compiler which generate code (stub and skeleton) - dependent of your ORB, OA, programming language, and machine - to connect the objects (clients and implementations respectively) to the ORB.

```
module Counter {
    interface Count {
        attribute long sum;
        long increment();
        }
    }
```

```
package Counter
public interface Count extends CORBA.Object {
    public int sum() throws CORBA.SystemException;
    public void sum(int _val) throws CORBA.SystemException;
    public int increment() throws CORBA.SystemException;
    }
```

Figure 2.6: Example of interface mapped to Java [OH98]

In figure 2.6 you can see how an IDL interface definition looks like, as well as its mapping to Java (some service IDL interfaces are also listed in appendix A). Its syntax is quite similar to C++, and is fixed in [OMG95]. It is actually a subset of the proposed ANSI C++ standard with additional constructs to support the operation invocation mechanism. Mappings had been defined to C, C++, Smalltalk and Java, up to now.

An IDL specification consists of one or more type definitions, constant definitions, exception definitions, module definitions, or interface definitions. Since the first three do not need any explanation, just to say that a module is used to scope IDL identifiers.

An interface definition consist of header and body. The header contains the name of the interface and its inheritance specification if any (implicitly derives from *Object* which is defined in the *CORBA* module). IDL allows multiple inheritance. However, the operations cannot be redefined in derived interfaces, and there is no notion of implementation inheritance (just interface inheritance). Object implementations are free to utilize any inheritance features of their implementation languages, independent of IDL inheritance.

The body contains constant declarations, type declarations, exception declarations, attribute declarations, and a set of named operations and the parameters to those operations. Empty interfaces are permitted. The parameters to the operations, as well as the attributes, can be basic types (i.e. *integer*, *boolean*, *char*, *octet*, *any*, etc.), template types (i.e. *sequence*, or *string*) constructed types (i.e. *struct*, *union*, or *enum*), or user-defined interfaces. The parameters can be *in*, *out* or *inout*.

### 2.6.2 The Object Implementation

Once the negotiation is finished and the contract is signed, this is almost a filling gaps exercise. The IDL compiler will automatically generate:

- Server-side skeleton.
- Client-side stub.
- Mappings to the corresponding language (see figure 2.6).
- Code template (optional).

The mappings are the IDL interface translated to the desired language, and the optional "code template" is a file where the programmer can add the concrete code implementing the services offered by the object (possibly using services offered by other CORBA objects connected to any ORB). Now it is time to fill the body for every service (method), implement the full object and publish it on the ORB.

The actual implementation of the object is not a problem, since it is just implementing the interface generated by the IDL compiler as any other in the implementation language used. It does not matter the object will be connected on the ORB. Its implementation will look exactly like any other object except for the interface it implements (the object interface) and the class it extends (the skeleton). Thus, the object implementation by itself is not enough to connect to the ORB. Some extra code (*Server*) is needed to create the object instances of a concrete implementation and loop waiting for requests. This *Server* must be registered in its turn in the ORB *Implementation Repository*, for instance from the command line (e.g. calling "putit serverName fullPathName activationMode" in an ORBIX ORB).

Knowing the available servers and the objects they contain, the ORB launches or just calls (if already running) the proper server depending on the demanded object service. Some different activation modes or mechanisms are proposed in the CORBA specification for this purpose. Those allow the ORB to control the number of servers that are running and avoid overloading unnecessary the machine it is running on. Nevertheless, the client must view all objects on the ORB as being up and running all the time, waiting on the client to invoke their operations. The activation mode is indicated when the server is registered in the *Implementation Repository*. The different available modes are explained in [Bak97]:

- Primary activation modes:

  **Shared,** all of the objects with the same server name (created in the same server) on a given host are managed by the same process on that host. It is the default mode.

  **Unshared,** only one object at a time can be active in one server. This is of use when the same server provides several object implementations.

  **Per-method-call,** each invocation will result in the creation of an individual process which will be destroyed at the end of the operation.

  **Persistent server,** the server is launched manually prior to any invocation on its objects. Afterwards it is treated as in *Shared* mode.

- Variations for the *Shared* and *Unshared* modes:

  **Per-client,** activations of the same server by different end-users will cause a different process to be created for each such end-user. Different processes owned by the same end-user will share the same server process.

  **Per-client-process,** activations of the same server by different client processes will cause a different process to be created for each such client process. Different processes, even if owned by the same end-user, will use different server processes.

  **Multiple-client,** activation of the same server by different end-users will share the same process. This is the default mode.

```
class CountServer {
    static public void main(String[] args) {
        try {
            CORBA.ORB orb = CORBA.ORB.init();                  // Get a reference to the ORB
            CORBA.BOA boa = orb.BOA_init();                    // Get a reference to the BOA
            CountImpl count = new CountImpl("My Count");       // Create the Count object and give it a name
            boa.obj_is_ready(count);                           // Export the ORB newly created object
            boa.impl_is_ready();                               // Ready to service requests (loop "forever")
            }
        catch(CORBA.SystemException e) {
            System.err.println(e);
            }
        }
    }
```

Figure 2.7: Example of server code [OH98]

In figure 2.7 you can see an example of server code. It is quite simple and easy to understand. In order to register an object implementation it follows these steps:

1. Get a reference for the (default) local ORB. If more than one ORB is running on the same machine, any of them could be specified.
2. Get a reference for the (default) BOA. As for the ORB, any one could be specified, if there would be more than one.
3. Create the object (as any other would be created) or objects the server contains. There is not any restriction about a server containing more than one object implementation.
4. Pass the interface and implementation name to the BOA, and enter the object reference in a *Naming Service*. In this case, the IDL compiler automatically generated the code necessary to do all that in the constructor of the skeleton.
5. Inform the ORB the object or objects are ready to be used.
6. Inform the ORB the server finished creating objects and loops forever (or just for a while) waiting for requests of the created objects. If the server falls down, the ORB should automatically start it up again.

0. Moreover, the server should catch any exception possibly thrown by its connection to the ORB.

### 2.6.3 The Client

Once the server and its objects are registered in the ORB, only the client code remains to enjoy it. A *Client* is nothing else than a piece of code that uses an object reference to request CORBA services. An object reference is a token that may be invoked or passed as a parameter to an invocation on a different object. In order to get an object reference, the *Client* connects to the ORB and does one of these:

- Uses a naming service (see *Object Naming Service* in 2.5.1).
- Uses a trade service (see *Object Trade Service* in 2.5.1).
- Uses an object factory (see *Life Cycle Service* in 2.5.1).
- Gets a persistent reference from any storage system (e.g. a file where it was recorded as a string).

The *Client* does not need anything else than that reference to use an object. If it has the object-type-specific stubs, it can access them as library routines in its program, calling them in the normal way in its programming language. If not, the *Client* can use the DII to request services. In any case, the ORB hides communication complexity and all its problems, allowing the *Client* to treat the reference as a very own object.

```
class CountClient {
    static public void main(String[] args) {
        try {
            CORBA.ORB orb = CORBA.ORB.init();                            // Initialize the ORB
            Counter.Count counter = Counter.Count_var.bind("My Count");  // Bind to the Count object
            ...
            counter.sum((int)0);                                         // Use the object as any other
            ...
            counter.increment();
            ...
            }
        catch(CORBA.SystemException e) {
            System.err.println(e);
            }
        }
    }
```

Figure 2.8: Example of client code [OH98]

In figure 2.8 you can see an example of client code (note the method calls look exactly like any other Java method call). As well as the server code, it is quite simple and understandable:

1. Get a reference to the ORB. It could be specified which one, if more than one are running on the local machine.
2. Locate the desired object (or objects).
3. Use it as any other object in the same address space.

0. Moreover, the client should catch any exception potentially thrown by the CORBA objects.

As you can see in figure 2.9, a dynamic invocation would not be so difficult neither. It keeps looking like Java code, but is really slower. However, the DII is much more flexible than the specific stubs and allows different kinds of invocations as explained in [Bak97]:

**Blocking,** the client is blocked until the call has been transmitted to the target object, the target object's code has been run, and the reply has arrived back at the client. It is the normal semantics of function calls.

```
...
CORBA.InterfaceDef CountInterface = counter._get_interface();         // Get the Counter interface
CORBA._InterfaceDef.FullInterfaceDescription intDesc = CountInterface.describe_interface();
                                                                      // Get the description of the interface
if (intDesc.operations[0].name.compareTo("increment")==0) {           // Check the interface description
    CORBA.Request request = counter._request("increment");            // Get the desired king of request
    request.result().value().from_long(0);                            // Fill parameters and return values
    request.invoke();                                                 // Invoke the request
    }
...
```

Figure 2.9: Example of DII code [OH98]

**Non-blocking,** the caller is allowed to run in parallel with the request and wait for the reply later. This is just possible while using DII.

**Store-and-forward,** the request is stored in a persistent store before being sent to the target object. This is really useful in implementing batch applications.

**Publish-and-subscribe,** a message is sent on a specific topic, and any object interested in that topic can receive it.

## 2.7 Let's stick to realities

It does not matter what the CORBA standard promises but what its implementations actually offer, and by now, they are pretty separated. Nowadays, the CORBA implementations seem to be quite green. The CORBA standard is just a newborn still growing up. Therefore, its particular implementations are yet more younger, testing different ways to accomplish the standard, and looking for their market place. This entails difficulties in the usage, and uncertainty about its future.

Here is a list of some products implementing part of the CORBA standard (ORBs and DBs):

- Object Request Brokers (table 2.1 shortens the features offered and promised by them in 1997, and almost all are widely commented and compared in [Sie96]).

  **Orbix** is a product from IONA Technologies (used as the example ORB throughout [Bak97]). It offers an IDL compiler, which generates the stubs, skeletons, a BOA specific implementation class to be inherited by any object, and headers for the object implementation C++ classes; some C++ defines to be used instead of the specific BOA implementation class if it cannot be inherited; a command to register the servers in the ORB; and the ORB daemon. All the

| Features | ObjectBroker | SOMobjects | NEO | ORB plus | Orbix | DAIS |
|---|---|---|---|---|---|---|
| *Protocols* | | | | | | |
| IIOP | $\surd$ | $\surd$ | $\surd$ | $\surd$ | $\surd$ | $\surd$ |
| IR | $\surd$ | $\surd$ | $\surd$ | '97 | $\surd$ | $\surd$ |
| Static calls | $\surd$ | $\surd$ | $\surd$ | $\surd$ | $\surd$ | $\surd$ |
| Dynamic calls | $\surd$ | $\surd$ | $\surd$ | '97 | $\surd$ | $\surd$ |
| *Language Bindings* | | | | | | |
| C | $\surd$ | $\surd$ | $\surd$ | '97 | '97 | $\surd$ |
| C++ | $\surd$ | $\surd$ | $\surd$ | $\surd$ | $\surd$ | $\surd$ |
| Java | '97 | '97 | $\surd$ | '97 | $\surd$ | $\surd$ |
| Smalltalk | '97 | $\surd$ | $\emptyset$ | $\surd$ | $\surd$ | $\emptyset$ |
| Cobol | $\emptyset$ | $\surd$ | $\emptyset$ | $\emptyset$ | $\surd$ | '97 |
| Ada | $\emptyset$ | $\emptyset$ | $\emptyset$ | $\emptyset$ | $\surd$ | $\emptyset$ |
| *CORBA services* | | | | | | |
| Naming | $\surd$ | $\surd$ | $\surd$ | $\surd$ | $\surd$ | '97 |
| Events | '97 | '97 | $\surd$ | $\surd$ | $\surd$ | $\surd$ |
| Life Cycle | '98 | '97 | $\surd$ | $\surd$ | '97 | '98 |
| Trader | '97 | '97 | $\emptyset$ | '97 | '97 | $\surd$ |
| Transactions | '97 | '97 | '97 | '98 | $\surd$ | '97 |
| Concurrency | '98 | '97 | '97 | '98 | '97 | $\emptyset$ |
| Security | '97 | $\surd$ | '97 | '97 | '97 | $\surd$ |
| Persistence | '98 | '97 | $\surd$ | '98 | $\surd$ | '97 |
| Externalization | '98 | $\surd$ | $\emptyset$ | '98 | '97 | $\emptyset$ |
| Query | '98 | '98 | $\surd$ | $\emptyset$ | '97 | '98 |
| Collections | '98 | '98 | $\emptyset$ | $\emptyset$ | '97 | $\emptyset$ |
| Relationships | '98 | '97 | $\surd$ | $\emptyset$ | '97 | $\emptyset$ |
| Time | '98 | '98 | $\emptyset$ | $\emptyset$ | '97 | '98 |
| Licensing | '98 | '98 | $\emptyset$ | $\emptyset$ | '97 | '98 |
| Properties | '98 | '98 | $\surd$ | $\emptyset$ | '97 | '98 |

Table 2.1: Commercial ORBs scorecard (Source: Standish Group, February, 1997) [OHE97]

classes generated by the IDL compiler must be compiled and appropriately linked to generate the client and the server. Then, by means of the provided command, the server has to be registered in the ORB. Besides, some basic services are provided (i.e. *Event Service*, *Object Transaction Service*, *Initialization Service*, *Life Cycle Service*), and even some in two ways: the CORBA standard form and the Orbix proprietary form (i.e. *Naming Service*, *Externalization Service*). Orbix is available on more than 20 operating systems.

**ObjectBroker** is Digital Equipment Corporation's implementation of the CORBA specifications. It is the oldest ORB, nowadays available on 21 different platforms (e.g. UNIX, OpenVMS, Windows, etc.). It offers different mechanisms to wrap legacy applications (some that are not CORBA compliant) as well as extensions to the IDL that aid in describing, structuring, and generating distributed applications. This ORB gives you two additional languages to organize and describe the object implementations (i.e. *Method Mapping Language*, and *Implementation*

*Mapping Language*). A compiler generates the stubs, and skeletons to match the IDL, MML, and IML specified. It also provides Kerberos authentication if desired.

**SOMobjects** is the IBM product, a library based ORB. It was initially conceived as a set of tools to build and use DLLs by means of object oriented technology. The appearance of CORBA led it to the distributed version (DSOM). It keeps its initial terminology, and, of course, the most of its non-standard mechanisms. A compiler is used to, starting from the IDL source file, automatically generate the implementation template file (within which the class implementation will be defined) for the desired programming language. That generated file contains stub procedures for each method of the server class. The same compiler generates the needed code in the client programming language. DSOM is available for IBM platforms and Microsoft Windows.

**DAIS** is another ORB developed in ICL Object Software Laboratories and launched in October 1993. It implements the *Trader Service*, *Life Cycle Service*, *Alert Service*, *Naming/Integrated Directory Service*, *Concurrency Service*, *Object Transaction Service*, and *Security Service* (note some of them are not even mentioned in the CORBA specification). It is available in quite a lot of platforms: IBM, Sun, Digital, ICL, etc.

**NEO** is a SunSoft product family to develop and deploy networked object applications based on the CORBA standard (note that Sun cofounded the OMG). It extends the standard by offering a DDL to describe the persistent state of the objects, and an Object Server Language (OSL) to describe the server's behaviour. All that is compiled and gives rise to a default, expected to be modified server implementation (just throwing *CORBA::NO_IMPLEMENT* exceptions), as well as the corresponding stubs. Since Java is Sun Technology too, it is closely related to NEO, and is used, for example, to ship the client side stubs across the internet, facilitating their distribution.

**ORB plus** is the Hewlett-Packard CORBA implementation. It tries to be a lightweight ORB, and, for example offers a *Simplified Object Adapter*. It supports *Events*, *Naming*, and *Life Cycle* object services. It provides an IDL compiler which generates interface classes and types, server base classes, and server skeleton types as well as the stub and skeleton code. All that will be compiled and liked with the corresponding server and client code.

- Databases (just a couple of examples):

**ObjectStore** is one of the best ODBMSs in the market. It offers an ODA to make the objects in the database accessible remotely by means of an Orbix ORB. The objects' code does not need to be changed. Just the compiling and liking processes change.

**ORACLE** , one of the biggest enterprises offering RDBMSs, joined forces in 1997 with Visigenic (another ORB implementor whose ORB is called "Visibroker") to incorporate object technology to its products. The last one, shipped in 1999, called ORACLE 8i ("i" stands for internet) claims for implementing native CORBA protocols. Actually, they implement the IIOP, but are not able to say how to use it, and how easy it could be.

Lots of tools offering partial CORBA implementations are in the market, all of them claiming to be CORBA compliant. However, all provide proprietary solutions together with or instead of those standardized in the CORBA specification (it is a matter of distinguishing the products to increase sales). Moreover, CORBA specifications use to be the mixing of those existing solutions. Therefore, none of the products in the market, at specifications shipping time, fully fulfill them, but at most a part. Once the standard is fixed, all the products must migrate to it, and this is neither simple nor fast. It can take years. This means an object or piece of code running on a given ORB will likely not work on a different one. It makes the development of a CORBA distributed system harder than it seems. However, on the other hand, that diversity gives you the freedom to choose what fits your needs the best.

The CORBA standard is really young (the CORBA 3.0 is just about to come). Thus, you cannot expect its implementations being mature. CORBA needs time, the support of the software enterprises, and the patience of the potential customers.

# Chapter 3

# CORBA in heterogeneous DBMSs

## 3.1 HEROS

HEROS stands for HEteRogeneous Object System. It is a tightly coupled object oriented heterogeneous DBMS developed in the Pontificia Universidade Católica do Rio de Janeiro (Brazil). Its implementation is described in [AULM98] as providing location and data models transparency, minimum interferences on the processing of its CDBs, multiple data abstractions to better represent the semantics of its CDBs, consistency maintenance of replicated data when these data are mapped into the global schema, and physical data independence.

The prototype described in the paper uses OO standards to integrate and coordinate information resources. It was developed on a UNIX environment, using CORBA as a middleware to facilitate the interoperation of the different components. The programming language used was C++, and the component databases were Oracle and Postgres (located in different machines). It supports operations at the federation level (using a global transaction model) as well as at the CDB level. A four level schema architecture with an OO model as Canonical Data Model (CDM) is proposed for the integration:

**Local schema,** defined using the data model of the corresponding CDB.

**Export schema,** which is already in the CDM, and represents what is shared by a given CDB with the federation.

**Global schema,** also in the CDM, shows the integration of the different component schemas.

**External schema,** that is just an application view of the *Global schema*.

The CDBs of study differ on DBMS, data model, computational environment and physical location. The integration mechanisms have to deal with those heterogeneities. CORBA, by means of the object architecture defined, takes care of the differences in location, hardware and environment; and HEROS offers mechanisms to handle synonyms, homonyms, object replication, measure units and generalization differences, once they have been detected by the federated database administrator. The system performs the translation between data models, but leaves to the administrator the detection of possible semantic heterogeneities.

Three main CORBA interfaces are defined:

**Federation** interacts with the applications.

**Component** interacts with the CDBs.

**Factory** creates and lists existing objects (needed due to the Life Cycle CORBA service).

A hierarchy of classes hanging from *Component* is used to represent meta-information associated to each CDB. The *Component* class is specified attending to the environment of the components (i.e. OO or relational), that are specified again in concrete systems (i.e. Oracle, Postgres, HEROS, ...). When a database joins the federation, an instance of *Component* class is generated in the corresponding subclass, depending on its environment and particular software. At this time, the translation rules are defined and attached to the given environment instance. It means the mappings depend just on the CDB being relational or OO, and not at all on the concrete software of use.

About the transaction model, HEROS allows updates and uses the compensable-compensating transactions presented in the Sagas model. An HEROS transaction is an open nested transaction. It gives visibility of partial results, which avoids other transactions lock of data access. The transaction is a sequence of compensatable transactions, one vital (or pivot) transaction, and the compensating transaction list corresponding to the compensatable transactions. A vital transaction is a traditional, non-compensatable transaction whose commit implies the commit of the whole global transaction. The compensating transactions semantically undo what was done by the compensatable transactions, if a rollback is needed. A flat transaction is just a degenerated nested transaction which only contains the vital transaction.

The class *Federation* is responsible for the global management of the transaction, while the subclasses of *Component* are responsible for the corresponding local management of transactions. The CDBs do not distinguish between global and local transaction, and are not responsible for the resubmission of the transaction if it is cancelled. However, it is assumed the CDBs use the strict 2PC protocol for the concurrency control to guarantee serializable and recoverable schedules on all executed transactions.

The system does not provide data materialization. Thus, the queries as well as the updates go straight to the CDBs. The *Federation* class uses the mapping data stored in the global schema to generate the operation tree, where each node corresponds to an operation over a CDB in the export schema, and the composition operation to be applied to the responses to build the federated result. Each operation is passed to the corresponding *Component* instance to perform it.

### 3.1.1 CORBA specific decisions made

- The data granularity chosen was the CDB. It means a CORBA object containing data resources corresponds to a CDB.
- In order to encapsulate the different CDBs, the interface methods are mapped to one of many implementations. The same interface is used by all the CDBs.
- Since the objects are well known (i.e. *Federation*, *Component*, etc.), the invocation is always static
- Servers were defined with *Persistent* activation mode, because they need to be registered so in the concrete ORB used in order to be activated later by the applications.
- The objects are created by means of a factory (i.e. *Factory* class).
- Every object is named (using the *Naming Service*).
- The activation policy is *Unshared* for all the objects (to allow parallelist) but the factory, whose policy is *Shared*.

## 3.2 MIND

MIND stands for METU INteroperable DBMS and is a system developed in the Middle East Technical University (METU) in Turkey. It is described in [DDK+96] as a multidatabase system based on OMG's distributed object management architecture.

In this project, the CDBs are encapsulated in generic database CORBA objects. One generic IDL interface is defined and multiple implementations are provided (a different one for each DBMS). A common data model (based on IDL) and a single global query language (based on SQL) make possible a unified access to the CDBs, allowing users to perform queries and updates on a global schema. It also provides serializable execution of global nested transactions without violating the autonomy of local DBs. Oracle, Sybase, Adabas and Mood are the DBMSs used in the prototype.

CORBA is used to handle the heterogeneity at the platform level and provide location and implementation transparency. The global queries are decomposed into subqueries which are sent to the ORB, that transfers them to the corresponding database server on the network. The subqueries are executed by using the Call Level Interface (CLI) routines of the local DBMSs, and the results are returned to the client, again through the ORB, as a single response. The clients are not aware of where the CDBs are or how they do their tasks. Moreover, new DBMSs can be added to the system without affecting the existing.

CORBA does not help to solve semantic heterogeneities, therefore, tools are given to the DBA to deal with domain conflict (i.e. differences in extensions: identical, intersecting, inclusion or disjoint), and structural conflicts (i.e. differences in intensions: homonyms, synonyms, attribute domain, scale, constraints, operations, etc.). A four-level schema architecture is proposed to help the integration of the different database schemas:

**Local schema,** the schemas of the local CDBs.

**Export schema,** translation of the *Local schema* to the CDM.

**Derived (Federated) schema,** combination of the *Export schemas* into an integrated one. Includes information on data distribution generated when integrating. It is built by the DBA as a view over the *Export schemas*.

**External schema,** corresponding to the *Derived schema* plus extra information or constraints needed for a user or application. It could be just a subset of the *Derived schema*.

The basic components of the system, depicted in figure 3.1, are:

**Schema Information Manager,** keeps the schemas information.

**Ticket Server,** generates globally unique, monotonically increasing numbers to stamp the subtransactions. It is used to guarantee serializability of global transactions.

**Factory,** is the omnipresent, named object that allows the creation of others (i.e. *Global Query Manager*, *Global Query Processor*, etc.).

**Global Query Manager,** parses, decomposes, and optimizes the global queries according to the information contained in the *Schema Information Manager*. It also provides global transaction management to ensure serializability (using the *Ticket Server* to enrich the subtransactions with ticket operations). Exists one *Global Query Manager* per user interacting with the system and its location is dynamically determined by the ORB (usually the local host). It can be used to perform more than one query.

**Global Query Processor,** is responsible from processing partial results returned by the *Local Database Agents*.

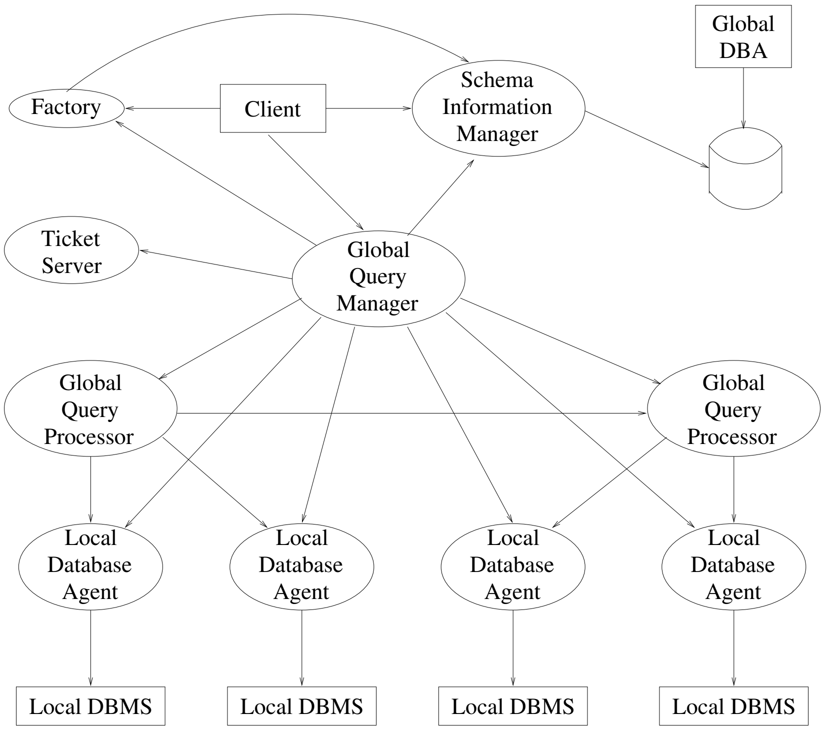


Figure 3.1: MIND global view [DDK+96]

**Local Database Agent,** maintains export schemas provided by the local DBMSs represented in the CDM, and translates the queries from the global query language to the local one. It provides an interface to the CDB.

A query submition implies the creation of a new *Global Query Manager*, which obtains schema information from the *Schema Information Manager* in order to decompose the query. Afterwards, the manager creates the needed *Local Database Agents*, and sends each subquery to the corresponding agent using the *Non-blocking* call mode (this allows the different agents to work in parallel). The *Global Query Manager* keeps a list of expected partial results. Whenever two results that need to be processed together are ready, the *Global Query Manager* creates a *Global Query Processor* to process them (allowing parallelism in building the result, as well). The final result is sent back to the client.

The last version of MIND implements a technique for global concurrency control of nested transactions called *Nested Tickets Method for Nested Transactions.* It makes consistent the execution order of sibling subtransactions at all sites. The main idea of this technique is to give tickets to each global transaction as well as its childs. Then, each subtransaction is forced into conflict with its siblings through its parent's ticket at all related sites. The subtransaction is aborted (thanks to the 2PC protocol being supported by the CDBs) if its ticket value is smaller than that of its sibling transaction previously executed at the same site. Note that it is supposed to exist a grandparent never-ending transaction whose ticket value is zero. Therefore, all transactions are interrelated. It is a semi-lattice where a relation between two transactions can always be stated by means of a common ancestor.

### 3.2.1 CORBA specific decisions made

- As in the HEROS project, the data granularity chosen was the CDB. The whole database is encapsulated into a CORBA object.
- In order to encapsulate the different CDBs, the interface methods are mapped to one of many implementations. The same interface is used by all the CDBs.
- The invocation is static again, because the CORBA objects are fixed and well known too (i.e. *Factory*, *Global Query Manager*, etc.). However, since the *Non-blocking* call mode is supported only by DII, it is used in some concrete cases (like the *SendQuery* method of *Local Database Agent*).
- The objects are created by means of a factory (i.e. *Factory* class).
- *Factory*, *Ticket Server*, and *Schema Information Manager* are registered in the *Naming Service*. They serve the whole system continuously. Thus they are not created on demand but during the start up phase of the system.
- *Shared* activation policy is used for *Factory*, *Ticket Server*, and *Schema Information Manager*, mainly because of their short activation time. The other objects are activated in *Unshared* mode to provide parallelism and keep separated contexts for each different transaction.

## 3.3 BLOOM

As was explained in section 1.2, the aim of this paper, besides introducing CORBA, is studying how a particular distributed information system architecture (say BLOOM execution architecture) can be implemented using distributed object technology. In view of previous sections, CORBA and BLOOM seem to fit into each other.

Each one of the pieces depicted in figure 1.1 (i.e. *Security Controller*, *Query Transformer*, *Query Decomposer*, *Level Security Translator*, *Transaction Manager*, *Subresult Translator*, *Result Consolidator*, *Result Transformer*, *Directory* as well as local DBMSs) can be wrapped into a CORBA object. Once the code is wrapped, the CORBA architecture takes care of the system heterogeneities. The different BLOOM modules will not need to be aware of being running on a single machine or across a wide area network of absolutely different machines running code programmed in different languages; they just need to worry about semantic heterogeneities and data schemas translation. Moreover, the CORBA services could also help in the specific implementation of some modules. Since *Concurrency Control Service*, *Transaction Service*, and *Security Service* (outlined in section 2.5) have already been well defined by the OMG, they could be used in the transaction and security managers.

Firstly, before implementing the execution architecture, the best distribution of the modules across the network should be studied. It is a trade-off between the load of the machines, the communication speed between them, and the amount of data to be transferred. Ideally, the distribution should be dynamic, taking benefit from the machines and communication channels load at any time. However, that would need lots of information about the network and the machines, and the willingness of the CDBs administrators to have the modules running on their machines. Being much more realistic, the distribution of the modules will be quite static and mainly depending on the administrators of the different CDBs, and the available machines, as well as the network architecture. It will be absolutely different for each particular instance, and will always be fixed more by the existing components than by the desired performance of the cooperative system.

CORBA will be really helpful specially at this point because of its location transparency and flexibility. Wrapping the different modules separatedly, will increase that flexibility on distribution and migration. CORBA follows a tiny, reusable objects policy, and it seems good to keep it in this case, because:

- Facilitates software reusability.
- By itself, does not cause unreasonable overhead, if the modules keep running on the same machine. The overhead appears more due to communications between machines than between different processes on the same machine.
- Helps to cause the acceptance of modules running on local machines by their administrators, since they are small.
- The modules can be easily spread over the network avoiding an unnecessary overload at any particular site.
- The modules can be redistributed in as needed basis, when the federation is modified (a member leaves or a new one joins it).
- Different instances of the same module could run at the same time if needed. For example, it seems good having just one *Query Decomposer* but several *Query Translator*s.

- Allows parallelism between different modules. The *Subresult Translators* could do their task running in parallel with the *Result Consolidators* joining the results already translated, for instance.

However, wrapping small pieces is not always desirable. In the case of the CDBs, the best choice seems to be wrapping the whole DB into a single CORBA object (*Thick Granularity*), and let the local DBMS do its tasks. Nevertheless, some other possibilities could be taken into account: wrapping each table or tuple (if relational) into a different object (*Fine Granularity*). In some cases *Fine Granularity* could be necessary, for example due to security restrictions, to avoid overload on a given CDB or to implement special search or order algorithms. Still in general, it seems better having everybody doing its own job, instead of trying to do what is already done. A full DBMS could be implemented just using CORBA services, and use the CDBs as simple repositories. But why should we build a new DBMS, if what is already done works well?

In order to make easy and smooth the integration, it seems good letting CORBA hide different implementations of the same interface. It means having one to many mappings between interface and implementations. It will be specially useful for the component DBMSs and the translators (i.e. *Result Translator*, *Query Translator*, and *Level Security Translator*). We could have a different translator implementation for every data model or even schema, but only one interface to call their methods. It means hiding the complexity of one module to another. For example, the *Query Decomposer* does not need to be aware of which model a given subquery is going to be translated to. The decomposer just need to know how to split a federated query into pieces and where to send each piece. Having different methods or parameters depending on the data model to be translated to is not needed at all. The same is applicable to the different CDBs, the best is having the same interface to all of them despite their specific implementation. Having a different interface depending on the DBMS that the interface should hide does not seem to make too much sense.

The next point is how to create the object instances. The first possibility is having the objects compiled and linked with the applications, which would then be able to solve their own necessities. Obviously, this would not be a good choice, it would throw away all the benefits we have been talking about CORBA. A better possibility is having a given number of stand-alone object instances always running, waiting for requests. The references to those objects could be obtained using the *Naming Service*, or having them persistently stored in files (as strings). This method does not seem enough flexible yet, because it implies a fixed number of objects, and a name distribution mechanism. The distribution mechanism could be the *Trade Service*, but it is hard to use (better to avoid using it while possible).

Another and really good possibility is having factories which produce new objects on demand. This avoids registering every object in the *Naming Service* and storing references into files. Besides, we could ask as many objects as we need (and where we need them). The problem is how those factories are created and who does it. Here we go back to the discarted possibilities: they must be already running and registered in the *Naming Service* or their references stored into files. It is not a problem now, because the number of factories is smaller than the number of objects. Moreover, having a fixed number of factories is not problematic, either, because they will not be a bottleneck in the execution process. You just need a factory running on the machine you want to be able to run objects on, register it in the *Naming Service*, and ask it for objects implementing the desired interface.

We can have an unique huge factory creating all the possible kinds of objects, or have a different specific tiny factory for every different kind. Probably, the best solution would be in between. Having a huge factory means recompiling and distributing it every time an object changes, and maybe, creating a bottle neck, if everybody asks objects to it. Having a specific factory for every kind of object means a lot of factories running and names to record or register. The reasonable solution would be having a factory for every interface or small set of related interfaces. With this, a site only needs to have factories to create the kinds of objects it wants to run. Besides, the same factory could create all possible different implementations of that interface. For instance, we would have a factory to create the *Query Translators*, which, given a parameter, would return the proper implementation to translate from the CDM to the desired native data model.

The activation policy is another choice to be done, see section 2.6.2. Remember CORBA offers four different possibilities: *Shared*, *Unshared*, *Persistent*, and *Per-method-call*.

- The *Shared* mode would be used in the objects that do not need parallelism: the factories, and maybe the *Directory* of the system.
- The *Unshared* mode allows parallelism for calls to objects running on the same machine. A different process is created for every user or process depending on the submode (i.e. *Per-client*, or *Per-client-process*). Using this mode avoids a small query waiting for a long one to be executed in a given site. It could be used in almost all the modules to provide parallelism.
- The *Persistent* mode means the server is always started up by hand, never automatically. Therefore, it is never halted after an inactivity period of time. It is useful for modules hard to start up, or those that record its state between different calls. It might be the case of the *Directory*.
- The *Per-method-call* is the opposite to the *Persistent* mode, an object is started up every time a method is called. It saves resources by avoiding unnecessary processes in memory. It could be used with objects not keeping state and having fast start-up.

The last important point to talk about is the invocation (static vs dynamic), see section 2.6.3. The static invocation using stubs is faster, that is clear. Moreover, there is not any problem using it, because all object interfaces are well-known. However, there are some cases where the dynamic invocation fits better. Remember there are three invocation modes only available using dynamic invocation: *Non-blocking*, *Store-and-Forward*, and *Publish-and-Subscribe*.

- The *Non-blocking* mode avoids the client waiting for the response of the target object. Its usage is essential to allow parallelism.
- *Store-and-Forward* stores the call to be executed later. Really useful for batch systems, maybe running at night.
- *Publish-and-Subscribe* allows to throw a call into a channel. Any server listening to that channel can serve it. It provides a high-level flexibility, because allows to build a "producers-consumers" architecture. When a client needs to request a service, instead of looking for a given instance of an object, it leaves its call into the channel, and whoever listening to it can serve the request.

The last one of those modes means a radical change in the implementation. However, the others could be provided as an optional feature of the objects, already offering static method invocation. They could be implemented together with the static calls and activated by a given parameter or call to the object. Their performance is not so bad if the *Request* object is already prepared to be used. It should be tested in particular cases.

## 3.4 Others

These had been just examples of multidatabase systems using CORBA to solve system heterogeneities. Of course, other projects exist, for example IRO-DB, or Jupiter. Curiously, none of these projects takes place in USA. TSIMMIS (in the Stanford University) and Information Manifold (in AT&T) talk about "wrappers" and "mediators". However, they do not say anything about using CORBA to glue everything.

# Chapter 4

# CORBA in the persistence of data

## 4.1 The different ways

The usage of CORBA in the implementation of the execution architecture of the cooperative system has been studied in the previous chapter. However, it can be useful in another important point: the persistence of data. Independenly of helping the interaction between the different modules of the execution architecture, CORBA offers tools to the implementation of wrappers for the DBMSs. It is really handy hiding the heterogeneities in the access to the different CDBs.

Since the CORBA system has been thought as a "mecano", there is not only one way to get persistence in a CORBA environment. Probably, none of these ways is perfect, all of them have pros and cons, some are firmly rooted in the market while others have a doubtful future. Moreover, they are not exclusive between them, different combination are possible too. For sure, other people could find other possibilities for using CORBA to get data persistence, CORBA is not a closed system at all, but those presented here are:

- "Classical" three-tier approach (plus CORBA).
- Object Database Adapter.
- Object loaders.
- Externalization Service.

- Persistent Object Service.
- Object Query service.

### 4.1.1 "Classical" three-tier approach

This first approach to get persistence is diametrically opposite to the others described in next pages. It proposes an architecture where the CORBA objects themselves are not persistent, but they handle the persistent data, hiding their complexity, and actual location to the clients.

Isolating the client from the complexities of the DBMSs reduces perceptibly its code. Moreover, the client does not have to be modified if the storage of the data changes. It also helps to control the access to the data, since it is always through a clearly defined interface, its specific implementations are not mixed with the rest of the code, and the access is centralized.

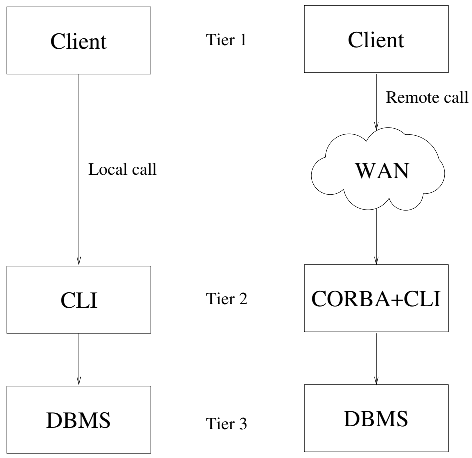


Figure 4.1: Three-tier architecture with and without CORBA

Actually, CORBA is not needed to implement this kind of architecture. CORBA is just the front-end to the data, any piece of software could be used as middleware. It is the typical three-tier architecture composed by "repository", "middleware" and "user interface". That middleware can be implemented using JDBC, ODBC, DCOM, or any other kind of CLI; and it is precisely used to hide the usage of those tools to the client application. It just sends the proper SQL statements to the DBMS, and passes the desired results to the client.

In spite of not being necessary, CORBA usage could make easy the communication and distribution of that middleware. At the same time, it wraps the access to the database and provides portability for the middleware code. An example of this usage of CORBA can be found in [OH98]. It studies how CORBA and Java can work together to obtain distributed applications. This book even gives code and some benchmarks to compare different ORBs and tier distributions.

Using CORBA only as a wrapper to hide the location of the code is wasting its potential. This approach neither makes good use of the object oriented technology in CORBA, nor its modularity, reusability, interface standardization, services, ... Obviously, CORBA hides the location of the code, but it can do lots of other things. This architecture should be mixed with some of the solutions proposed in next pages.

### 4.1.2 Just an Object Database Adapter

[OMG95] describes the structure and features of objects adapters. They are ORB dependent, and offer an interface to the object implementations. Since they are in between the ORB and the object implementation, they can be used to intercept the calls and perform the needed features. It is specifically contemplated in that specification the usage of an specialized OA (called *Object Database Adapter*) to get persistence. An ODA is suitable for the storage in ODBMS. It can use a connection to an object oriented database to provide access to the objects stored in it. The mention of this kind of things in the CORBA specification is a proof of the good relations existing between the OMG and the ODMG.

Moreover, the second appendix in [CBB+97] describes how ODBMS objects could participate as OMG objects, routing object invocations through identifiers provided by the ODBMS itself. Requests to the persistent objects, whether through the ORB or directly to the ODBMS, produce the same effect and are absolutely compatible. Those persistent objects look exactly like any other object accessible through the ORB, from the client point of view. Besides, to facilitate it, the ODMG Object Definition Language (ODL) is an extension of the IDL.

Registering all database objects in the ORB does not seem a good choice, mainly because it would produce a lot of unnecessary overhead. Therefore, the ODMG also describes the ODBMSs as having the capability to register subspaces of object identifiers with the ORB. That allows the ORB to handle requests to all of the objects in an ODBMS without the registration of each individual object. This seems a powerful tool for handling persistent objects.

A specific implementation of an ODA is presented in [ION97]. That implementation allows an Orbix ORB to store the objects in an ObjectStore database. It could also be seen in the opposite way as the ObjectStore objects being invoked by CORBA applications through the Orbix ORB. We could say that paper presents the complementarity between Orbix and ObjectStore.

When an invocation is made on a persistent object that is not currently loaded, Orbix will pass the invocation to the ODA, which will cause the object to be loaded from the ObjectStore database, and Orbix will then pass the invocation to that object. From the client point of view, the Orbix clients do not need to be aware of whether an object is both an Orbix object and an ObjectStore object at the same time, and the same applies for the ObjectStore clients. On the other hand, from the implementor point of view, a programmer just carries out the implementation steps of an Orbix object plus the implementation steps for ObjectStore:

1. Define the object interface using standard IDL.
2. Use the modified Orbix IDL compiler to compile the interface.
3. Declare the corresponding C++ class.
4. Process the class declaration with the ObjectStore schema generator.
5. Implement the C++ class.
6. Compile and link with the needed libraries (Orbix and ObjectStore), stubs, and skeletons.
7. Create the instances using the ObjectStore overloaded "new" operator.

This is probably the most simple, comfortable, graceful, and powerful way to get persistence for CORBA objects. However, by means of it, integration of legacy applications is not possible. Moreover, you are tied to some implementations, and have to use ODBMS. One of the probably most important benefits of CORBA is lost: its universality.

### 4.1.3 Object loaders

A specific Orbix mechanism to implement persistence is described in [Bak97]. It is suitable to access a RDBMS (or ODBMS without ODA), and consist of four steps:

1. Decide the OO to relational mappings.
2. Code the mappings.
3. Write or import a "loader" that will create a C++ object for each object invoked by clients.
4. Ensure that the RDBMS is updated at the end of a transaction and that objects are removed from memory.

It encourages (or discourages) you to implement the persistence by hand and almost from scratch. Some tools (i.e. Ontos, Persistence, RogueWave, etc.) are proposed to automate or semi-automate the process, but, even using them, it still seems long and hard. They could be seen as adhoc extensions of the ORB in order to get the desired object activation. Actually, implementing a loader is providing a specific OA for specific persistent objects.

When an operation invocation arrives at a process, Orbix searches for the target object in the process's object table. If the object is not found, the loader of the object will be informed about the object fault and provided with an opportunity to load the target object and resume the invocation transparently to the caller. Each object is associated with one of these loaders. If no loader is explicitly specified for an object, then it is associated with a default loader, implemented by Orbix. A loader is nothing else than a subclass of *CORBA::LoaderClass* overwriting the methods *load*, *save*, *record*, and *rename*.

**load** is called when the object is not found in memory and is responsible of the load the object.

**save** saves the object on process termination.

**record and rename** are used to choose the persistent identifiers for the instances.

This approach is quite at a low level, and absolutely dependent on the (Orbix) ORB. It is not part of the CORBA standard at all. Thus, it means a loss in portability. Moreover, it can always be avoided by using other similar solutions contemplated in the CORBA specification (e.g. *Externalization Service*). However, in some particular cases could be a really good solution as alternative to an ODA, if you are not using an ODBMS.

### 4.1.4 Persistence through externalization

As defined in [OMG94], externalizing an object is to record the object state in a stream of data. Objects which support the appropriate interfaces, and whose implementations adhere to the proper conventions can be externalized to a stream (in memory, on a disk file, across the network, etc.), and subsequently be internalized into a new object in the same or a different process. The externalized form of the object can exist for arbitrary amounts of time, be transported by means outside the ORB, and can be internalized in a different, disconnected ORB.

The externalization was conceived as an easily implemented service that could work with any kind of object. The stream concept is quite similar to that in C++, and has two basic operations: put data into, and get data out.

In using this CORBA service there are three different roles: client, stream and streamable object. The client invokes operations on the stream in order to read or write an object. On externalization, the stream records the object state; and on internalization, the stream invokes a factory to create a new, non initialized instance, and initializes it with the recorded state. Each streamable object has to implement its own way to be externalized and internalized using the *StreamIO* interface. Moreover, there is a Standard Stream Data Format (SSDF) which makes it possible to externalize an object on one system and know for sure that you can internalize it on any other system able to run that kind of object. The SSDF lets you exchange streams across dissimilar networks, operating system platforms, and storage implementations. The *StreamIO* is responsible for using the SSDF or not.

The usage of this service is driven by the six interfaces (distributed in two modules) written in appendix A.1:

**Stream** wraps the stream itself.

**StreamFactory** is the factory to create streams.

**FileStreamFactory** is a special kind of stream factory that allows to create streams on files.

**Streamable** is the interface implemented by the objects supporting their externalization and internalization.

**StreamableFactory** allows the streams to create new streamable objects where to internalize a state.

**StreamIO** provides operations to write and read all the IDL data types (possibly using the SSDF).

The steps to externalize an object are:

1. Call a stream factory to get the desired kind of stream.
2. Invoke the stream passing as a parameter the object reference to be externalized.
3. The stream tells the object to externalize itself to the stream.
4. The object writes its content to the stream (through the *StreamIO* interface).

Later on, may be in a different machine, the steps to internalize the previously externalized object are:

1. Invoke *Stream::internalize* on the desired stream.

2. The stream object looks inside the stream for a key that helps it to locate a factory that can create an object with an implementation that matches the object (or objects) in the stream.

3. The stream tells the streamable object to internalize itself.

4. The object reads its contents from the stream (through the *StreamIO* interface).

All this is part of the CORBA standard (concretely one of its services, as said in section 2.5). Therefore, it is always well-stated and the objects implementing it are interchangeable between different ORBs. Besides, part of its implementation could be provided by a vendor (i.e. *Stream*, *StreamFactory*, *FileStreamFactory*, and *StreamIO* implementations). Its main weakness is that it forces you to record the whole state of an object at a time, you are not allowed to externalize just a part of the state (it is all or nothing). This is because this service is not thought as to get persistence, but to copy or move object instances from one machine to another one that is not connected to the former. However, in spite of its simplicity, it is really handy. If all the objects were streamable, it could be used to ease the task of other services. For example, the *Persistent Object Service* would just need a translation mechanism from the SSDF to the desired storage mechanism in order to get persistence.

### 4.1.5 The Persistent Object Service

This service, as was said in section 2.5, supports the capability of making persistent all or part of the state of an object. It describes ways for the object to decide what state needs to be made persistent, and ways to store and retrieve that state. Its usage is not mandatory. Any object has the responsibility of managing its state, but can delegate part of the work to the POS. An object could only use some of the POS components or all of them. All this results in quite a freedom to the persistent objects' implementors.

The POS specification is a merging between two proposals, one coming from IBM and another one from SunSoft, submitted to the OMG. IBM proposal tried to integrate the RDBMSs into the OMA world, while the second one was directed towards establishing interfaces to the ODBMSs. As a result, the definitive POS specification is wide opened. It was conceived to make everybody happy, or at least, as much happy as possible. Any storage system can be plugged into the POS. Moreover, it ensures that client code remains unchanged as object datastores are changed, by presenting the interfaces as a contract between the POS and datastore vendors. Another important point to take into account about the conception of the POS is that, violating the OMG policy, it is not based upon well established market products.

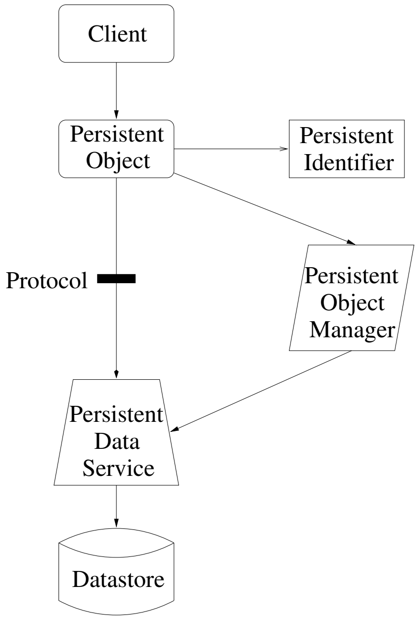


Figure 4.2: Components of the POS [OMG98]

The different components of the POS are depicted in figure 4.2 and described in [OMG94] and [OMG98] as:

**Client** is who asks for recording a persistent object. It can exist or not depending on the persistent object implementation offering the PO interface or not.

**Persistent Object (PO)** is an object whose state is recorded in a *Datastore*. The persistence is provided by the object itself, but by means of implementing the corresponding interface, it may let the client to control when the state is recorded. An object is persistent or not independently of whether it implements the PO interface or not. An object is persistent if it stores its data persistently by means of the POS.

**Persistent Identifier (PID)** describes the location on an object's persistent data in some *Datastore* and generates a string identifier for that data. An object must have a PID in order to store its data persistently. It is passed as parameter to the POM, and is used to identify one or more locations within a *Datastore*. Note it is different from the OID in that the PID identifies data location, while the OID identifies a CORBA object. A PID cannot be used as target for an invocation; it can only be used in the context of the POS.

**Persistent Object Manager (POM)** provides a uniform interface for the implementation of an object's persistence operations. An object has a single POM to which it routes its high-level persistence operations. Its role is simply to decide which PDS to pass the calls to. This allows the PO to use different PDSs without changing its implementation but the PID. This routing function of the POM serves to shield the client from having to know the details of how and where actual data storage takes place. Thus, the POM will use non-standard operations to find out which protocols and PDSs a given object can work with.

**Persistent Data Service (PDS)** provides a uniform interface for any combination of *Datastore* and *Protocol*, and coordinates the basic persistence operations for a single object. It translates from the object world to the specific *Datastore* world.

**Protocol** provides one of several ways to get data in and out of an object. It is the way in which the object interacts with the PDS. There is no an unified protocol, three different ones are defined in the services specification: the *Direct Attribute* (DA) protocol, the ODMG protocol, and the *Dynamic Data Object* (DDO) protocol. Moreover, any other environment or language specific protocol could be used, as well. For example, SSDF could be the fourth.

**Datastore** provides one of several ways (i.e. flat file, RDBMS, ODBMS, etc.) to store an object's data independently of the address containing the object. It will typically offer a CLI to access the data in it.

With all this, [Ses96] describes the "typical" persistence request (a store invocation) as the client setting up the PID and invoking the store on the PO. The PO passes on the store request to the POM. The POM looks at the PID and the object to determine which PDS can handle the request. The POM forwards the store request to the appropriate PDS. The PDS then takes over the store, interacting directly with the object and the *Datastore*.

Obviously, the POS just defines the IDL interfaces (listed in appendix A.2) and does not say anything about their implementations. Different implementations are possible and even encouraged. The existence of all those modules gives a great freedom degree to the implementors. Anyone of the modules could be implemented by a different one, and everything would work together without problems. Concretely, it means different storage systems could be used at the same time, and even interchanged, in spite of ranging from simple file systems to DBMSs (while supporting the same protocol).

Those interfaces propose two different ways to control the persistence of an object. The fist one (*connect* method) establishes a "permanent" connection between the volatile and persistent data of an object. Any change (while the connection is established) in the former is automatically reflected in the *Datastore*. After disconnection (*disconnect* method), the persistent data keep the values. About the second way, it allows a client to control when data is stored and restored (methods *store* and *restore*). There exists another method (*delete*) which deletes the object's persistent data from the *Datastore* location indicated by a PID.

The POS proposes a solution to wrap storage mechanisms (object oriented or not), so that they can be used as object technology. It means the system is opened to any kind of data storage (including RDBMSs). However, a problem arises when trying to use an ODBMS: it losses simplicity and efficiency. Since the DBMS is not aware of the actual data structure of the objects it is recording, a direct mapping between memory and the storage system cannot be generated. The advantages of that mapping are probably the main reasons to use an ODBMS instead of an RDBMS extended for objects. Therefore, the ODBMS vendors are not completely happy with this service (they promote the usage of an ODA in order to get persistence).

[KPT96] is a paper summarizing the weaknesses of the POS. The first one is the underspecified semantics of operations (i.e. *connect*, *disconnect*, etc.). The specification does not specify what concretely had to do some of the operations, it is left to the implementor's discretion. Something similar happens talking about the POM, because proprietary solutions are expected to be introduced. Besides, the reusability of other CORBA services is just briefly mentioned. However, these problems were probably generated on purpose. The OMG left some concretions for future revisions, which will be done after the shipment of some commercial products implementing the standard (it was previously mentioned that, exceptionally, this service is not based upon market products).

Other commented problem in the standard is the lack of a "Compound Persistent Object Service". It would be responsible for the storage of the inter-object references. This will be really hard since the PID and the object are loosely coupled.

### 4.1.6 The Object Query Service

Probably, the most suitable of the CORBA services for being used in BLOOM is the OQS, previously introduced in section 2.5. In spite of the fact that it does not provide object persistence by itself, it is really useful to access information in storage systems. As described in [OMG94] and [OMG98], the OQS provides query operations on collections of objects. In this context, "query" stands for selection as well as insertion, updating, and deletion. Those documents even mention that the OQS could coordinate multiple nested query evaluators in a federated service architecture.

Despite that another CORBA service is specifically devoted to collections, they are briefly described as part of the query service. It offers interfaces for creating and manipulating collections of objects, possibly obtained as a result of a query. Those collections are defined as objects, with methods for adding and removing members. Any system managing extensions of objects (as complex as needed) can be wrapped into a collection object. Iterators ("cursors") are explicitly specified, as well, and appear always bound to a given collection. They are used to manipulate collections, traversing them and retrieving their objects.

The OQS is a front-end conceived to unify CORBA objects, RDBMSs, and ODBMSs into a single query target. As everything in the CORBA world, it is done by specifying an IDL interface, or set of interfaces. Anything implementing that interface can participate in the query system as a first class object. It is expected the database vendors will implement this feature as part of their database systems. The interfaces offered by the OQS, and listed in appendix A.3 are:

**QueryEvaluator** Evaluates query predicates and executes query operations using the specified query language.

**Query** is the interface encapsulating a query by which it can be previously designed, saved, precompiled, and treated as any other object. It supports four operations: *prepare*, *execute*, *get_status*, and *get_result*.

**QueryManager** is a specialization of *QueryEvaluator* which creates *Query* instances (it is a query factory at the same time).

**CollectionFactory** is just a factory creating *Collection* instances.

**Collection** allows you to manipulate grouped objects.

**QueriableCollection** is a specialization of *Collection* and *QueryEvaluator*, both at the same time. Thus, it allows to evaluate queries on a given set of objects.

**Iterator** is a movable pointer into a *Collection*.

The query process begins when a client sends a query to a *Query Evaluator*, which could just be a collection supporting the corresponding interface, or a complex DBMS. The *Query Evaluator* passes the query predicate to the collection, which then evaluates the predicate and performs query operations on an appropriate member object, receives any result, combines such results with all other participating object results, and returns this to the caller (as a single object or a collection). As said above, there is no problem in nesting the process, that is the members of a collection could be collections, and the members of a *QueriableCollection* could be *QueriableCollection*s.

The steps to execute a query and obtain the result are:

1. Create a *Query* object by invoking *create* on a *QueryManager*.
2. Prepare (precompile) the query.
3. Execute the query as many time as you like.
4. Get the result.

5. Create an *Iterator* on the returned collection (if a collection was returned).

6. Read the first element in the collection.

7. Go to the next object in the collection and read it as well.

The first three steps could be simplified by just submitting the query by invoking *evaluate* on the *QueryEvaluator*. This will be much faster if the query is going to be executed only once. However, since in this case it is not precompiled nor stored, its repeated execution would be pretty slower.

In describing the OQS, [OMG98] uses the term "object" in the general sense to include data. It means the contents of a collection do not have to be CORBA objects but just a set of data. This solves the problem found in other services, here the objects implementation does not need to be modified nor extended to get persistence. The data is stored in a storage system and retrieved using a query language. A completely new view point is offered: less work, less service. You are not recording first class CORBA objects but just their "plain" data. It seems thought as a simplification to win people for the CORBA cause.

In the same way, it is expected the *QueryEvaluator*s to support at least SQL-92 Query or OQL-93 Basic, to be useful for relational as well as for object oriented users. The query languages are evolving fast, and this will change as soon as one supporting relations and objects appear in the market.

## 4.2 Meeting BLOOM

CORBA is conceived like a "mecano" with complementary pieces. None of the solutions proposed above solves all the problems in the BLOOM architecture, rather all them working together will improve the BLOOM system. In spite of that, some seem more useful than others, depending on where you want to use them.

At first, the POS seems really attractive to wrap the CDBs. It is a service conceived to get persistence which covers all the possibilities. However, some problems arise when looking in depth: it is too complex and has not been implemented yet. If implementations of some of its interfaces (i.e. PDS, POM, and PID) are not in the market, the adhoc implementation of the whole POS seems unfeasible. The problem is that this does not seem going to happen because it is the most questioned service. There exists a confrontation of interests between RDBMSs and ODBMSs vendors. The former do not completely support CORBA, while the others are more interested on the ODA than on the POS. The OMG is close to issuing a RFP for POS 2, having the coexistence with POS 1.0 interfaces as a requirement. Maybe that will be the solution to the problem.

The easiest and best way to access the CDB would be by means of an ODA. Nevertheless, one of the strongest constraints in the BLOOM architecture is to keep the autonomy of the CDBs. That autonomy is no problem if it is the case of an ODBMS (almost for sure it is going to have an ODA in the market), but there are no ODAs for RDBMSs, and the CDBs should likely have to transfer part of their autonomy to the federation in order to build that relational ODA (let apart its implementation costs).

The Externalization Service or specific object loaders seem to be a quite simple way to get persistence. They are really easy to implement and no "external" aid is needed. If every object implements its own storage method, everything goes smoothly. This is a magnitude problem: who is going to implement the hundreds or thousands of different methods to record and recover the existing objects if they are not implemented yet? Furthermore, the BLOOM CDBs have to maintain their autonomy. Therefore, their objects cannot be modified in order to get persistence (we run up against the same wall again).

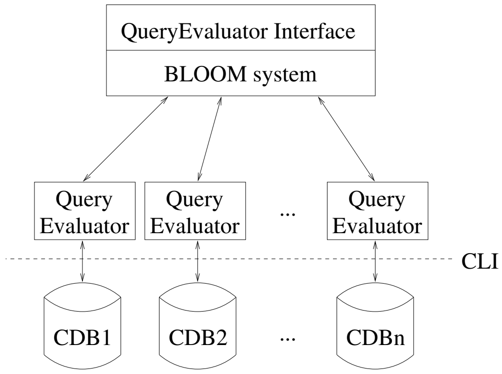


Figure 4.3: BLOOM-OQS architecture

Probably, the problems outcrop because we are trying to store what is already stored. The data in the CDBs are already persistent. Rather, just a query system is needed. A simple three-tier architecture seems to fit well here, because it will allow the CDBs to keep their autonomy. Their machines do not need to be modified at all if they offer a way to query the databases (this does not seem an unfounded assumption, because almost all DBMSs offer a CLI). Besides, the typical three-tier architecture could be improved to benefit from CORBA. If every CDB is wrapped into a *QueryEvaluator*, CORBA will hide the system heterogeneities (just the semantic ones left). See figure 4.3.

The absolute autonomy and security of the CDB can be easily guaranteed just by means of the CLI operations each one offers to the federation, and the specific implementation of the *QueryEvaluator*s. The heterogeneities in the CDBs are smoothed out at the same level. Furthermore, CORBA will hide the actual location of the puzzle pieces, which will facilitate the evolution of the system. Moreover, all the BLOOM system could be wrapped into a *QueryEvaluator* as well, to obtain a nestable architecture. Offering a *QueryEvaluator* interface to the federation will not be an excessive effort, and will let plug the federation into another federation.

The OQS does not seem to be necessary in all this. Its interfaces are so simple, we could just forget those and easily redefine them to our own liking. However, why should we reinvent what is already invented and standardized. We do not need to waste time thinking what was already clearly stated. Moreover, using a standard would make easy the joining of new CDBs to the federation, not to mention in the near future, hopefully, the DBMSs could offer their own implementation of the OQS interfaces, complementing the current CLIs, to be integrated in a CORBA environment.

Obviously, OQS is not the perfect solution to be always used everywhere. CORBA offers lots of possibilities and all them should be carefully taken into account to implement a cooperative system. As it was previously said, all the proposals in this chapter could be combined to fit the needs of the system, mainly depending on the kind of CDBs.

Leaving the data in the CDBs aside, storing the data of the different modules in the BLOOM execution architecture is another completely different story. We do not have to keep any autonomy at all, and they must be implemented from scratch. It means we can choose the CORBA service fitting the best into our needs. The *Directory* is probably the most special and interesting of the modules attending to their persistence. It is a database about the databases, keeps the information needed to integrate them (i.e. schemas at the different levels, mappings between those schemas, security information, transactional data, etc.). Once the POS is discarded, the externalization service seems a good, wide accepted alternative. However, since there could be lots of different objects, and it is more a database than a distributed system, a database storage system would fit much better. Doubtless, if we can choose the database to use, the best choice is an ODA. Using a database offering an ODA implementation for the given ORB (or ORBs) will provide transparent persistence for all the data in the CORBA objects, or transparent CORBA access to any data in the database (depending on the point of view). This way we get the distribution of a CORBA system plus the persistence storage of a database without any extra effort.

On the contrary, the other modules in the BLOOM execution architecture (those that are not the *Directory*) are more a part of a distributed system than a database. Therefore, the best is the simplest solution: the *Externalization Service*. It will allow the modules to store any possibly persistent data they handle without any problem. Moreover, together with the *Life Cycle Service*, it can be used to copy or move a given module execution from one machine to another.

# Chapter 5

# Conclusions

## 5.1 Two storage trends

The main obstacle to the imposition of CORBA is the fight between ODBMSs and RDBMSs. They are struggling for the market, and CORBA is a corner-stone in the future of software. It is the center nobody wants to lose. The storage system is probably the most important part of an information system. The way it is offered drives the future of the whole information system technologies.

The difference can be stated as providing a single-level or a two-level storage system. In the former, the client is not aware of whether the object is in memory or disk. The DBMS (ODBMS in this case) hides the location of the data to the users. From the user point of view, the objects are always in memory. The other kind of storage systems (RDBMSs) clearly separates memory and persistent storage. The user is responsible for explicitly retrieve and store the data.

Nowadays, most databases are relational. The ODBMSs are limited to some specific business areas (i.e. CAD, CAE, etc.), and do not seem to gain market space in others. Nevertheless, since CORBA uses object technologies, it seems the ODBMSs should be treated favourably. Moreover, the single-level storage provides better performance, and solves the impedance mismatch problem. However, the weight of RDBMSs in business is not negligible. Some CORBA specifications are a trade-off between what should be and what really is (e.g. POS specification and its interfaces).

Probably, the POS is the most clear example of what is happening. It was proposed to allow any kind of storage mechanism. It does not say how but what, and its interfaces are as general as possible in order to be used by relational as well as OO storage systems. However, the generality, in this case, favors a two-level store model at the expense of a single-level one, because it offers two operations, i.e. *store* and *restore*, which clearly is a two-level store view. The signs of this battle are also found in the OQS which forces, at least, two different query languages: SQL and OQL.

It is not mandatory to use the POS to get persistence nor the OQS to query storage systems, any other facility could be used instead of them. Thus, as a result of all these problems, none of the contenders fully backs those disputed services. The ODBMSs bet on the ODA solution, while the RDBMSs do not seem to bet on anything, yet. Trying to satisfy everybody is not always the best. The storage disagreement is a brake for CORBA.

The differences will likely (hopefully) be smoothed by means of the Transaction service, which can make a two-level store look like a one-level store, or vice versa, depending on the point of view. A one-level storage system could be seen as two-level storage system with the *restore*s grouped at the begin and the *store*s grouped at the commit of transaction (as proposed in [Ses96]).

## 5.2 Squaring CORBA and BLOOM

BLOOM and CORBA are two of a kind. The two of them make a good match to manage heterogeneities in multidatabase systems. While CORBA handles the system heterogeneities, BLOOM tries to solve the semantic ones. Clearly, it is a teamwork, where each one is specialized in a different facet. CORBA saves BLOOM problems in dealing with "second class" heterogeneities.

The system heterogeneities (in hardware, operating system, communication protocols, etc.) are cumbersome to study the integration of autonomous (or semi-autonomous) information systems. They are well understood, and have already proposed solutions (CORBA is a good example of this). Thus, when trying to study such systems, we could assume those heterogeneities are already solved. By means of CORBA, BLOOM can forget them and only look at the semantics of the different schemas to be integrated.

This does not mean we can just say the system heterogeneities are solved and go straight to the semantics. The interaction between both systems must be carefully studied. CORBA is a new born, still evolving standard, and there does not exist an implementation of BLOOM, yet. Therefore, neither supply nor demand are well stated.

This paper tried to give an overview of how that interaction could be seen. It seems clear to divide it into two different parts. The one is how CORBA can be used to wrap the CDBs, and the other is how CORBA can wrap the modules in the BLOOM execution architecture to help their distribution.

On wrapping the CDBs, CORBA has potential. However, at present day, it is not evident how will be done. There are a lot of helpful services, which have not been fully implemented, yet. Some, because have been standardized recently. Others, because it is not obvious how the standard will be concreted. Sometimes, the standard is quite vague, and the implementors got a high degree of freedom. It seems CORBA will do a good job, but how it will cannot be ensured by now. The most promising service is the OQS. However, an implementation is not available, yet.

About the second way CORBA can help BLOOM, it seems much more realistic. It does not depend so much on the services, but on the ORB itself. The ORB was standardized before the services. Thus, its implementations are better established in the market, and it is much easier to say what it does and what it will do. The idea here is to wrap each BLOOM module into a CORBA object to be able to ship them across the network. This will solve the location problems, and will help to distribute the load of the system between the different machines participating in the cooperative information system. By means of CORBA, every machine (actually machine administrator) can dynamically choose which modules are allowed to be run on it.

It is certain that CORBA will help BLOOM to build a federation of databases. Maybe, the question is when. Full CORBA implementations are not available today, and the standard is still evolving and growing. Nevertheless, there can be no doubt that when the standard become stable and the implementations well rooted in the market, CORBA will be the indispensable tool to build a cooperative information system.

# Glossary

**API** Application Programming Interface.

**BLOOM** BarceLona Object Oriented Model. Note that it is used to refer to the data model itself, as well as the whole project (canonical model, schemas architecture and execution architecture).

**BOA** Basic Object Adapter.

**CAD** Computer Aided Design.

**CAE** Computer Aided Engineering.

**CDB** Component DataBase.

**CDM** Canonical Data Model.

**CLI** Call Level Interface.

**COM$^{TM}$** Component Object Model (*Microsoft*).

**Cooperative Information System** Set of information systems, possibly distributed over large and complex computer/communication networks, which manage large amount of information and computing services, and support individual or collaborative human work. [MDJ$^+$98]

**CORBA** Common Object Request Broker Architecture.

**DA** Direct Access protocol.

**DB** DataBase.

**DBA** DataBase Administrator.

**DBMS** DataBase Management System.

**DDL** Data Definition Language.

**DDO** Dynamic Data Object.

**DII** Dynamic Invocation Interface.

**DLL** Dynamically Loaded Library.

**DSI** Dynamic Skeleton Interface.

**DSOM**$^{TM}$ Dynamic SOM (*IBM*).

**FDBMS** Federated DataBase Management System.

**Federated Database Management System** Collection of cooperating but autonomous component database systems, possibly heterogeneous. [SL90]

**GUI** Graphical User Interface.

**HEROS** HEteRogeneous Object System. [AULM98]

**IDL** Interface Definition Language.

**IIOP** Internet Inter-ORB Protocol.

**IML**$^{TM}$ Implementation Mapping Language (*Digital*).

**Interface Definition Language** Language used to define CORBA object interfaces (quite similar to C++).

**JDBC**$^{TM}$ Java DataBase Connectivity (*Sun Microsystems*).

**METU** Middle East Technical University (Turkey).

**MIND** METU INteroperable DBMS. [DDK+96]

**MML**$^{TM}$ Method Mapping Language (*Digital*).

**OA** (Generic) Object Adapter.

**Object** Identifiable, encapsulated entity that provides one or more services that can be requested by a client. [OMG95]

**Object reference** Object name that reliably denotes a particular object. Specifically, an *Object Reference* will identify the same object each time the reference is used in a request. An object may be denoted by multiple, distinct object references. [OMG95]

**ObjectStore**$^{TM}$ Concrete ODBMS implementation (*Ideal Object*).

**ODA** Object Database Adapter.

**ODBC** Open DataBase Connectivity.

**ODBMS** Object oriented DBMS.

**ODL** Object Definition Language.

**ODMG** Object Database Management Group.

**OID** Object IDentifier.

**OLE**$^{TM}$ Object Linking and Embedding (*Microsoft*).

**OMA** Object Management Architecture.

**OMG**$^{TM}$ Object Management Group.

**OO** Object Oriented.

**OQL** Object Query Language (defined by the ODMG).

**OQS** Object Query Service.

**ORB** Object Request Broker.

**OSL**$^{TM}$ Object Server Language (*Sun Microsystems*).

**Orbix**$^{TM}$ Concrete ORB implementation (*IONA Technologies*).

**PDS** Persistent Data Storage.

**PID** Persistent IDentifier.

**PO** Persistent Object.

**POA** Persistent Object Adapter.

**POM** Persistent Object Manager.

**POS** Persistent Object Service.

**Request** Event by means of which a client requests a service. The information associated with a request consist of an operation, a target object, zero or more (actual) parameters used to pass data to the target object, and an optional request context. [OMG95]

**RDBMS** Relational DBMS.

**RFI** Request For Information.

**RFP** Request For Proposals.

**RMI**$^{TM}$ Remote Method Invocation (*Sun Microsystems*).

**SD** Synchronized Data.

**SOM**$^{TM}$ System Object Model (*IBM*).

**SQL** Standard Query Language.

**SSDF** Standard Stream Data Format.

# Bibliography


[AULM98] E. M. Antunes-Uchoa, S. Lifschitz, and R. N. Melo. HEROS: A heterogeneous object-oriented database system. *Lecture Notes in Computer Science*, 1460:435–447, 1998.

[Bak97] Sean Baker. *CORBA Distributed Objects - Using Orbix.* ACM Press, Adison Wesley, 1997.

[CBB+97] R. G. G. Cattell, D. Barry, D. Bartels, M. Berler, J. Eastman, S. Gamerman, D. Jordan, A. Springer, H. Strickland, and D. Wade. *The Object Database Standard: ODMG 2.0.* Morgan Kaufmann Publishers, Los Altos (CA), USA, 1997.

[DDK+96] A. Dogac, C. Dengi, E. Kilic, G. Ozhan, F. Ozcan, S. Nural, C. Evrendilek, U. Halici, B. Arpinar, P. Koksal, and S. Mancuhan. A multidatabase system implementation on CORBA. In *Sixth International Workshop on Research Issues in Data Engineering - Interoperability of Nontraditional Database Systems*, pages 2–11, Washington - Brussels - Tokyo, February 1996. IEEE Computer Society.

[ION97] IONA Technologies. ORBIX + ObjectStore adapter. White paper, IONA Technologies PLC., 1997.

[KPT96] Jan Kleindienst, František Plášil, and Petr Tuma. Lessons learned from implementing the CORBA persistent object service. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, volume 31, 10 of *ACM SIGPLAN Notices*, pages 150–167, New York, October6–10 1996. ACM Press.

[KS98] G. Kappel and B. Schroeder. Distributed light-weight persistence in Java — A tour on RMI- and CORBA-based solutions. *Lecture Notes in Computer Science*, 1460:411–424, 1998.

[MDJ+98] Georgio De Michelis, Eric Dubois, Matthias Jarke, Florian Matthes, John Mylopoulos, Michael P. Papazoglou, Klaus Pohl, Joachim Schmidt, Carson Woo, and Eric Yu. Cooperative information systems: a manifesto. In Michael P. Papazoglou and Gunter Schlageter, editors, *Cooperative Information Systems*, pages 315–363. Academic Press, San Diego, 1998.

[OH98] Robert Orfali and Dan Harkey. *Client/Server Programming with JAVA and CORBA.* John Wiley & Sons, New York, 2 edition, 1998.

[OHE97] Robert Orfali, Dan Harkey, and Jeri Edwards. *Instant CORBA*. Wiley Computer Publishing, John Wiley and Sons Inc., 1997.

[OMG94] OMG. Object services architecture. Documentation available at http://www.omg.org, Object Management Group, 1994. Revision 8.0.

[OMG95] OMG. The common object request broker: Architecture and specification. Documentation available at http://www.omg.org, Object Management Group, July 1995. Revision 2.0.

[OMG98] OMG. CORBAservices: Common object services specification. Documentation available at http://www.omg.org, Object Management Group, December 1998.

[ROSC97] Elena Rodríguez, Marta Oliva, Fèlix Saltor, and Benet Campderrich. On schema and functional architectures for multilevel secure and multiuser model federated database systems. In S. Conrad et al., editor, *Proceedings of the International CAiSE'97 Workshop on Engineering Federated Database Systems (EFDBS'97)*, pages 93–104. Springer, Magdeburg (Germany), 1997.

[Ses96] Roger Sessions. *Object Persistence – beyond object-oriented databases*. Prentice Hall, 1996.

[Sie96] Jon Siegel. *CORBA: Fundamentals and Programming*. John Wiley & Sons Inc., New York, 1 edition, 1996.

[SL90] Amit P. Sheth and James A. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Surveys*, 22(3):183–236, September 1990. Also published in/as: Bellcore, TM-STS-016302, Jun.1990.

[Sri97] Prashant Sridharan. *Advanced Java Networking*. Prentice Hall, 1997. ISBN 0-13-749136.

[SV97] Douglas C. Schmidt and Steve Vinoski. Object interconnections. *SIGS C++ Report*, April 1997.

[Vin93] S. Vinoski. Distributed object computing with CORBA. *C++ Report*, 5(6):32–38, July-August 1993.

[Vin97] Steve Vinoski. CORBA: Integrating diverse applications within distributed heterogeneous environments. *IEEE Communications*, 35(2):46–55, February 1997.

# Appendix A

# Service IDL interfaces

## A.1 Externalization Service

```
module CosExternalization {
    exception InvalidFileNameError {};
    exception ContextAlreadyRegisteredError {};

    interface Stream : CosLifeCycle::LifeCycleObject {
        void externalize(in CosStream::Streamable theObject);
        CosStream::Streamable internalize(in CosLifeCycle::FactoryFinder there)
            raises (CosLifeCycle::NoFactory, StreamDataFormatError);
        void begin_context() raises (ContextAlreadyRegistered);
        void end_context();
        void flush();
        };
    interface StreamFactory {
        Stream create();
        };
    interface FileStreamFactory {
        Stream create(in string theFileName) raises (InvalidFileNameError);
        };
    };
```

```
module CosStream {
    exception ObjectCreationError {};
    exception StreamDataFormatError {};

    interface StreamIO;
    interface Streamable : CosObjectIdentity::IdentifiableObject {
        readonly attribute CosLifeCycle::Key external_form_id;

        void externalize_to_stream(in StreamIO targetStreamIO);
        void internalize_from_stream(in StreamIO targetStreamIO, in CosLifeCycle::FactoryFinder there)
            raises (CosLifeCycle::NoFactory, ObjectCreationError, StreamDataFormatError);
        };
    interface StreamableFactory {
        Streamable create_uninitialized();
        };
    interface StreamIO {
        void write_object(in Streamable obj);
        void write_string(in string aString);
        void write_char(in char aChar);
        ...
        Streamable read_object() raises (StreamDataFormatError);
        string read_string() raises (StreamDataFormatError);
        char read_char() raises (StreamDataFormatError);
        ...
        };
    };
```

## A.2 Persistent Object Service

```
module CosPersistencePO {
    interface PO {
        attribute CosPersistencePID::PID p;

        CosPersistencePDS::PDS connect (in CosPersistencePID::PID p);
        void disconnect(in CosPersistencePID::PID p);
        void store(in CosPersistencePID::PID p);
        void restore(in CosPersistencePID::PID p);
        void delete(in CosPersistencePID::PID p);
        };
    interface SD {
        void pre_store();
        void post_restore();
        };
    };
```

```
module CosPersistencePOM {
    interface POM {
        CosPersistencePDS::PDS connect (in Object obj, in CosPersistencePID::PID p);
        void disconnect(in CosPersistencePID::PID p);
        void store(in CosPersistencePID::PID p);
        void restore(in CosPersistencePID::PID p);
        void delete(in CosPersistencePID::PID p);
        };
    };
```

```
module CosPersistencePDS {
    interface PDS {
        PDS connect (in Object obj, in CosPersistencePID::PID p);
        void disconnect(in CosPersistencePID::PID p);
        void store(in CosPersistencePID::PID p);
        void restore(in CosPersistencePID::PID p);
        void delete(in CosPersistencePID::PID p);
        };
    };
```

```
module CosPersistencePDS_DA {
    typedef string DAObjectID;
    typedef sequence<string> AttributeNames;
    typedef string ClusterID;
    typedef sequence<ClusterID> ClusterIDs;

    interface PID_DA : CosPersistencePID::PID {
        attribute DAObjectID oid;
        };
    interface DAObject {
        boolean dado_same(in DAObject d);
        DAObjectID dado_oid();
        PID_DA dado_pid();
        void dado_remove();
        void dado_free();
        };
    interface DAObjectFactory {
        DAObject create();
        };
    interface DAObjectFactoryFinder {
        DAObjectFactory find_factory(in string key);
        };
    interface PDS_DA : CosPersistencePDS::PDS {
        DAObject get_data();
        void set_data(in DAObject new_data);
        DAObject lookup(in DAObjectID id);
        PID_DA get_pid();
        PID_DA get_object_pid(in DAObject dao);
        DAObjectFactoryFinder data_factories();
        };
    interface DynamicAttributeAccess {
        AttributeNames attribute_names();
        any attribute_get(in string name);
        void attribute_set(in string name, in any value);
        };
    interface PDS_ClusteredDA : PDS_DA {
        ClusterID cluster_id();
        string cluster_kind();
        ClusterIDs clusters_of();
        PDS_ClusteredDA create_cluster(in string kind);
        PDS_ClusteredDA open_cluster(in ClusterID cluster);
        PDS_ClusteredDA copy_cluster(in PDS_DA source);
        };
    };
```

```
module CosPersistenceDDO {
    interface DDO {
        attribute string object_type;
        attribute CosPersistencePID::PID p;

        short add_data();
        short add_data_property(in short data_id);
        short add_data_count();
        short add_data_property_count(in short data_id);
        void get_data_property(in short data_id, in short property_id, out string property_name, out any property_value);
        void set_data_property(in short data_id, in short property_id, in string property_name, in any property_value);
        void get_data(in short data_id, out string data_name, out any data_value);
        void set_data(in short data_id, in string data_name, in any data_value);
        };
    };
```

## A.3 Object Query Service

```
module CosQueryCollection {
    exception ElementInvalid ;
    exception IteratorInvalid ;
    exception PositionInvalid ;

    enum ValueType { TypeBoolean, TypeChar, TypeOctet, TypeShort, TypeUShort, TypeLong, ..., TypeNumeric};
    struct Decimal { long precision; long scale; sequence<octet> value; };
    union Value switch(ValueType) {
        case TypeBoolean: boolean b;
        ...
        case TypeNumerical: Decimal n;
        };
    typedef boolean Null;
    union FieldValue switch (Null) {
        case FALSE: Value v;
        };
    typedef sequence<FieldValue> Record;
    typedef string wstring;
    struct NVPair { wstring name; any value; }
    typedef sequence<NVPair> ParameterList;

    interface Collection;
    interface Iterator;
    interface CollectionFactory {
        Collection create(in ParameterList params);
        };
    interface Collection {
        readonly attribute long cardinality;

        void add_element(in any element) raises (ElementInvalid);
        void add_all_elements(in Collection elements) raises (ElementInvalid);
        void insert_element_at(in any element, in Iterator where) raises (IteratorInvalid, ElementInvalid);
        void replace_element_at(in any element, in Iterator where) raises (IteratorInvalid, PositionInvalid, ElementInvalid);
        void remove_element_at(in Iterator where) raises (IteratorInvalid, PositionInvalid);
        void remove_all_elements();
        any retrieve_element_at(in Iterator where) raises (IteratorInvalid, PositionInvalid);
        Iterator create_iterator();
        };
    interface Iterator {
        any next() raises (IteratorInvalid, PositionInvalid);
        void reset();
        boolean more();
        };
    };
```

```
module CosQuery {
    exception QueryInvalid ;
    exception QueryProcessingError string why;;
    exception QueryTypeInvalid ;

    enum QueryStatus { complete, incomplete };
    typedef CosQueryCollection::ParameterList ParameterList;
    typedef CORBA::InterfaceDef QLType;

    interface QueryLanguageType {};
    interface SQLQuery : QueryLanguageType {};
    interface SQL_92Query : SQLQuery {};
    interface OQL : QueryLanguageType {};
    interface OQLBasic : OQL {};
    interface OQL_93 : OQL {};
    interface OQL_93Basic : OQL_93, OQL Basic {};
    interface QueryEvaluator {
        readonly attribute sequence<QLType> ql_types;
        readonly attribute QLType default_ql_type;

        any evaluate(in string query, in QLType ql_type, in ParameterList params)
            raises (QueryTypeInvalid, QueryInvalid, QueryProcessingError);
        };
    interface QueriableCollection : QueryEvaluator, CosQueryCollection:Collection {};
    interface QueryManager : QueryEvaluator {
        Query create(in string query, in QLType ql_type, in ParameterList params) raises (QueryTypeInvalid, QueryInvalid);
        };
    interface Query {
        readonly attribute QueryManager query_mgr;

        void prepare(in ParameterList params) raises (QueryProcessingError);
        void execute(in ParameterList params) raises (QueryProcessingError);
        QueryStatus get_status();
        any get_result();
        };
    };
```