CORBA: A PRACTICAL INTRODUCTION

Serguei Kolos [1])
CERN, Geneva, Switzerland

Abstract
Common Object Request Broker Architecture (CORBA) is the recent answer for the need for interoperability in the modern distributed computing environment. This architecture allows objects to communicate with one another regardless of their location and implementation. CORBA hides the differences between the operating systems, programming languages and address spaces for the objects implementations.
This text will give an overview of the CORBA along with the Java programming example that illustrates the process of the CORBA based distributed system development. At the end some examples of how CORBA is currently being used in HEP experiments will be given.

1. INTRODUCTION

Throughout the history of computing the demand for distributed computation has been growing permanently. The evolution in this area has been always moving to the direction of increasing level of abstraction for the communication implementation. The first major step on this road was so-called *socket*[1] interface that unifies the way in which a low-level protocol have been used. It is a relatively simple set of generic procedures that allow bidirectional data exchange between applications regardless of their location. The applications could be executing either on the same or on different computers even if those computers run different operating systems.

The Remote Procedure Call (RPC)[2] layer brings into use the idea of control messages exchange instead of just data exchange between distributed applications. RPC gives the possibility to unify local and remote procedures invocations at the level of programming language code. The remote procedures could be declared and called exactly in the same way as the local ones with the only omission that the invocation errors shall be handled differently.

Another step has been done by the Distributed Computing Environment (DCE)[3] system. DCE is a product of the Open Software Foundation (OSF)[4] that puts into practice the idea to use a special language for the declaration of communication interfaces. The Interface Definition Language (IDL) has been promoted to provide a service interface description. The IDL declaration is the only information required for the development of both a service provider and a service requestor for the interface it describes. The important result of the IDL innovation is the ability to use different programming languages for the implementation of the applications that are taking part in communications.

Each round in this evolution was another step on the road of unification of the local and distributed computing models. The resent step in this evolution was the introduction of the CORBA standard by the Object Management Group (OMG)[5] in 1991. The major innovation of the OMG that quickly made CORBA the leading communication standard was the application of the Object technology to the communication domain. CORBA puts into practice the Object Oriented Design (OOD) methods for software engineering. OOD as a method of modelling a problem by taking a balanced view about objects and the operations performed upon them, was proposed by Grady Booch[6]. The classical OOD model does not take care of the objects' implementation. It defines the object interfaces and relationships regardless of where the objects are and how they have been implemented. But for a number of years this model had been rather a theoretical abstraction because of

1. On leave from the Petersburg Nuclear Physics Institute, Gatchina, Russia

the essential gap between the OOD model and implementation technologies. The CORBA standard is a great contribution to the applicability of the OOD technology to a real computing problems. CORBA acts as a bridge between the object design methods and objects' implementation. The Object-Oriented IDL perfectly maps to the OOD model allowing to express in a formal way any OOD paradigm. Most of the modern OOD tools like Rose[7], StP[8], Together[9], etc. are able to generate automatically the interfaces in OMG Interface Definition Language (IDL)[1] from the design artifacts. This IDL definition is substantial for the implementation of the objects that can be used either locally or remotely.

The next two chapters give the overview of the CORBA standard and the OMG reference architecture. An example that shows how CORBA can be used for the implementation of the system has been designed using OOD methods will be shown in the next chapter. At the last chapter some references to the applications of CORBA in HEP experiments are given.

2. CORBA OVERVIEW

CORBA standard is based on the classical object model[10] that defines two kinds of distinct entities: classes that support encapsulation, inheritance and polymorphism and objects that are classes' instances. The fundamental principle that was recognized by CORBA is the independence of the behavior of an object from its implementation. The behavior of an object is defined by it's interface, where the interface is a set of object method signatures. From this point of view the CORBA standard can be seen as consisting of two logical levels:

- Interface definition level: Defines syntax and semantics for the objects' interface description by means of OMG IDL[11]. The way in which IDL interfaces can be mapped to a programming language is standardized by the CORBA Language Mappings[12]. The Language Mapping specifications provide a link to the next CORBA level.
- Communication media level: Specifies how communications have to be implemented. For this purpose the Object Request Broker (ORB) specification is introduced. The ORB acts as an inter-object communication bus providing a set of interfaces for object creation, registration and access control.

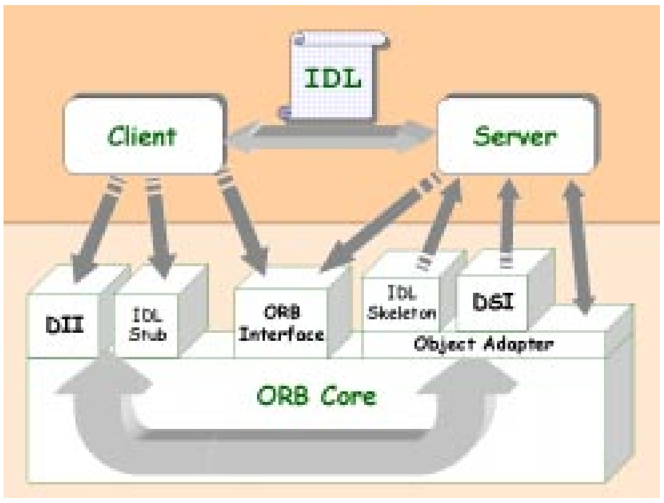


Fig. 1 The structure of the CORBA standard

Figure 1 shows the structure of the CORBA standard. Different components of the ORB are briefly described in the next paragraphs. Those who are interested in a more detailed description we can address to the respective chapters of the [13].

1. Do not confuse it with the DCE IDL. The OMG IDL is essentially different from the DCE's one.

2.1 Object Interface Definition

The basis for communication description is formed by the OMG IDL that was the first component defined by the CORBA standard. The OMG IDL is widely accepted as the *de facto* standard for objects' interface description. It is often used as a general purpose interface description tool even for a description of a non distributed objects interfaces. There are two fundamental features of the OMG IDL that make it so popular:

- OMG IDL is able to provide a comprehensive object interface description using intuitive self-explained semantics. A service description, has been done on OMG IDL, provides all the necessary information for the independent development of both a service provider and a service consumer.
- Any IDL construct can be easily mapped to most of the existing programming languages.

These facilities make the OMG IDL extremely useful for the OOD. It is IDL that bridges object design and object implementation phases of the software life cycle. This chapter gives an overview of the IDL and explains what the CORBA Language Mappings are.

2.1.1 OMG Interface Definition Language

As it was already mentioned an IDL declaration looks very self-explained and understandable because the OMG IDL uses the same grammar and lexical rules as C++. Most of the IDL's keywords are well known for the software developers and ordinarily it takes them just a few hours to learn IDL.

The key concept of an IDL declaration is an interface. Interface declaration consists of an interface header and interface body. Interface header consists of the optional keyword '*abstract*', the interface name and optional inheritance specification. An interface body contains attributes and operations declarations. Figure 2 shows an example of the IDL interfaces declaration.

```
abstract interface container {
   long size();
};

interface iterator: container {
   any next();
}

interface list: container {
   void add(in any elem);
   iterator create_iterator();
};
```

Fig. 2 IDL interfaces declaration

An interface can be derived from another interface, which is then called a base interface. A derived interface can declare it's own attributes and operations. Multiple inheritance is allowed - an interface may be derived from any number of base interfaces. The order of derivation is not significant.

Data can be defined only as attributes of the interfaces. The attribute declaration means that a respective accessor an mutuator methods will be available for this attribute. If an attributed is prefixed with the read-only keyword only the accessor method will be available for the specific attribute. Thus the attribute construct does not declare a storage for the data value but instead just defines a data access interface. The IDL declarations shown in Table 1 are equivalent.

Table 1 Two alternative ways of interface description

`interface person {`	`interface person {`
	`void set_age(in octet ag);`
`attribute octet age;`	`octet get_age();`
`read-only attribute string name;`	`string get_name();`
`};`	`};`

The OMG IDL specification define a number of basic types that can be used for the attributes and method parameters declarations. These types covers all the possible basic computational units in programming languages. They are shown in Table 2.

Table 2 Basic OMG IDL types

OMG IDL type	Matched by
char, wchar	simple and wide character literals
octet	unsigned 1-byte value
short, unsigned short	signed and unsigned 2-byte values
long, unsigned long	signed and unsigned 4-byte values
long long, unsigned long long	signed and unsigned 8-byte values
float, double	simple and double precision floating point values
string, wstring	sequences of simple and wide character literals
enum	set of user defined values

These basic types can be used to construct complex ones by utilizing the structure, union and sequence patterns. Figure 3 shows an example of the complex types definition.

```
enum PrimitiveKind{
   pk_unknown, pk_char, pk_octet, pk_short, pk_ushort,
   pk_long, pk_ulong, pk_float, pk_double, pk_boolean, pk_string,
};

union DataEntry switch(PrimitiveKind){
   case pk_char:          char           de_char;
   case pk_octet:         octet          de_uchar;
   case pk_short:         short          de_short;
   case pk_ushort:        unsigned short de_ushort;
   case pk_long:          long           de_long;
   case pk_ulong:         unsigned long  de_ulong;
   case pk_float:         float          de_float;
   case pk_double:        double         de_double;
   case pk_boolean:       boolean        de_bool;
   case pk_string:        string         de_string;
};

struct NamedDataEntry{
   string     name;
   DataEntry  entry;
};

typedef sequence<DataEntry> Data;

typedef sequence<NamedDataEntry> NamedData;
```

Fig. 3 OMG IDL complex types

There are some other constructs in IDL that have not been mentioned yet. They are *exceptions* that are used to indicate the exceptional conditions of the methods execution, the *oneway* keyword used to declare the asynchronous style of the method invocation, the *module* that is the namespace declaration that prevents the names declared in the scope of this module from clashes with other IDL declarations, etc. For the complete specification of the IDL syntax and semantics see [11].

OMG IDL is a permanently evolving standard that tracks the major innovations in the OOD world while keeping the backward compatibility with the previous versions of the specification. It give us a good reason to be optimistic for the future of the OMG IDL as a basic method of objects' interface description.

2.1.2 Languages Mappings

IDL is purely declarative language. It is used to declare interfaces and can not be used for their implementation. IDL is used to express the OOD patterns in a formal way but it is still an abstraction that requires another type of formal definition that specifies how IDL interfaces can be mapped to the 'real world' of software implementation. Such specification is provided by the CORBA standard by means of Language Mappings for several programming languages.

A Language Mapping defines a programming language counterpart for each construct of the OMG IDL. This definition allows to generate programming code from the IDL declaration automatically. This programming code is an interface declaration that is equivalent to the IDL one but is expressed by means of a programming language.

For various languages these mappings are essentially different. For example *interface* declaration corresponds to *class* in C++, *interface* in Java and *structure* in C language. All the conformity between the IDL and a programming language are formally described by the standard in such a way that source code compatibility between different ORBs will very probably be possible in the nearest future. Table 3 shows the Java mappings for the IDL types accordingly to the OMG specification[14].

Table 3 Java mappings for OMG IDL types

OMG IDL type	Java mapping types
short, long, long long	short, int, long
unsigned short, unsigned long, unsigned long long	unsigned short, unsigned int, unsigned long
float, double	float, double
char, boolean, octet	char, boolean, byte
any	org.omg.CORBA.Any class
string, wstring	string
struct, union, enum, exception	class
sequence<type>	type[]
interface	interface
module	package

Currently, the CORBA standard defines mappings for the following languages: C, C++, Smalltalk, COBOL, Ada, Java and Lisp. Several independent companies develop their own ORBs with the mappings to languages that are not part of the CORBA standard. For example Xerox[15] has an ORB called Inter Language Unification (ILU)[16] with mappings to the Python and Modula2 languages. There are no limitations for the use of such language mappings for the CORBA object development. The CORBA client implemented in Lisp can call the methods of a CORBA server that have been written in any of the ‘standard’ languages and vice versa. The only trouble with a non-standard mappings is a possible source code incompatibility between different ORB implementations.

2.2 Object Request Broker architecture

As it was already mentioned a programming language code can be automatically generated from the IDL declaration. This task is done by a special application called an IDL translator or an IDL compiler. The IDL compiler takes an IDL statements as input and produces a code for a specific programming language according to the OMG language mapping specification. Thus the IDL compiler is a glue that links together two CORBA levels: an abstract interface definition and a concrete ORB connection implementation.

2.2.1 Stubs and Skeletons

The target code generated by the IDL translator appears in pairs: client-side and server-side. The client-side mapping is called *stub* or *proxy* - it is a mechanism that creates and issues requests from a client. The server-side code is called *skeleton* - it is a mechanism that delivers requests to CORBA Object implementations. Such code is statically bound with the respective server or client code at compile and link time. Figure 4 shows the workflow for CORBA application development.

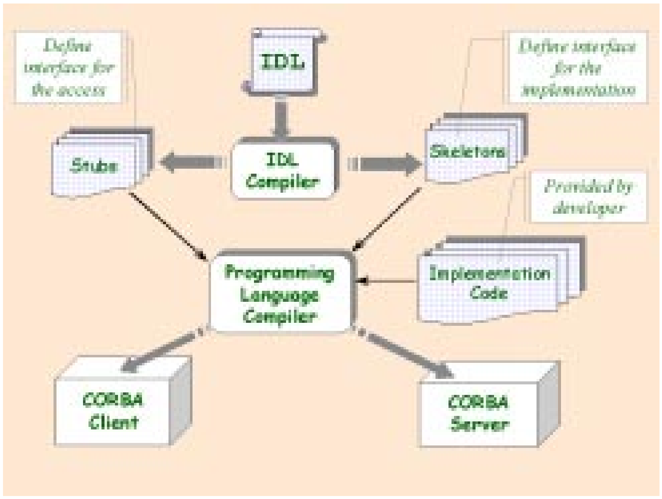

Define interface for the access
IDL
Define interface for the implementation
Stubs
IDL Compiler
Skeletons
Provided by developer
Programming Language Compiler
Implementation Code
CORBA Client
CORBA Server

Fig. 4 Workflow for CORBA application development

Before compilation the skeleton code needs to be fleshed out with the actual implementation code for each method. Stubs are basically complete. The *stub* methods can be used directly to request a service described by the IDL.

When a *stub*’s method is called by a client application it marshals a request to the ORB Core doing a conversion of a request from the programming language representation to one that is suitable for transmission over the connection to the target object. Then the ORB Core is responsible for the request transportation to the server that holds the target object and passing this request to the *skeleton* code. The *skeleton* unmarshals the request doing a conversion to a programming language, that is not necessarily the same as for the client application, and dispatches the request to the appropriate object. Dispatching through stubs and skeletons are often called *static invocation*. It is shown in Figure 5.

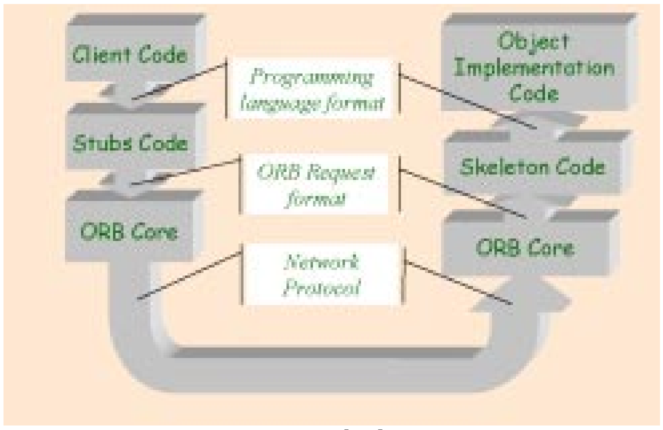


Fig. 5 Static method invocation

It is necessary to mention that the ORB Core that does the request transportation can be represented in practice by the combination of the two Core components belonging to different ORBs. In other words a server and a client application can be implemented using different ORBs.

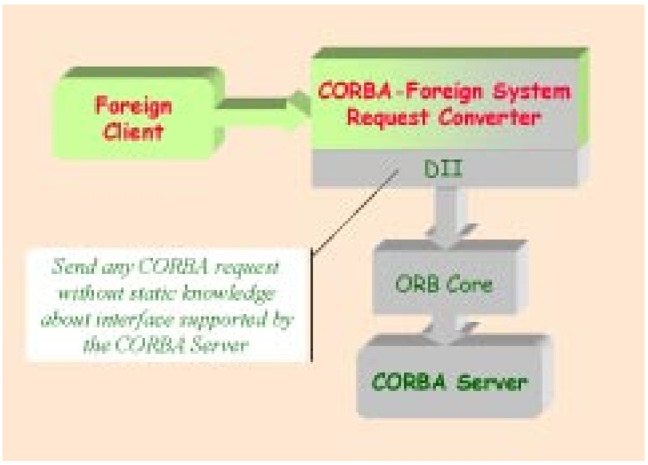
2.2.2 Dynamic Invocation

In addition to static invocation via stubs and skeletons, CORBA supports two interfaces for dynamic invocation:

- Dynamic Invocation Interface (DII)[17] supports dynamic client request invocation.
- Dynamic Skeleton Interface (DSI)[18] provides dynamic request dispatching to the implementation objects.

DII and DSI can be viewed as a generic stub and generic skeleton respectively. These interfaces are provided by the ORB and are independent from IDL interfaces of the objects being invoked. The main purpose of these generic interfaces is to support the implementation of the bridges between CORBA and non-CORBA communication systems.

Using DII a client application can invoke requests without having compile-time knowledge of the object's interfaces. A request consists of an object reference, an operation name and a list of parameters. Each parameter has a name and a value. Parameters' order is essential and must confirm to the order in which they are defined for the interface. Figure 6 shows how the generic bridge can be implemented using DII.


Foreign Client
CORBA-Foreign System Request Converter
DII
Send any CORBA request without static knowledge about interface supported by the CORBA Server
ORB Core
CORBA Server
Fig. 6 The DII based bridge between an ORB and a non-CORBA system

The counterpart for the DII is DSI. DSI allows servers to be implemented without having skeletons for the objects compiled statically into the program. This concept was introduced in CORBA 2.0 as a possible mean of interoperability implementation. The DSI based bridge architecture is presented in Figure 7.

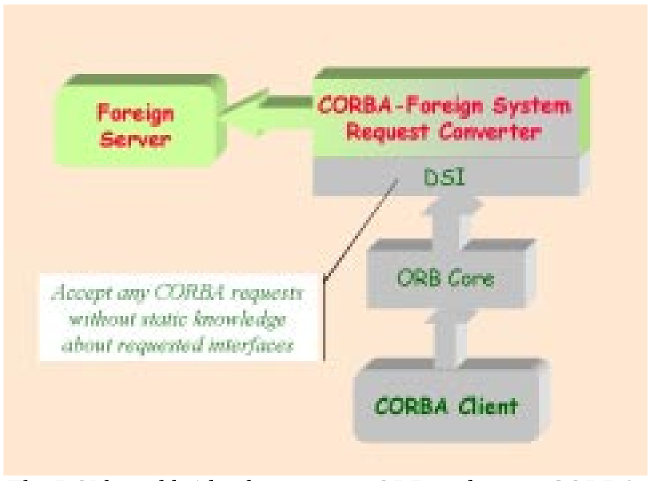

Foreign Server
CORBA-Foreign System Request Converter
DSI
ORB Core
Accept any CORBA requests without static knowledge about requested interfaces
CORBA Client

Fig. 7 The DSI based bridge between an ORB and a non-CORBA system

2.2.3 ORB Core

The main component of an Object Request Broker is called the ORB Core. It serves as a communication bus for the request transportation. The ORB Core assures the transparency of the following communication aspects:

- Object location: Mutual location of the server and client objects is transparent on the level of implementation code. The objects can be parts of one program instance, sharing runtime support in one memory image; they can be parts running in different program instances on different machines connected either with a local or global network.
- Communication protocol: The ORB has to use the same communication protocol (e.g. TCP/IP, UDP, RPC over TCP/IP, etc.) on both client and server ends of connection. But it is completely transparent for the programming objects which communication mechanism is actually used. This mechanism can be swapped with another one without affecting the objects' implementation.

2.2.4 Object Adapter

The Object Adapter provides an intermediate layer between the object implementation and the ORB Core. The OA is responsible for the following operations:

- Object reference handling: Object references are created in servers. Once they have been created, they may be exported to clients. From this model's perspective, object references encapsulate object identity information and information required by the ORB to identify and locate the server and OA with which the object is associated.
- Object request transportation: When a client issues a request, the ORB first locates an appropriate server and then locates the appropriate OA within that server. Once the ORB has found the appropriate OA, it delivers the request to that OA. Then the OA invokes the appropriate method on the request's target object.

The first specification for the object adapter provided by OMG was the Basic Object Adapter (BOA). But very soon it was found that some important BOA operations that had not been clearly defined by the standard were implemented very differently by various ORB vendors. The main differences have been the object registration and object activation operations which result in nontrivial portability problems between different ORBs.

OMG recognized these problems and a new Portable Object Adapter (POA)[19] specification was established in the recent CORBA versions. POA addresses the following issues that have been missed or not fully addressed by the BOA specification:

- Object portability: Allow programmers to construct object implementations that are portable between different ORB products.
- Object persistence: Provide support for objects whose lifetimes span multiple server lifetimes.
- Object activation: Provide support for transparent activation of objects.

2.2.5 ORB Interface

The ORB Interface[20] defines the application program interface for the operations implemented by the ORB. These operations are the same among all CORBA brokers and do not depend on the particular object adapter used. The main responsibility of the ORB Interface is to provide a portable means by which service implementation objects can be accessed by clients willing to utilize these services. The ORB Interface defines such access mechanism by the means of object reference handling. An object reference may be translated into a string by the operation *object_to_string*. The string value may be stored or communicated in whatever ways strings may be manipulated. Subsequently, the *string_to_object* operation will accept a string produced by object_to_string and returns the corresponding object reference. The string format must be recognized by any ORB implementation. Figure 8 shows how a client application can establish a connection to a servant using the ORB Interface methods described above.

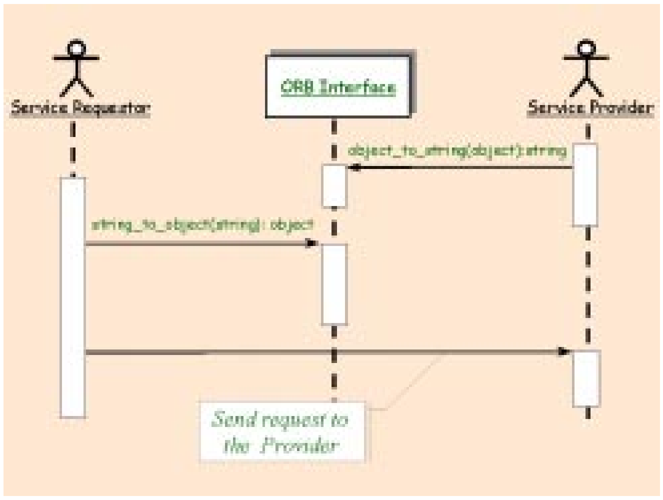

Service Requestor
ORB Interface
Service Provider
object_to_string(object):string
string_to_object(string): object
Send request to the Provider

Fig. 8 CORBA objects connectivity Sequence diagram (UML notation)

2.3 Interoperability

The original request for interoperability was issued by the OMG in 1993. It defined interoperability as an ability of a client on ORB ‘A’ to invoke an OMG IDL defined operation on an object on ORB ‘B’. It is assumed that CORBA brokers ‘A’ and ‘B’ have been developed independently. In the recent CORBA specifications the idea of interoperability is defined as “Interoperability is a comprehensive flexible approach to support network of objects implemented and managed by multiple heterogeneous CORBA-compliant ORBs”[21].

The interoperability architecture defines two concepts to achieve interoperability: mediate and immediate bridging of ORBs. Let us assume that any ORB implementation or any other communication standard forms its own domain of communication. For the mediate bridges, elements of one domain are transformed from the format internal to this domain to another format that is agreed to be common for all domains participating in interactions. For immediate bridging, elements of the interaction are transformed directly from the internal format of one domain to the internal format of the other.

The Generic Inter-ORB Protocol (GIOP)[22] can be seen as the common basis for the mediated bridging approach implementation. The protocol specification is the common agreed format that is recognized by any interoperable ORB. GIOP specifies a standard transfer syntax and a set of message formats for communications between ORBs. The GIOP protocol is simple, scalable and relatively easy to implement.

The Internet Inter-ORB Protocol (IIOP)[22] specifies how the GIOP messages are exchanged through TCP/IPC connections. IIOP can be seen as a mapping of GIOP for a specific transport. The IIOP is the standard protocol that is supported by almost any of the current ORBs as the default one. Thus, practically, most of the existing CORBA brokers are interoperable on the level of IIOP and are able to communicate with one another.

An example of an immediate bridge is a bridge between a CORBA and non-CORBA system for which there is no common intermediate message exchange format. The Figures 6 and 7 shown the examples of such an approach.

An ORB is considered to be fully interoperability-compliant when it supports both the IIOP protocol and standard CORBA interfaces such as ORB Interface, DSI and DII.

3. OBJECT MANAGEMENT ARCHITECTURE

While the modern computing paradigm tends to the distributed computing, the distributed software products becomes more and more complex and the software life circle issues becomes more and more important. The critical parameters for the modern software are: the time to develop it, the ability to maintain and enhance it and the time it takes to learn to use it. The Object Management Group provides a reference architecture that is called Object Management Architecture (OMA)[23] in order to address these issues. The OMA defines a common framework that is intended to simplify the information systems development and support via the definition of the joint public services based on the common standard. The communications heart of the OMA is Object Request Broker component. As it is shown on Figure 9 the ORB joins three main OMA components:

- Applications Objects: These are the CORBA objects implemented by independent developers intended to fulfill their specific needs. These objects can be reused by the other developers and they might become a candidates for the OMG standardization.
- Object Services: The Object Services standardize the life cycle management of the distributed objects. They will be explained in more details in the following section.
- Common Facilities: They provides a set of generic application functions that can be configured to the requirements of a specific configuration. The facilities already formalized by the OMG are Internationalization, Time and Mobile Agent [24].

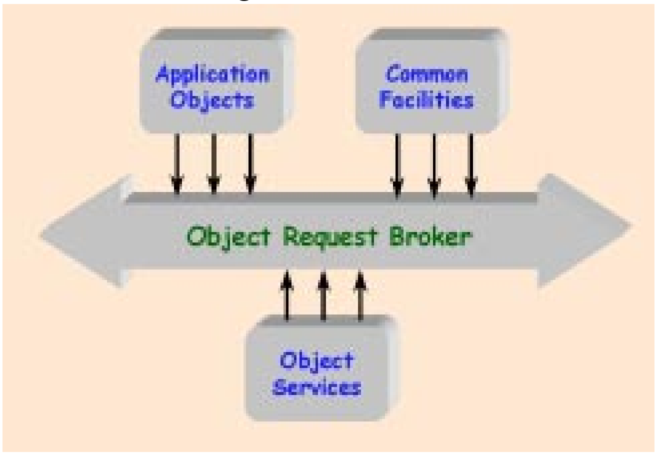


Fig. 9 Object Management Architecture

3.1 OMG Services

The Object Services cover different aspects of the object management including the objects creation, control access to objects, objects relocations and the objects relationship maintenance. The Object Service components provide the generic environment in which single objects can perform their tasks. Standardization of Object Services leads to consistency over different applications and improved productivity for the developer. Specifications for the Object Services that have been adopted as standards by the OMG are contained in [25]. There are sixteen service specifications provided by the OMG at the moment. The first service for which the definition has been provided and the most often used one is the Naming Service that will be explained in a more details below in order to give an impression of a CORBA Service essence to the reader.

3.1.1 Naming Service

The Naming Service[26] specification defines a federated (hierarchical) naming service that is commonly used to allow to use a human-readable format for the programming objects references. It does this by providing the name-to-object associations from which any object can be uniquely identified by the associated name.

A name-to-object association is called a name binding. A name binding is always defined relative to a naming context. A naming context is an object that contains a set of name bindings in which each name is unique. Different names can be bound to an object in the same or different contexts at the same time. To resolve a name is to determine the object associated with the name in a given context. To bind a name is to create a name binding in a given context. A name is always resolved relative to a context, there are no absolute names. Because a context is like any other object, it can also be bound to a name in a naming context. Figure 10 shows how the Naming Service can be used for the objects registration and access control. One can notice that this figure is very similar to the Figure 8 that shows how object connection can be established via the ORB Interface facility. The Naming Service methods are intended to be used instead of the *string_to_object* and *object_to_string* pair.

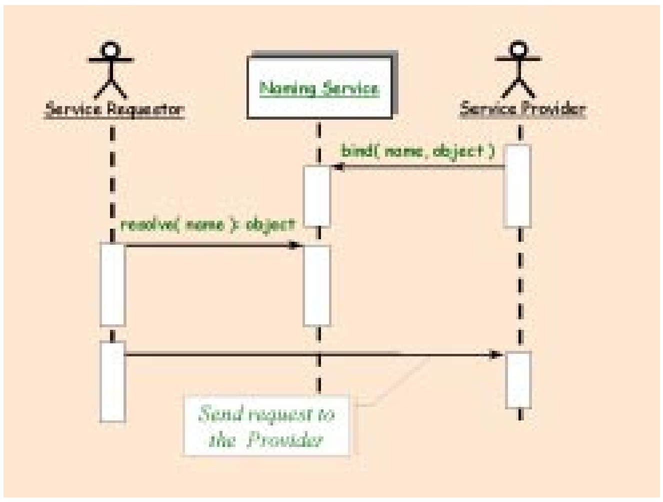


Fig. 10 Naming Service Sequence diagram (UML notation)

The CosNaming Module is a collection of interfaces that together define the naming service. This module is described in OMG IDL and contains two interfaces:

- The NamingContext interface - allows objects bindings and names resolution;
- The BindingIterator interface - allows to iterate through the bindings.

4. DEVELOPING A DISTRIBUTED APPLICATION WITH CORBA

In this chapter a comprehensive example of CORBA based distributed system design and development will be given. For the design and implementation described in this chapter the following software development tools have been used:

- Rational Rose[7] framework has been used for the diagram drawing and classes definition. Rose is an analysis and design framework that enables business analysts and software developers to specify business models and software applications graphically.
- Two ORBs have been used for the development: the JavaIDL[27] that is a Sun Microsystems Java ORB included to the JDK 1.2[28] and ORBacus 3.1.3[29] that is a C++/Java ORB of Object Oriented Concept[30].
- The *idltojava*[31] compiler version 1.2 that is the IDL compiler of Sun Microsystems has been used to generate Java *stub* code.
- The *idl* compiler that is the part of ORBacus 3.1.3 distribution has been used to generate C++ *skeleton* code.

4.1 Problem definition and proposed solution

High energy physics experiments investigate reactions between colliding elementary particles. To this purpose data on the particles leaving the collision point are recorded in large detectors and stored in digital form. The set of data recorded per collision is called an event. The events are the basic units for further investigations, which are done by powerful pattern recognition and analysis programs. One of the approaches used for the physical data estimation is a visual analysis of single events. For the event visualization a special class of application called Event Display is used. An Event Display provides independent method for the estimation of the information relevance by visualizing the event taken from the event storage system.

One of the important issues for the Event Display is the possibility to run it remotely, i.e. on a computer that does not have direct access to the event storage system. The possible solution might be to run the Event Display application on the machine that holds the event storage system while redirecting it’s output to the user’s machine via the standard X server display redirection facility. The critical point here is the network performance that sometimes is not enough to use this approach. As a solution we propose to separate out Event Display into two subtasks:

- Event Painter task that is the application that retrieves event information from the data storage and paints the event image in memory, but does not display it. This application is running on the machine with a direct data storage access. This Event Painter is in fact the classical Event Display application with the only difference that the Painter does not display the event image it has prepared.
- Event Visualizer is the task that retrieves the image from the Event Painter task via a CORBA interface and display this image on a user machine. This application is running on a remote machine that can not access the event storage system directly.

The Event Visualizer identifies an event it is willing to present. This identification can be done by passing the run and event numbers to the Event Painter. The Painter prepares an image in memory and passes it back in the machine independent format. For this simple example one of the well known graphics formats can be used here, for example JPEG, GIF, PNG, etc. The most suitable one in fact is a GIF format because it supports a suitable data compression without major loss of the image quality and can be easily displayed by the standard means of Java language. Figure 11 shows the information exchange between these tasks.

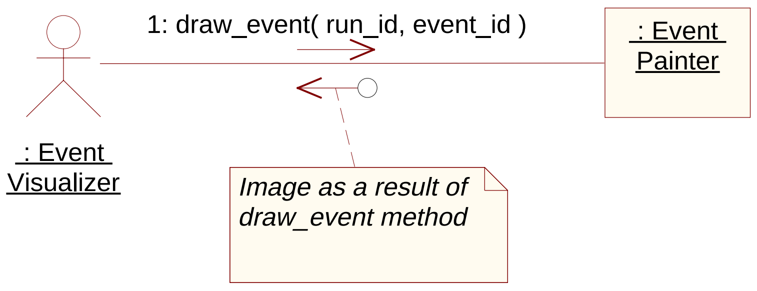

1: draw_event(run_id, event_id)
: Event Painter
: Event Visualizer
Image as a result of draw_event method

Fig. 11 Event Display Collaboration diagram (UML notation)

In reality the complexity of the modern detectors and events has increased so significant that simple non-interactive event image is not always suitable for the visual analysis. The modern Event Display is an interactive tool that can perform different image transformations like zoom, rotation, etc. upon a user request. For such facility it is necessary to use another image description format, for example XML[32].

Use of CORBA for this system implementation offers the important advantage: the most suitable programming languages can be chosen for the Painter and Visualizer implementation. It seems that visualization task can be better implemented on Java but the Painter may require a different language that is defined by the event storage system API. Most of the event storages that are widely used in physics experiments are not accessible from Java and provides ordinarily a Fortran or C/C++ interfaces. So that it worth using a C++ for the Event Painter implementation.

Another significant advantage of the proposed approach is independence of the Event Visualizer from the physical nature of events and detector geometry. Different experiments require different visualization technics for the experimental data representation. Therefor different implementations of the Event Painter must be provided but the same Event Visualizer can be used for all of them.

The definition of the CORBA interface for the Event Painter task and implementations for the both event analysis and event visualization tasks are discussed below.

4.2 Events access use cases

As it was discussed in the previous section the Event Painter interface is able to draw an event that is identified by the run and event numbers supplied by the Event Visualizer. But the Event Visualizer must have a way to find these numbers because it can not retrieve them from data storage itself since it has no access to it. The simplest way is to add to the Event Painter interface the methods that return the lists of run and event numbers. Thus the first operation to be done by the Event Visualizer is a request for the list of valid run numbers. Then for any run number it asks for a list of valid event numbers. This operation can result in the *BadRunNumber* exception if the wrong run number has been provided. In the opposite case the event drawing operation can be requested. The possible exceptions for this operation are: *BadRunNumber* and *BadEventNumber.* Upon a successful completion the event image can be displayed. Figure 12 shows the use cases described above.

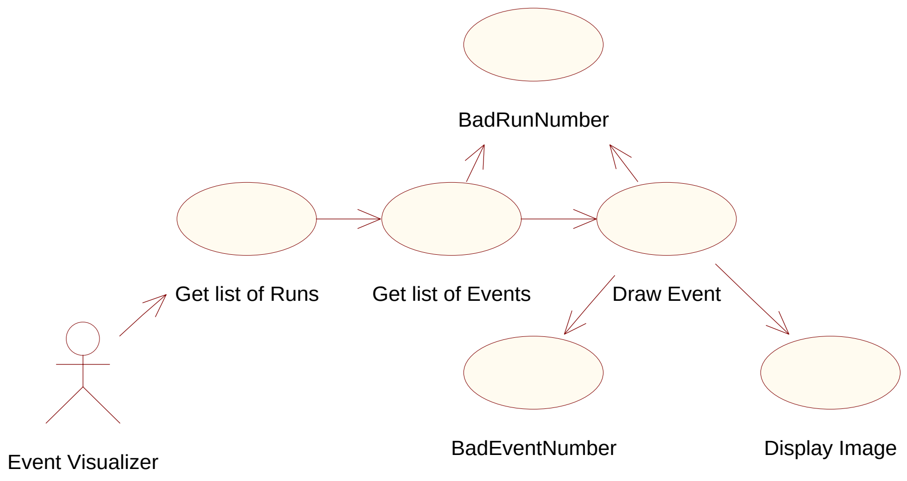


Fig. 12 Event Display Use-Case diagram (UML notation)

4.3 Classes definition

The next step towards the implementation is the definition of the classes that should be capable of handling the use cases described above. The interface called Painter and four data types have been defined for this purpose. The Painter interface declares the methods which cover all the aspects of the interface required by the Event Visualizer task. The data types support the return values and possible exceptions for the Painter's methods. These classes are shown on the Figure 13.

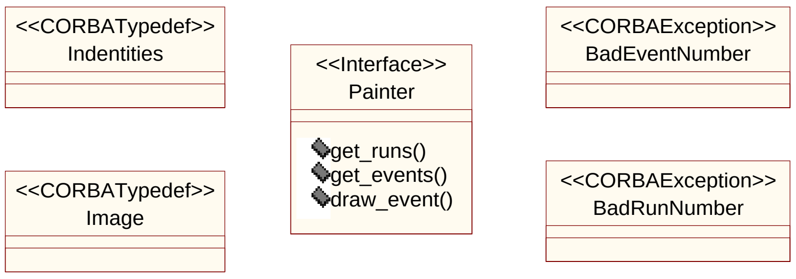


Fig. 13 Event Display Class diagram (UML notation)

The *Identities* is a type name for the sequence of the long integer values. It is used to specify a result for the *get_runs* and *get_events* methods. The *Image* type is defined as a sequence of bytes and is intended to be a return type of the *draw_event* method. The *BadEventNumber* and *BadRunNumber* are utilized to indicate the exceptional conditions for the Painter's methods execution. *BadEventNumber* can be raised by the *draw_event* method and *BadRunNumber* by both *get_events* and *draw_events* methods. All this information has been attached to the respective classes with the help of the Rose class diagram editor. This information is required to support the automatic IDL file generation.

4.4 OMG IDL declaration

Figure 14 shows the IDL declaration generated by Rose from the class diagram shown earlier.

```
module Event {
   typedef sequence<octet> Image;
   typedef sequence<long> Identities;

   exception BadRunNumber {};
   exception BadEventNumber {};

   interface Painter {
      Identities get_runs();
      Identities get_events(in long run_id) raises (BadRunNumber);
      Image draw_event(in long run, in long event, in long width, in long
                         height) raises (BadRunNumber, BadEventNumber);
   };
};
```

Fig. 14 Event Display interface (IDL)

It is necessary to run a specific IDL compiler to produce a *stub* and *skeleton* code from this IDL declaration. Since it was decided to use different languages for the Event Painter and Event Visualizer implementation different IDL compilers have to be used. The *idltojava* of Sun provides a Java *stubs* for the Event Visualizer and *idl* compiler by OOC has been used to generate a C++ *skeleton* for the Event Painter.

4.5 Event Painter implementation

4.5.1 Event Painter interface implementation

Figure 15 shows how to declare the C++ implementation class for the Event Painter interface. The Event_Painter_skel class that is a descendant of the Event_Painter_impl has been generated by the IDL compiler.

```
class Event_Painter_impl: public Event_Painter_skel
{
public:

   virtual Event_Identities* get_runs() {};
   virtual Event_Identities* get_events(CORBA_Long run_number) {};
   virtual Event_Image* draw_event(CORBA_Long run,
                                     CORBA_Long event,
                                     CORBA_Long width,
                                     CORBA_Long height);
};
```

Fig. 15 Declaration of the class that implements Event Painter (C++ language)

The virtual methods declared in the *Event_Painter_impl* class are inherited from the *Event_Painter_skel* class where they are defined as pure virtual methods. In order to provide an implementation for the Event Painter it is necessary to implement all these methods. Figure 16 shows how this implementation can be done. The functions *next_run* in line 8 and the *next_event* in line 21 represent a virtual API to the event storage system. For the actual implementation they shall be replaced with the adequate real data storage API calls.

```
1:   Event_Identities * Event_Painter_impl::get_runs()
2:   {
3:   // create the result sequence
4:      Event_Identities list = new Event_Identities;
5:
6:   // retrieve run numbers from the event storage system
7:      unsigned long run_number;
8:      while((run_number = next_run())!= -1)
9:         list.add(run_number);
10:
11:     return list;
12:  };
13:
14:  Event_Identities * Event_Painter_impl::get_events(CORBA_Long run_number)
15:  {
16:  // create the result sequence
```

```
    Event_Identities list = new Event_Identities;

// retrieve event numbers from the event storage system
    unsigned long event_number;
    while((event_number = next_event(run_number))!= -1)
       list.add(event_number);

    return list;
};

Event_Image * Event_Painter_impl::draw_event(
       CORBA_Long run, CORBA_Long event, CORBA_Long width, CORBA_Long height
)
{
    unsigned char * image_bytes;
    unsigned long image_size;
// prepare the image in memory
    .....
// create a bytes sequence to pass back to the Visualizer
    Event_Image * image = new Event_Image(image_size, image_size, image_bytes,
                                true);
    return image;
}
```

Fig. 16 Event Painter interface implementation (C++ language)

4.5.2 Event Painter task implementation

Since we have implemented the Event Painter interface there is only one thing that remains to be done - the EventPainter_impl class instance has to be created and registered with the ORB. Figure 17 shows how this can be done.

```
int main(int argc, char* argv[], char*[])
{
     try
    {
    // Create ORB and BOA
       CORBA_ORB_var orb = CORBA_ORB_init(argc, argv);
       CORBA_BOA_var boa = orb -> BOA_init(argc, argv);

    // Create implementation object
       Event_Painter_var p = new Event_Painter_impl();

    // Print stringified object reference to the standard output
       CORBA_String_var s = orb -> object_to_string(p);
       cout << s << endl << flush;

    // Run implementation
       boa -> impl_is_ready(CORBA_ImplementationDef::_nil());
    }
    catch(CORBA_SystemException& ex)
    {
       OBPrintException(ex);
       return 1;
    }
    return 0;
}

```

Fig. 17 Event Painter task implementation

The ORB initialization is done in line 6. The *CORBA_ORB_init* method creates an instance of the CORBA_ORB class that encapsulates all the methods of the ORB Interface that has been described in section 2.2.5. In the next line the instance of the Basic Object Adapter is created. The servant registration with the BOA instance is done implicitly during the construction of the *Event_Painter_impl* object. The BOA instance is used later in line 17 to call *impl_is_ready* method that is responsible for the acceptance of the external requests. The ORB instance is used to call the *object_to_string* method to convert the Event Painter object reference to string. This string is printed to the standard output stream and is intended to be used by the Event Visualizer for the access to the Event Painter.

4.6 Event Visualizer implementation

In order to deal with the interface defined for the Event Painter no additional code is required. The simple Java client application that uses directly the classes generated by the IDL compiler is shown in Figure 18. This code contains just the essential information and does not show all the aspects of the Java interface creation.

```
public class EventVisualizer extends JFrame{
   static org.omg.CORBA.ORB orb;
   static JLabel label;

   public EventVisualizer () {
   // create all the necessary graphical components here
      ....
   // this Jlabel will be used to display event image
      getContentPane().add(label = new JLabel());
   }

   public static void main(String args[]){
   // create and show application’s main frame
      EventVisualizer client = new EventVisualizer();
      client.show();

   // initialize ORB
      orb = org.omg.CORBA.ORB.init((String[])null, null);
   // convert parameter string to the Event Painter reference
      org.omg.CORBA.Object obj = orb.string_to_object(args[0]);
      Event.Painter ed = Event.PainterHelper.narrow(obj);

   // try to draw an event
      try{
         // call event_draw method for the event ‘m’ of the run ‘n’
         byte[] data = ed.draw_event(run_id, event_id, frame.getWidth(),
                             frame.getHeight());
      // display image (GIF and JPEG images can be displayed)
         label.setIcon(new ImageIcon(data));
      }
      // catch bad run number exception
      catch(Event.BadRunNumber ex){
         System.err.println("Bad Run number is used.");
      }
      // catch bad event number exception
      catch(Event.BadEventNumber ex){
         System.err.println("Bad Event number is used.");
      }
   }
}

```

Fig. 18 Event Visualizer implementation (Java)

The call to the *init* method of the org.omg.CORBA.ORB class in line 18 returns the ORB instance that can be used to call the *string_to_object* method. This method takes the first command line argument passed to the Visualizer application and tries to convert it to the Event Painter object reference. The *string_to_object* method always returns a reference to a generic CORBA object so it is necessary to cast this reference to a specific type that is *Event.Painter* in this case. It is done in line 21 via the *narrow* method. Then assuming that we know the run (*run_id* variable) and event (*event_id* variable) numbers we can display the event. In order to perform this the *draw_event* method is called in line 26. If the run and event number are valid the event image can be displayed by calling *setIcon* method of the *javax.swing.JLabel* class as it is done in line 28. For simplicity the requests for the list of runs and list of events are not shown here, but they should be done in the same way as the *draw_event* request and valid run and event numbers shall be presented to a user in order to give him a possibility to chose which ones he is interesting in. For example the Visualizer application shown on the Figure 17 uses the *javax.swing.JTree* class to represent this information.

4.7 Running applications

The Event Painter can be started by issuing the following command in the Unix shell (assuming that the executable name is **event_painter**):

```
prompt> event_painter > Reference.file
```

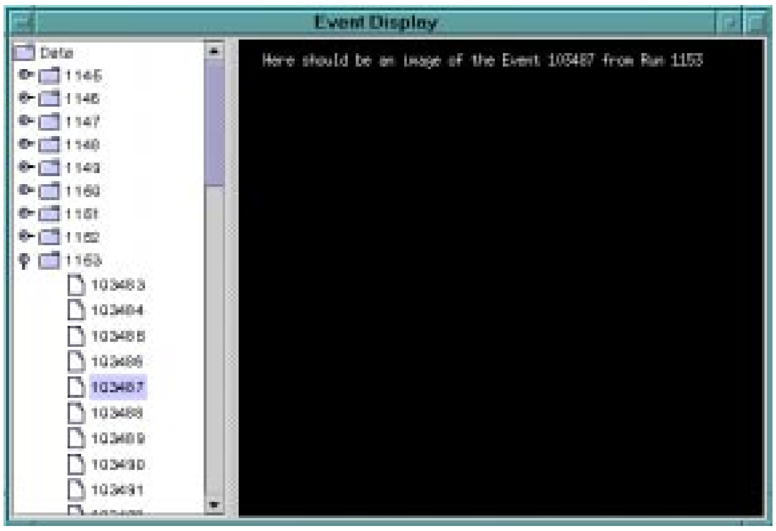
This command starts the Painter application that prints the *Event_Painter_impl* objects reference to the 'Reference.file' file. This reference shall be used as a parameter for the Event Visualizer in order to let him access the Painter object.

```
prompt> java EventVisualizer 'cat Reference.file'
```

The Visualizer application can be started on any machine - it is not necessary to run it on the same one on which the Painter is working. The only thing to worry about is the availability of the Reference.file file on that machine.

There is another way to establish connection between Event Visualizer and Event Painter without using intermediate file for the object reference storage. The CORBA Naming Service described in section 3.1.1 can be used to publish the Event Painter's object reference by associating it with some well known name. The Event Visualizer has to call the resolve method of Naming Service interface with this name as parameter in order to get the Painter reference.

The Figure 19 shows how Event Visualizer application looks like. It uses a tree to represent all the possible run numbers and events that belong to these runs. When the user selects an event number in the tree the *draw_event* method of the Event Painter is called and the image on the right side of the window is updated. The image shown by the Event Visualizer on the Figure 19 it created by the very simple Event Painter application. It does not paint a real event. It simply draws an image with the indication of the requested event and run numbers in order to illustrates the capacity for work of the proposed approach.



Here should be an image of the Event 103487 from Run 1153

Fig. 19 Event Visualizer main window

4.8 Remarks about Java RMI

The Event Display example has been shown might be implemented with another distributed technology, for example with Java Remote Method Invocation (RMI)[33]. The Java RMI is an excellent communication technology in the Java language domain. Other languages can involved only via the Java Native Interface (JNI)[34] that is currently available for C and C++ only. For example the Event Painter task might be implemented in Java using JNI for the data storage access. But JNI is another technology to be learned and another layer to be added to an application increasing the overall system complexity. It worth thinking about using a CORBA broker with the natural C++ language mapping instead of calling the necessary C++ methods via the JNI.

What else should be taken into account while choosing the implementation technology is a legacy issue: it is not certain in 10 years we will still use Java. CORBA in contrary to Java is a standard that does not relay to a particular programming language and is able to assimilate a new languages as it has been done already with Java.

Nevertheless Java RMI fits perfectly to the modern object design patterns. It is more simple to learn and to use because the role of the Interface Definition Language is played by the Java language itself. The only serious drawback is the number of operating systems for which Java is available. But this limitation was partly resolved recently by the common efforts of Sun and OMG. Sun has implemented the RMI interface over the IIOP protocol[35] and OMG defines the mappings from Java language to the OMG IDL[36]. It is possible now to connect a client implemented with RMI and a server developed with CORBA and vice versa.

5. APPLICATIONS OF CORBA IN THE HEP ENVIRONMENT

Ordinarily, CORBA brokers are not used for the implementation of the on-line software that is responsible for the fast physical data transportation. The reason is the overhead of the ORB over pure network protocol communication (UDP or TCP/IP). This overhead is introduced by automatically generated stub and skeleton code that is executed for each remote method invocation and by another level of the communication protocol (generally IIOP) that is used by most of the ORBs for the interoperability reasons.

But for the control systems and off-line data access the CORBA implementations have been started to be used recently. Here there are a few references to the large HEP experiments in which CORBA brokers have been used.

5.1 ATLAS[37] Trigger/DAQ prototype -1 at CERN (Geneva)

The goal of the TDAQ Prototype -1 project[38] is to produce a prototype system representing a “full slice” of a DAQ suitable for evaluating candidate technologies and architectures for the final ATLAS DAQ system on the LHC accelerator at CERN. The back-end DAQ component of the project encompasses the software to configure, control and monitor the DAQ but excludes the processing and transportation of physics data. All the communications in back-end subsystem are implemented on top of Inter Language Unification (ILU) system. ILU is implemented by Xerox and can be thought of as a CORBA ORB system (though with omissions from and extensions to the CORBA specification). The detailed description of how CORBA is used for the ATLAS TDAQ prototype -1 can be found in [39].

5.2 Textor[40] plasma-physics experiment at Plasmaphysics Institute (Julich)

In this experiment CORBA has been used to implement an interface to the distributed database providing data access over Internet. This database contains the measurements data for the Textor-94 experiment and the current system is using the Objectivity database. More details can be found in [41].

5.3 PHENIX[42] on-line control system (Brookhaven National Laboratory)

The PHENIX detector at the Relativistic Heavy Ion Collider (RHIC) will study the dynamics of ultra-relativistic heavy ion collisions and search for exotic states of matter, most notably the Quark Gluon Plasma (QGP). The PHENIX online control system is responsible for the configuration, control and monitoring of the PHENIX detector data acquisition system and ancillary control hardware. The online system consists of a large number of embedded commercial and custom processors as well as custom software processes which are involved in the collection, monitoring and control of the detector and the event data. These processing elements are distributed over a diverse set of computing platforms including VME based Power PC controllers, Pentium based NT systems, and SUN Solaris SPARC processors. The IONA Technologies Orbix CORBA broker has been used as the communication mechanism for the PHENIX online system [43].

5.4 BaBar configuration databases (Stanford Linear Accelerator Center)

The BaBar[44] experiment at the Stanford Linear Accelerator Center is designed to study the CP violation in decays of B mesons produced in electron-positron interactions. BaBar has chosen an Object Oriented Database Management System, Objectivity/DB, as the underlying storage technology. The online system has also adopted Objectivity to store the ambient data and the configuring parameters of various hardware and software components of the detector. To provide access to the ambient data before and after they are stored in the database a CORBA interface has been developed and implemented[45]. It allows Java based browsers to analyze and display the data while they are being accumulated.

6. CONCLUSION

The OMG was founded in 1989 by 11 companies. Now it is composed of more then 900 members, among of which there are most of the leading software development companies. The first version of the CORBA standard (1.0) was issued in 1991. It included mostly the IDL definition and C language mapping. After that a new revision of the standard appeared almost each year. Based on the CORBA users’ feedback all these revisions included important improvements like the interoperability architecture and Java mapping in the CORBA 2.0, POA specification in the CORBA 2.2, Java to IDL mapping in the CORBA 2.3. OMG has invested essential efforts to the integration with the other communication standards like COM/OLE and Java RMI.

All these efforts, have been invested by the OMG to the CORBA standard, result in an incredibly large number of ORB implementations. There are many good quality ORBs available now including free and commercial ones with a very wide range of operating systems and programming languages supported. The 10 years of CORBA evolution give an impressive example of a good quality standard development and maintenance. At the moment the CORBA standard is recognized as a very powerful and useful object communication model by the programming community and it looks very likely that it will carry on this leading role in the software communication domain.

In the future plans of the OMG the most important issues are the Internet integration, the quality of service control support and CORBA component model development. More information about these categories can be found at the OMG announces Web page (http://sisyphus.omg.org/technology/corba/corba3releaseinfo.htm).

7. REFERENCES

[1] Unix Network Programming: Networking APIs: Sockets and Xti, W. Richard Stevens.
[2] Power Programming with RPC, John Bloomer, published by O'Reilly, 1992.
[3] Distributed Computing Environment homepage http://www.osf.org/dce/
[4] Open Software Foundation homepage http://www.osf.org/
[5] The Object Management Group official home page is http://www.omg.org/
[6] Object-oriented Analysis and Design with Applications, Grady Booch.
[7] Rose is a visual modeling tool of Rational Software, http://www.rational.com/products/rose/index.jtmpl.
[8] Software trough Pictures is a visual modeling framework of Aonix, http://www.aonix.com/content/products.html#stp
[9] Together is a Java, full UML modeler for Simultaneous Design-and-Code Editing of Togethersoft, http://www.togethersoft.com/together/matrix.html
[10] Advanced C++ Programming Styles and Idioms, James O. Coplien.
[11] CORBA/IIOP 2.3.1 Specification, chapter 3-IDL Syntax and Semantics, http://cgi.omg.org/cgi-bin/doc?formal/99-07-07
[12] CORBA Language Mapping Specifications Available Electronically, http://www.omg.org/technology/documents/formal/corba_language_mapping_specifica.htm
[13] CORBA/IIOP 2.3.1 Specification, http://www.omg.org/technology/documents/formal/corba_2.htm
[14] OMG IDL to Java Language Mapping, Formal/99-07-53, http://www.omg.org/technology/documents/formal/omg_idl_to_java_language_mapping.htm
[15] The Document company XEROX, http://www.parc.xerox.com/parc-go.html
[16] Inter-Language Unification, ftp://ftp.parc.xerox.com/pub/ilu/ilu.html
[17] CORBA/IIOP 2.3.1 Specification, chapter 7-Dynamic Invocation Interface, http://cgi.omg.org/cgi-bin/doc?formal/99-07-11
[18] CORBA/IIOP 2.3.1 Specification, chapter 8-Dynamic Skeleton Interface, http://cgi.omg.org/cgi-bin/doc?formal/99-07-12
[19] CORBA/IIOP 2.3.1 Specification, chapter 11-Portable Object Adapter, http://cgi.omg.org/cgi-bin/doc?formal/99-07-15
[20] CORBA/IIOP 2.3.1 Specification, chapter 4-ORB Interface, http://cgi.omg.org/cgi-bin/doc?formal/99-07-08
[21] CORBA/IIOP 2.3.1 Specification, chapter 12 - Interoperability Overview, http://cgi.omg.org/cgi-bin/doc?formal/99-07-16
[22] CORBA/IIOP 2.3.1 Specification, chapter 15 General Inter-ORB Protocol, http://cgi.omg.org/cgi-bin/doc?formal/99-10-11
[23] The complete Discussion of the OMA, formal/00-06-41, http://www.omg.org/technology/documents/formal/object_management_architecture.htm
[24] CORBA Common Facilities Specifications, http://www.omg.org/technology/documents/formal/corba_common_facilities_specific.htm
[25] CORBA Services, OMG formal documents, http://www.omg.org/technology/documents/formal/corba_services_available_electro.htm
[26] Naming Service, version 1.0, http://www.omg.org/technology/documents/formal/naming_service.htm

[27] The Java IDL tutorial, http://java.sun.com/docs/books/tutorial/idl/index.html
[28] Java 2 SDK, Standard Edition Documentation, http://www.javasoft.com/products/jdk/1.2/docs/index.html
[29] ORBacus for C++ and Java, http://www.ooc.com/products/orbacus.html
[30] Object Oriented Concepts, Inc., Canada, http://www.ooc.com/
[31] A freely distributed tool for converting IDL interface definitions to Java stub and skeleton files, available at http://java.sun.com/products/jdk/1.2/docs/guide/idl/index.html
[32] Extensible Markup Language (XML) 1.0, http://www.w3.org/TR/1998/REC-xml-19980210
[33] Java Remote Method Invocation, http://java.sun.com/products/jdk/rmi/index.html
[34] Java Native Interface, http://java.sun.com/j2se/1.3/docs/guide/jni/index.html
[35] RMI over IIOP 1.0.1, http://www.javasoft.com/products/rmi-iiop/index.html
[36] Java Language Mapping to OMG IDL, http://www.omg.org/technology/documents/formal/java_language_mapping_to_omg_idl.htm
[37] ATLAS homepage http://atlasinfo.cern.ch:80/Atlas/Welcome.html
[38] ATLAS Trigger/DAQ Prototype-1 homepage http://atddoc.cern.ch/Atlas/
[39] Applications of Corba in the Atlas prototype DAQ, S. Kolos, R. Jones. L.Mapelli, Y. Ryabov, 11th IEEE NPSS Real Time Conference Proceedings, 1999, pp 469-474
[40] Textor-94 experiment homepage is http://www.fz-juelich.de/ipp
[41] Objectivity / Corba distributed database performance on gigabit SUN-Ultra-10 cluster, L.Gommans and others, 11th IEEE NPSS Real Time Conference Proceedings, 1999, 442-445
[42] Overview of PHENIX Online System, C.Witzig, 10th IEEE Real Time Conference Proceedings, 1998, pp 541-543
[43] Use of CORBA in the PHENIX Distributed Online Computing System, E.Desmond and others, 11th IEEE NPSS Real Time Conference Proceedings, 1999, pp 487-491
[44] BaBar homepage http://www.slac.stanford.edu/BFROOT/
[45] Ambient and Configuration Databases for the BaBar Online System, G. Zioulas and others, 11th IEEE NPSS Real Time Conference Proceedings, 1999, pp 548-550