

# CORBA and Object Services

Jan Kleindienst<sup>2</sup>, František Plášil<sup>1,2</sup>, Petr Tůma<sup>1</sup>

<sup>1</sup>*Charles University, Faculty of Mathematics and Physics,  
Department of Software Engineering  
Malostranské náměstí 25, 118 00 Prague 1, Czech Republic  
e-mail: {plasil, tuma}@kki.ms.mff.cuni.cz*

<sup>2</sup>*Institute of Computer Science, Czech Academy of Sciences  
Pod vodárenskou věží, 180 00 Prague, Czech Republic  
e-mail: {kleindie, plasil}@uivt.cas.cz*

**Abstract.** The paper provides an overview of the basic concepts of OMG CORBA. It summarizes the main ideas behind the OMG Reference Model Architecture and the components it defines. Particular attention is paid to the Object Request Broker and Object Services. An evolving example is used to illustrate the CORBA philosophy. The authors draw on their long-term practical experience with CORBA, particularly with implementing the Persistent Object Service [KPT96].

## 1 About OMG

The Object Management Group, Inc. (OMG), founded in 1989, is an international organization grouping system vendors, software developers, and end-users. OMG is currently supported by more than 500 members, including market giants such as DEC, HP, IBM, Microsoft, and SUN. The main goal of OMG is to promote object-oriented technology by defining "a living, evolving standard with realized parts, so that application developers can deliver their applications with off-the-shelf components for commons facilities like object storage, class structure, peripheral interface, user interface, etc." [OMG95c]. To achieve this goal, OMG has defined the Object Management Architecture (OMA) characterized by definition of two models: the *Core Object Model* (and its extensions called *profiles*) and the *OMA Reference Model*. An application conforming to the OMA Reference Model implicitly meets the key OMG objectives of *portability*, *interoperability*, and *reusability* that are considered a "holy grail" of the software industry.

## 1.1 OMG Structure and Technology Adoption Process

The heart of OMG is its board of directors (BOD) that votes upon all documents to be adopted by OMG. Technical issues are dealt with by the Technical Committee (TC). The TC guarantees that the proposed software components comply with the spirit of the OMA Reference Model in which case it recommends them to the BOD. The TC spans several working groups, e.g. Task Forces (TFs), Special Interest Groups (SIGs), and subcommittees. A TF is responsible for generating Requests for Information (RFIs), and, subsequently, Requests for Proposals (RFPs). A RFI should gather information and suggestions on a particular technical issue both from the OMG and non-OMG software communities. A RFI encourages proposals for solving the technical issue that it addresses. Typically, proposals are not adopted on the first attempt, but they must undergo several revisions. To ensure that proposals are feasible and backed up by the respondent's intention to make them commercially available, each proposal must contain a "proof of concept" section describing the steps already taken by the submitter towards implementation. Proposals are then usually combined into a joint submission(s). At this stage, the TF recommends this submission(s) to the TC which consequently votes upon recommending it to the BOD. Through its approval by the BOD, the proposed technology is officially adopted by OMG. The time of the adoption phase may vary from six months to several years.

## 1.2 OMG Object Model and OMA Reference Model

OMG's understanding of the object-oriented paradigm is reflected in the Core 92 Object Model (Core92) [OMG95c]. Core92 defines such concepts as object, inheritance, subtyping, operations, signatures, etc. Additional concepts can be added to Core92 to create an extension (*component*). A component should not replace, duplicate, and remove concepts. Components should be orthogonal to each other. A *profile* is a combination of Core92 and one or more components. Typical examples of profiles are the CORBA profile (Section 2), the Common Object Model, the ODP Reference Model, and the (proposed) Core 95 Object Model [OMG95e]. With the intention to provide a broad object-oriented architectural framework for the development of object technology, OMG defined the OMA Reference Model [OMG95c], published in 1992. The OMA Reference Model is comprised of the following four components:

The *Object Request Broker* component serves as a means for delivering requests and responses among objects. It is a backbone of the OMA Reference Model, interconnecting all the remaining components.

The *Object Services* component provides a standardized functionality (defined in the form of object interfaces), e.g. for class and instance management, storage, integrity, security, query, and versioning.

The *Common Facilities* component uses Object Services and defines a collection of facilities (such as a common mail facility) that a group of applications is likely to have in common.

The *Application Objects* component is not standardized by OMG. Application Objects use Common Facilities and Object Services via the Object Request Broker.

### 1.3 CORBA Evolution

The Common Object Request Broker Architecture (CORBA) is a common framework that separates requesters of services (clients) from providers of services (object implementations) and, via ORB, allows for sending messages between them. The CORBA architecture is based on CORBA/OM derived from OMG/OM. CORBA/OM is a *concrete* object model that deals with the format of requests being delivered from clients to object implementations, operations upon objects, objects' interfaces and attributes, objects' data types, and objects' implementation.

CORBA was adopted by OMG in December 1991 as a joint submission of DEC, HP, etc. The first version was denoted as CORBA 1.1 and was, in 1994, replaced by CORBA 1.2 which brought more or less "cosmetic changes" with respect to CORBA 1.1. A much more important development step was CORBA 2.0, which was adopted in December 1994 and which overcomes the drawbacks of CORBA 1.2 by specifying ways for interconnecting different ORBs, and also by suggesting means for establishing interoperability between CORBA-based and non-CORBA-based environments, such as Microsoft's COM/OLE. Since the difference between CORBA 1.2 and 2.0 is quite dramatic, we will distinguish these two versions when necessary. Throughout the paper, wherever we use *CORBA* we mean CORBA 2.0 [OMG95a].

### 1.4 Common Facilities

Common Facilities (CF) provide a higher-level functionality than Object Services; informally, they are application oriented, while Object Services are "system oriented". A major expected advantage of Common Facilities (they are still in the stage of RFP) is that if properly designed, they may reduce the amount of code being written from scratch and become shared among CORBA-based applications. Common facilities, with interfaces defined in an object-oriented manner, are divided into two categories: *horizontal CF* and *vertical CF*. Horizontal CF are intended for cooperation among typical specialized applications, e.g. document editing, graphical user interface. To reflect this intention, they are divided into four groups: *User interface* (e.g. similar to those provided by OLE and OpenDoc), *information management* (e.g. Compound document storage, data interchange), *system management* (e.g. configuring, installing distributed object components), and *task management* (e.g. workflow, e-mail). Vertical CF provide object-oriented interfaces for vertical market segments (finance, insurance, etc.).

## 2 CORBA

### 2.1 Principles

According to the OMA Reference Model, the role of an ORB is to provide means for communication between objects and their clients. For the purposes of defining concepts specific to ORB functionality, CORBA has introduced the *CORBA Object Model* as an extension to the OMG Core Object Model.

The CORBA Object Model defines an *object* as an encapsulated entity that provides requestable services and a *client* as an entity that requests these services. It should, however, be noted, that being a client is not a static property - an entity is referred to as a client only at the moment when a service is requested. A *request* for a service is associated with a requested operation, a target object and, optionally, *parameters* and a *context*. As an outcome of the service, a result can be returned to the client. Should an abnormal condition occur, an *exception* can be returned, possibly with additional return parameters characteristic for the exception. Requestable services are available as *operations*. An operation is identified by its name and *operation signature*, consisting of a specification of the request parameters (types and information flow direction), a specification of the operation result, a specification of possible exceptions together with accompanying parameters, a specification of additional contextual information, and a specification of the execution semantics (either *at-most-once* or *best-effort*).

The set of operations requestable from an object is determined by its interfaces. An *interface* is a collection of operation signatures.

### 2.2 Interface Definition Language

Object interfaces are defined using the OMG Interface Definition Language (IDL) introduced in the CORBA specification [OMG95a]. The definition written in IDL completely specifies an interface and provides all information needed by the client to request services via the interface.

IDL is based on ANSI C++ grammar with several extensions. An IDL specification consists of one or more module, interface, exception, constant or type definitions.

Throughout the text, we will demonstrate several features of a CORBA environment through an evolving example. In this example, we introduce the interface *Book* providing the operation *Find()* for searching the text of the book and the readonly attribute *sName()* for retrieving the book's name. The example is based on the Orbix, a CORBA implementation done by IONA Technologies [ORBIXa, ORBIXb].

In our example, the IDL definition of a book's interface could be as follows:

```
interface Book {  
    readonly attribute string sName;
```

```
    unsigned short Find (in string sWhat);  
};
```

An interface of an object can also contain an *attribute*. This syntactical construct implies generating two operations, one for getting and the second one for setting the attribute's value; a readonly attribute implies generating only the operation for getting the attribute's value.

Being a purely descriptive language, IDL is not used to request the specified services. Instead, the IDL definitions are mapped to the client's programming language to be used by its standard constructs. CORBA 2.0 standardizes the IDL mapping to C, C++ and Smalltalk, standard mappings to other languages either already exist (e.g. Java), or are expected to follow (e.g. COBOL).

Suppose that in our example both the client and the implementation of the *Book* interface use C++. In the C++ environment, the *Book* interface is mapped to a `Book` class with corresponding methods:

```
class Book: public virtual CORBA::Object {  
public:  
    virtual char *sName (ENV env);  
    virtual unsigned short Find (const char *sWhat, ENV env);  
};
```

The parameter `env` servers mainly for registering exceptions.

## 2.3 The Structure of a CORBA ORB

The structure of a CORBA ORB is depicted on Fig. 1. The OMG standard defines the structure in terms of interfaces and their semantics. In an actual implementation, however, there is no obligation to reflect the structure; it is only necessary to preserve the interfaces and their semantics.

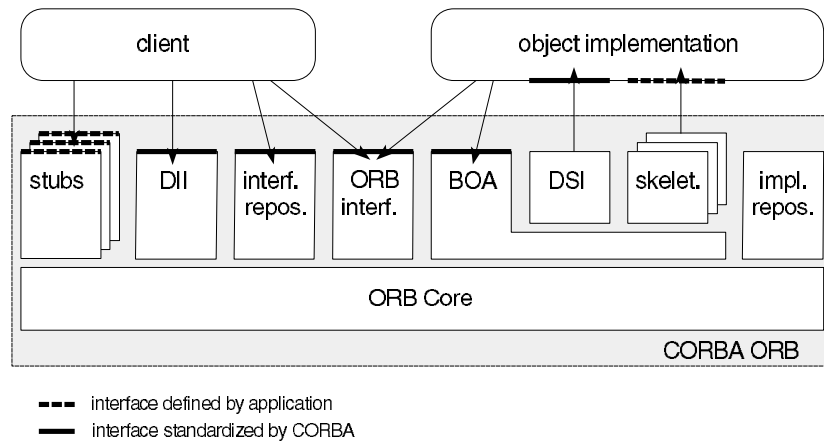
### 2.3.1 Client Side (Stubs, DII, Interface Repository)

To request a service from an object, the client can use either static or dynamic invocation (Fig. 2). The static invocation mechanism requires the client to know the IDL definition of the requested service at compile time. The definition is used to automatically generate *stubs* for all requestable operations. To request an operation, the client calls the appropriate stub which passes the request to the ORB Core for delivery.

As opposed to static invocation, the dynamic invocation mechanism requires the client to know the IDL definition of a requested service at run time. To request an operation, the client specifies the target object, the name of the operation, and the parameters of the operation via a series of calls to the standardized *Dynamic Invocation Interface*.

Again, the request is then passed to the ORB Core for delivery. The semantics of the operation remains the same regardless of the invocation mechanism used. To obtain an IDL interface definition at run time, the client can use the *Interface Repository*. The repository makes IDL definitions (e.g. module, interface, constant and type definitions) available in form of persistent objects accessible via standardized interfaces. The ORB is responsible for finding the target implementation, preparing it to receive the request, and communicating the data of the request. The client need not care about the location of the object, language of the implementation, or other things not described in the interface.

On the client side, a target object is usually represented as a proxy (object) supporting the same interface as the target object. In the case of static invocation of an operation *m*, the proxy's method *m* calls the stub associated with *m*; the stub is responsible for



**Figure 1** Structure of a CORBA ORB

creating the request and passing it to the ORB Core for delivery to the target object. In case of dynamic invocation, the client's code creates the request dynamically by calling the DII interface; in principle, no proxy and stub are necessary.

On the client side in our example, a book is represented by a proxy referenced via *MyBook*. Static invocation might take the form:

```
iLineNr = MyBook->Find ("keyword",env); // static invocation
```

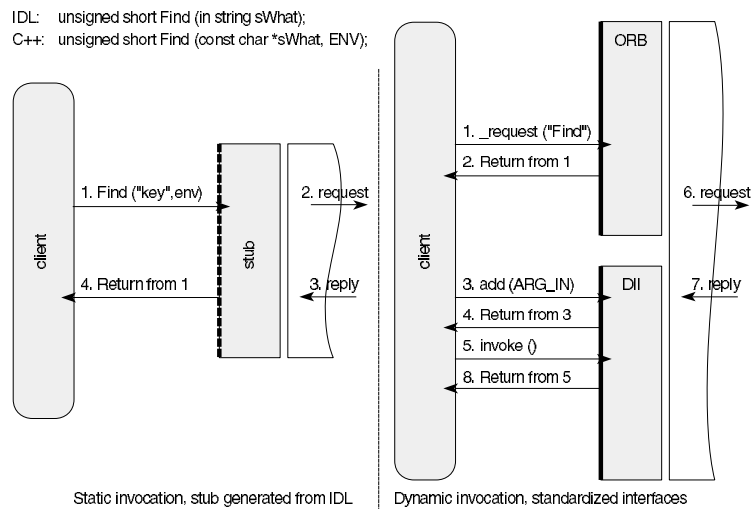
Dynamic invocation with the same effect could be:

```
Req = MyBook->_request ("Find"); // dynamic invocation
*(Req->arguments ()->add(ARG_IN)->
value ()) <<= "keyword";
Req->invoke ();
*(Req->result ()->value ()) >>= iLineNr;
```

Note that in Orbix, a proxy supports also the DII interface.

### 2.3.2 Implementation Side (Skeletons, DSI, Implementation Repository, Object Adapters)

Again, either the static or the dynamic mechanism can be used to convey the request to the actual service implementation (Fig. 3). The static mechanism requires the IDL definition of the requested service to be available at compile time. An automatically generated operation-specific *skeleton* is then used by the ORB to call the implementation of the requested service.



**Figure 2** Static and dynamic invocation on client side

The dynamic request delivery mechanism expects the implementation to conform to the standardized *Dynamic Skeleton Interface*. Through this interface, the ORB dispatches the request to the target object (to the service implementation). In analogy to the dynamic invocation mechanism on the client side, no compile time information about the interfaces is needed. The information necessary to locate and activate implementations of requested services is stored in the *Implementation Repository*. The implementation repository is specific to a particular ORB and environment and is not standardized by CORBA. To access services provided by the ORB itself, an implementation uses the *Object Adapter*. To satisfy the needs of diverse environments, the ORB can be equipped with several different object adapters. Each ORB, however, must provide the standardized *Basic Object Adapter (BOA)* interface. The operations of BOA include generation and interpretation of object references, authentication of clients, and activation and deactivation of target objects.

In the implementation, in our example, the static and dynamic request delivery mechanisms might take the forms:

```

class StaticBookImpl : public BookSkeleton {
public:
    char *sName (ENV) {return (sMyName);};
    unsigned short Find (const char *sWhat,ENV env)
        {return (FindInText (sMyText,sWhat));};
private:
    char *sMyName;
    char *sMyText;
};

class DynamicBookImpl : public DynamicImplementation {
public:
    void invoke (REQ Req, ENV env) {
        if (Req->op_name () == "Find") then {
            MyParams = new NVList ();
            *(MyParams->add(ARG_IN)->value ()) <=< (char *) NULL;
            Req->params (MyParams);
            Req->result
                (FindInText (sMyText,MyParams->item (0)->value ()) );
        } else ...
    }
// the rest omitted for sake of brevity

```

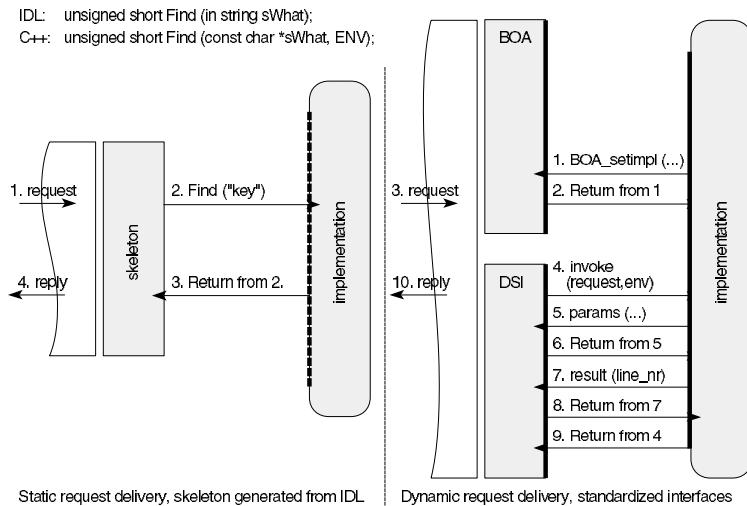
### 2.3.3 ORB Interface

For accessing the general services provided by an ORB, both client and implementation use the *ORB* interface. The operations of this interface include converting object references to strings and vice-versa, determining the object implementation and interface, duplicating and releasing copies of object references, testing object references for equivalence, and ORB initialization.

### 2.3.4 ORB Core, Interoperability

CORBA 1.1 left the implementation of the ORB Core totally to the ORB vendor. It was hence almost impossible to interconnect two or more CORBA-compliant ORBs released by different vendors. This lack is remedied by the CORBA 2.0 specification that defines two approaches for achieving interoperability: the *General Inter-ORB Protocol* (GIOP) and *Inter-ORB bridging*.

The General Inter-ORB Protocol specifies a set of low-level data representations and message formats for communication between ORBs. It is designed to be simple enough to work on top of any underlying connection-oriented transport protocol, such as TCP/IP and IPX/SPX. The GIOP based on TCP/IP is called the *Internet Inter-ORB Protocol* (IIOP). CORBA 2.0 requires GIOP as the mandatory protocol. To allow "out of the box" interoperability, CORBA 2.0 specifies the optional *Environment Specific Inter-ORB Protocol* (ESIOP) as an alternative to the GIOP. It has been mainly specified to allow integration of DCE [DCEwww] with CORBA (DCE/ESIOP).



**Figure 3** Static and dynamic request delivery on implementation side

The main idea of bridging lies in mapping requests from one vendor's format to another when crossing ORB boundaries. Bridging can be performed either at the ORB Core level as an ad hoc solution for every ORB<sub>i</sub>-ORB<sub>j</sub> pair (*in-line bridging*), or at the application level (*request-level bridging*). Request-level bridges mediate requests by utilizing DII and DSI mechanisms (Sections 2.3.1 and 2.3.2) to dynamically create and dispatch requests. Bridges are currently mainly used for connecting CORBA with non CORBA-compliant platforms, such as COM/OLE.

Compared to CORBA 1.0, by demanding either the presence of the IIOP protocol or a "half-bridge" that translates an ORB's native request format into the standardized IIOP request format and vice versa, CORBA 2.0 extends the requirements for fulfilling CORBA compliancy.

### 3 Object Services

The collection of Object Services is one of the fundamental components introduced by the OMA Reference Model. It provides functionality a CORBA-based application may use to increase its portability. One may consider it as a high-level library with standardized interfaces specified in IDL. The design and specification of Object Services have been guided and supervised by the Object Service Task Force (OSTF) that, during the years 1992-1996, issued five RFPs, each dealing with a different set of Object Services. The specification of the adopted Object Services was published by OMG in the manual entitled "CORBAservices: Common Object Services Specification" [OMG95f]. This is perhaps the reason why in some publications Object Services are also called *CORBA Object Services*.

### 3.1 Architecture and Design Principles

The key design principle, introduced by the OSTF [OMG95b] and denoted as the Bauhaus principle, reads: "*Minimize duplication of functionality. Functionality should belong to the most appropriate service. Each service should build on previous services when appropriate.*" This principle somewhat contradicts with another design principle stated in the same document: "*It should be possible to separately specify and implement each object service.*" In the paper [KPT96], we showed that complying with the both principles at the same time is not a trivial task.

### 3.2 Basic Set of Object Services

In the document [OMG95d] published in 1992, OSTF identified twenty-five Object Services and their mutual dependencies. The first four services were subject of RFP1 issued that year: the Naming, Event Notification, Lifecycle, and Persistence Services. Other RFPs followed, each dealing with a different subset of services: RFP2 - Concurrency, Externalization, Relationship, Transaction; RFP3 - Time, Security; RFP4 - Licensing, Properties, Query; RFP5 - Change Management, Collections, Trader. The document [OMG95b] also identifies other candidates that may, in the future, enrich the collection of the original twenty-five services, e.g. Narrowing, Index, or Recovery/Fault Management.

With respect to the limited size of this paper, we focus, in Sections 3.3 through 3.6, upon the Lifecycle, Events, Persistence, and Relationship Services, particularly as they are in our opinion of fundamental importance and also as we have practical experience with their implementation [KPT96].

### 3.3 Life Cycle Service

The Life Cycle Service provides its client with means to create, delete, copy, and move objects. Objects are created by *object factories*. These are simply other objects capable of creating and returning an object as a result of some sort of *create()* request. As the parameters required to create an object may vary among different object types, the factory interface is not standardized. A *generic factory* can be used to dispatch or coordinate calls to several object factories. Generic factories can be hierarchically organized. As it is not possible to dispatch calls to factories with potentially different proprietary interfaces (provided by different vendors), the generic factory can only dispatch calls to factories that inherit the standardized *GenericFactory* interface. Additional request parameters are passed in a form of *criteria* list, each parameter being stored as a named value.

An object is deleted by issuing a *remove()* request on its *LifeCycleObject* interface. Objects can make themselves unremovable; this property is signaled by returning an appropriate exception as a result of the *remove()* request. An object can be moved or copied to another location using the *move()* or *copy()* requests of its *LifeCycleObject* interface. To find a target location of the operation, the Life Cycle Service uses a

*factory finder*. By issuing the *find\_factories()* request, the client can ask a factory finder to return a set of factories capable of creating a new copy of the object in a target location. The specification does not standardize any mechanism for transferring the object state from one location to another.

### 3.4 Events Service

In some cases, synchronous communication provided by an ORB may not be suitable for an application. For such applications, the Events Service provides a decoupled inter-object communication model.

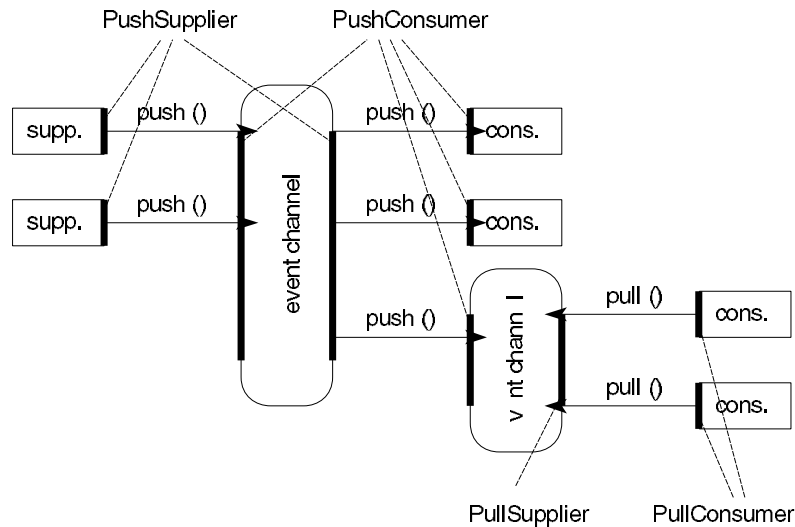
The Events Service defines the roles of event *suppliers* and event *consumers* for objects which generate and process events, respectively. Two approaches are defined depending upon who is active in the communication process - in the *push* model, a supplier calls its consumer to deliver data, whereas in the *pull* model, a consumer calls its supplier to request data. Although in principle possible, neither supplier nor consumer is expected to call the other communicating object directly. Instead, all events are passed through an *event channel*. From the supplier's point of view, the event channel acts as a consumer; from the consumer's point of view, the event channel acts as a supplier. The event channel also provides operations to register both suppliers and consumers. Subject to the quality of service, the event channel can provide one-to-one, one-to-many, or many-to-many communication, and other additional features.

Communication among objects can be either *generic* or *typed*. Generic communication relies on suppliers and consumers having a standardized interface capable of passing an object of the class *any* as event data. With typed communication, suppliers and consumers are expected to agree on a proprietary one-way operation to pass event data in a suitable format.

### 3.5 Persistent Object Service

In a slight discrepancy with its name, one of the goals of the Persistent Object Service (POS) specification was to avoid defining the internals of the Persistent Object Service as much as possible. Therefore, the POS specification aims at providing a common interface to define a general enough framework for all potential persistent service implementations without assuming much about their internal functionality.

The core of the service functionality is encapsulated within a component called the *Persistent Data Service* (PDS). A PDS is responsible for carrying out the basic persistent operations, such as *store()* and *restore()*, on persistent objects. These operations hide all communication between a PDS, persistent objects and, a *Datastore* behind a standardized interface. Internally, a PDS communicates with persistent objects via a proprietary *protocol*. Another proprietary mechanism is used to communicate with a Datastore. While neither of these two mechanisms is standardized, the specification contains examples of both the protocol and the Datastore interface.



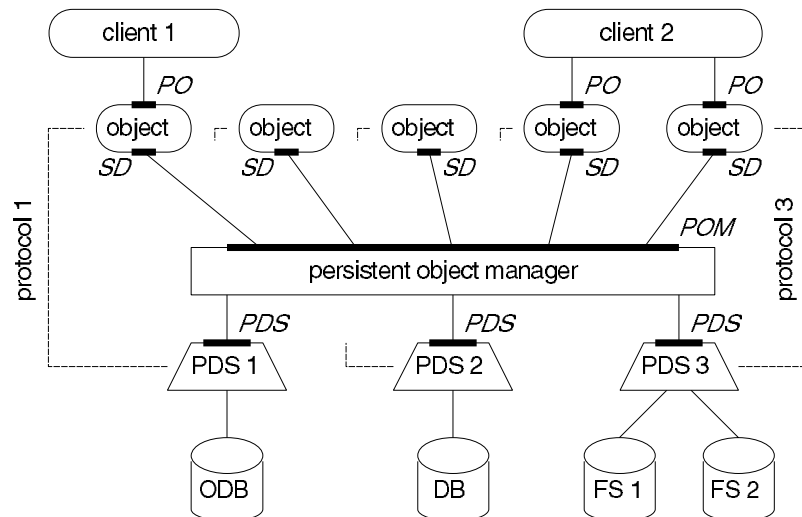
**Figure 4** An example of an Events Service application

A *Persistent Identifier* (PID) is used to describe the location of an object's persistent data in a Datastore. With the exception of the mandatory string attribute specifying a target Datastore type, the PID structure is not standardized. To facilitate use of several PDSs with possibly different protocols and underlying Datastores, the POS specification introduces the notion of the *Persistent Object Manager* (POM). The POM has a standardized interface similar to that of a PDS; its role is to dispatch incoming requests to the appropriate PDS. The behavior of the dispatch mechanism is not standardized.

The POS specification demands very little of persistent objects themselves. Apart from the obvious need to support an appropriate protocol, an object may inherit the standardized *SynchronizedData* interface. Through this interface, the POM notifies the object of operations on its persistent state. To allow external control of its persistency, a persistent object may also inherit the standardized *PersistentObject* interface, similar in functionality to both the POM and PDS interfaces.

### 3.6 Relationship Service

The Relationship Service serves as a tool for expressing inter-object relations in a similar way as E-R diagrams do. The service specification is divided into three levels. Level one concentrates on support for describing relationships among "regular" objects (called here related objects); level two provides support for manipulating graphs of related objects, and finally level three adds support for commonly used special cases of relationships.



**Figure 5** An example of a Persistent Object Service architecture

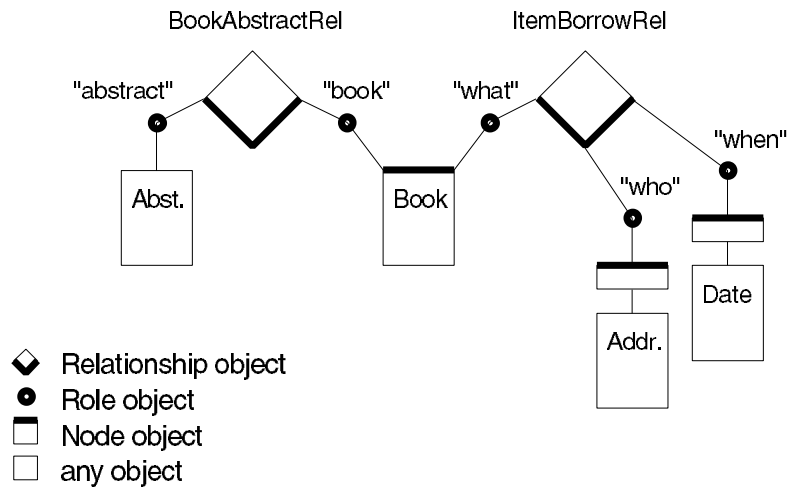
A related object participates in a relationship indirectly, via a *Role* object (role for short). While a role is associated with exactly one related object, the related object can have several roles representing it in relationship(s). A role can put constraints on the type of the related object it is associated with and on the number of relationships it is participating in. A relationship among roles is represented by a *Relationship* object. A relationship is characterized by its degree (arity) and the names of the participating roles; additional constraints can limit the types of roles that can participate in the relationship. Once created, a relationship is immutable. Both the *Role* and *Relationship* interfaces are standardized. A role can be queried for its associated object, a list of relationships in which it participates, and a list of related roles and objects. It is also possible to destroy all relationships in which a particular role participates. Similarly, the *Relationship* interface can be used to get a list of participating roles.

Related objects can be involved in a graph. The basic building block of a graph is *Node* object. Either a Node object can be associated with its related object via reference, or the related object can inherit the *Node* interface. Unlike related objects in general, a node is aware of its roles and can be queried to return a list of them. Furthermore, nodes impose additional constraints on the type of the associated roles.

Given a starting node, a graph can be traversed according to a given *Traversal* object. A *Traversal* object produces a sequence of *directed edges* of a graph. A directed edge indicates whether the traversal should continue to an adjacent node. The traversal object can use three different strategies to traverse a graph, i.e. breadth first, depth first

and best first. To determine the relevant nodes and edges, *Traversal* object queries a user-supplied *TraversalCriteria* object. Given a node, traversal criteria tells the *Traversal* object which of the associated relationships and related objects to visit. The specification also introduces the concept of *propagation value* to be used in conjunction with standardized traversal criteria in situations where it is not suitable or feasible to use a *TraversalCriteria* object provided by the client.

In many applications, one-to-many containment relations and one-to-one reference relations will be used. A definition of roles and relationships specialized to form these relations is included in level three of the Relationship Service specification.



**Figure 6** An example of a Relationship Service application

## 4 Relation to Other Standards and Environments

### 4.1 Relation to DCE

In this section, we identify the similarities of the both CORBA and DCE platforms, and, on the other hand, emphasize their differences.

#### **4.1.1 Similarities**

Promoted by the OSF organization, Distributed Computing Environment (DCE) [DCEwww] is similar to CORBA in its main goal - providing means for building distributed applications. Both environments export server functionality via an architecture-neutral collection of IDL interfaces. Both come with an IDL compiler that generates stubs and skeletons responsible for marshalling and unmarshalling data from the IDL source. Both platforms provide communication transparency and for both platforms the semantics of local calls differs from that of remote calls, since the latter do not have to be completed due to a network failure.

#### **4.1.2 Differences**

DCE is procedure-oriented (neither object-oriented nor object-based). DCE's IDL is based on C and does not support inheritance. However, the IDL compiler of DCE 1.2 should have an option of compiling IDL into C++ classes while supporting single inheritance. While CORBA allows the dynamic creation of requests via DII and DSI (Section 2.3.2), DCE does not provide such a feature.

The collection of Object Services, a fundamental component of OMA, furnishes CORBA with a powerful functionality. DCE does not have its functionality stratified into a similar set of services. On the other hand, many DCE facilities have their analogies in some of the OMG Object Services and vice versa. For example, OMG's Naming Service is an analogue of DCE's Directory Service that strictly defines the responsibility of the DCE cells for resolving local names via the Cell Directory Service (CDS) and global names via a collection of CDS servers. Another example might be the analogy of OMG's Concurrency Control Service with DCE's standardized POSIX thread implementation. OMG's Security Service is a good example showing how DCE's powerful Security Service providing both authentication and authorization inspired the CORBA designers during the process of proposing OMG's Security Service. Compared to DCE, the approach to security is generally believed to be the major disadvantage of the contemporary CORBA (OMG Security Service is still being worked on by the OMG Object Services TF). For some OMG services, such as the Transaction and the Event Notification Services, there is no analogy as DCE does not provide any facility for handling transactions or sending events.

#### **4.1.3 Issues of Building CORBA upon DCE**

At the time when the CORBA 2.0 specification was being prepared, most DCE supporters believed that the CORBA ORB Core would be built merely upon DCE, or that at least the DCE/ESIOP protocol would be required as a mandatory protocol. It did not happen. Instead, CORBA 2.0 ORBs are built upon the mandatory IIOP and the DCE/ESIOP protocol is only optional, even though the DCE/ESIOP protocol, compared to IIOP, includes advanced features such as Kerberos security, authenticated RPC, and an option to choose from connection-oriented or connectionless protocols. In our view, the main reason why the IIOP protocol was given a priority over

DCE/ESIOP is the OMG idea of using the Internet as a backbone for the ORB-to-ORB communication. The authors of our currently favorite book [OHE96] even claim that "*IIOP will eventually transform the Internet into a CORBA bus.*"

## **4.2 Relation to Java**

This section compares the CORBA and Java computational models and the underlying communication protocols IIOP and HTTP, and illustrates the complementary character of both platforms.

### **4.2.1 Different Computational Models**

The main characteristic of the CORBA computational model is locating most of the available system functionality encapsulated in objects on the implementation (server) side. As a CORBA application can actually contain many CORBA objects distributed on various machines with possibly different hardware and operating systems, the CORBA computational model is characterized by *distributed objects*.

Java facilitates a different computational model. At runtime, all remote objects required by a Java application are downloaded to the client and dynamically bound to become part of the application. Thus, in general, the application grows in time if it downloads objects from the remote server(s). The Java Virtual Machine (JVM) provides a platform-independent environment in which the precompiled code of the Java application (*bytecode*), together with the downloaded bytecode of the remote objects used by this application, is interpreted. It is worth emphasizing that this computational model does not facilitate any remote invocation from the client to the implementation (server) side as is done in the CORBA computational model. In Java, only the bytecode of the objects is distributed, not the object state. Thus, the Java computational model is characterized by *distributed code*. A more detailed comparison of CORBA and Java can be found in [JKW96].

### **4.2.2 Complementary Character of both Platforms**

The above discussion shows that Java complements CORBA rather than competing with it. There are also serious attempts to implement the CORBA architecture using the Java language. As typical examples let us mention HORB [Hir96] and BlackWidow [BlackW]. Even though some implementations perform rather slowly, some current releases of JVM bytecode interpreters furnished with automatically invoked compilers (*just-in-time* compilers) allow for efficient implementations of CORBA in Java, thus drastically improving the odds against more common C++ implementations of CORBA. While taking advantage of the both platforms, a rational marriage of CORBA and Java may result in the advent of the new era of distributed computing.

To illustrate the complementary character of both platforms, we focus briefly on developing Java environments, such as OrbixWeb [OWeb], BlackWidow [BlackW], and Joe/NEO [NEO95]. Typically, in these environments a Java *applet*, i.e. a Java application running under a WWW browser, located on a client provides a lightweight

front end to an ORB running on a remote server. Downloadable by any Java-enabled WWW browser, the applet may interact with the user by taking advantage of the standard Java library with built-in multimedia capabilities, thus providing a visually appealing graphical interface. The communication between the applet and the ORB is done via a classical CORBA stub-request-skeleton mechanism, usually through the IIOP protocol. The presence of this additional communication link between the client and the server relieves the burden of the HTTP daemon running at the server, since the applet requests and the ORB responses are routed via IIOP. In case of OrbixWeb, the ORB called Orbix is implemented in C++. BlackWidow has both components implemented in Java. Joe is implemented in Java and the ORB called NEO in C++.

### 4.2.3 Java Security Restrictions

Unfortunately, there is one significant restriction as an outcome of the Java tight security policy: a Java applet can open a communication link only to the server from which it has been downloaded (although this feature is not built-in in the language). Applied to the communication triangle - applet, HTTP daemon, ORB, this means that if an applet has been downloaded via a HTTP daemon from an imaginary server `applet.source.com`, it can open an IIOP connection to this server only. An attempt by the applet to connect to any other server would result in a Java security violation exception. Hence the ORB must reside on the same machine that runs the HTTP daemon. This may become an unjustifiable restriction for some applications. One ad hoc (and rather hacky) solution to this problem is to modify the *SecurityManager* class in the Java standard library. This class implements the Java security policy. For example, in case of running the Netscape browser with an appropriately tampered *SecurityManager* class in its `moz3_0.zip` file containing the Java standard library, the applet downloaded from `applet.source.com` could open a connection not only to this server running the HTTP daemon but also to another one, called for example `corba.server.com`, running the ORB. It is expected that in the future the user will be allowed to give up this restriction via a cleaner mechanism, e.g. an environment variable similar to the CLASSPATH environment variable which contains the directories accessible from a Java application.

### 4.2.4 IIOP versus HTTP

The IIOP protocol is used by CORBA as a user-transparent TCP/IP-based protocol for delivering requests. The HTTP protocol [HTTPwww], also built upon TCP/IP, is used by WWW browsers for downloading HTML documents as well as Java applets from WWW servers. CORBA's IIOP protocol overcomes the limitations of HTTP in the following way: first by allowing a client applet to use a wide range of data types supported by IIOP as opposed to HTTP which does not standardize data types allowed for transmission. Second, the combined power of simple HTTP operations (downloading a HTML document, a picture, sound, and an applet) and CGI scripts (scripts performed on the WWW server whose output is redirected to the WWW client) is now fundamentally expanded by allowing the client to call objects residing on a server directly without starting a CGI process upon every request. Third, the HTTP protocol opens a socket connection for every client request. The IIOP protocol,

on the other hand, leaves the connection opened during the entire session, thus reducing the socket initialization overhead. However, the recently specified HTTP 1.1 protocol allows persistent connections.

## **5 On the near future of CORBA (instead of conclusion)**

In the paper, we have presented an overview of OMG's role, CORBA basic concepts and structure, Object Services, and CORBA's relations to other standards and environments (particularly DCE and Java). In this concluding section, we emphasize and illustrate the dynamics of CORBA's evolution and provide the reader with landmarks which, in our view, frame the picture of CORBA's near future. The current trends in the evolution are clearly reflected in the OMG Task Force names (and SIG names in some cases) and particularly in the RFPs issued by the TFs recently.

As its name suggests, the **ORB and Object Services Platform TF** is involved in evolution of ORB and Object Services. The ORB interface is being improved, IDL type extensions and object multiple interfaces are being discussed, secure IIOP RFP has been issued, and the COM/CORBA interoperability is being considered. Work upon the following Object Services is in progress: the Startup, Trader, Collection, Time, and, especially, Security Services. The **Analysis and Design Platform TF** issued RFP1 with the intention to define common Object Analysis and Design (OA&D) meta-models (static, behavior, usage, and architectural models) and IDL specifications for model interchange among OA&D tools.

A lot of current OMG attention is paid to Common Facilities which are now being split into two OMA components: Common Facilities (originally Horizontal Common Facilities) and Domain Interfaces (originally Vertical Common Facilities) [OMG96]. The **Common Facilities Platform TF** is concerned particularly with the Financial, Internationalization and Time, Data Interchange and Mobile Agent, Printing, and System Management Facilities. The **Domain Technology Committee** has appointed TFs for specifying domain interfaces, e.g. for the following fields: asset and content management, telecommunication, manufacturing, and business objects. A great deal of OMG effort is directed towards the integration of CORBA and the Internet. The **Internet Special Interest Group** has been appointed by OMG to foster this integration.

All in all, CORBA-related activity has accelerated enormously, and it is becoming hard to follow the field in its entirety. Just for illustration, during the first six months of 1996, OMG produced roughly 300 documents, mostly proposals and responses from major software companies. This fact indicates the magnitude of OMG's impact on the software industry.

## Acknowledgements

The authors of this paper would like to express their thanks to Adam Dingle for proofreading the text and for valuable comments on contents of the Section 4.

## References

- [Ben95] R. Ben-Nathar: CORBA: A guide to Common Object Request Broker Architecture. McGraw-Hill. 1995.
- [BlackW] BlackWidow, PostModern Computing. URL: <http://www.pomoco.com>
- [DCEwww] Object Software Foundation, URL: <http://www.omg.org>
- [Hir96] S. Hirano: HORB Home Page. Work in Progress at Japan's Electrotechnical Laboratory (ETL), URL: <http://ring.etl.go.jp/openlab/horb>
- [HP95] HP ORB Plus 2.0, URL: <http://www.hp.com>
- [HTTPwww] Basic HTTP, <http://www.w3.org/pub/WWW/Protocols/HTTP/HTTP2.html>
- [IBM94a] IBM Corp. SOMobjects Developer Toolkit Users Guide, Version 2.1, 1994.
- [IBM94b] IBM Corp. SOMobjects Developer Toolkit Programmers Reference Manual Version 2.1, 1994.
- [JKW96] J. Kiniry, A. Johnson, M. Weiss: Distributed Computing: Java, CORBA, and DCE, Version 1.2, URL: <http://www.osf.org>, Feb 1996
- [KPT95] J. Kleindienst, F. Plášil, P. Tůma: Implementing CORBA Persistence Service, TR 117, Charles University Prague, Dept. of Software Engineering, 1995.
- [KPT96] J. Kleindienst, F. Plášil, P. Tůma: Lessons Learned from Implementing the CORBA Persistence Service, In Proceedings of OOPSLA'96, San Jose, Oct 1996
- [MoZa95] T.J. Mowbray, R. Zahavi: The Essential CORBA, J. Wiley & Sons, 1995.
- [NEO96] Solaris NEO Operating Environment, Product Overview, Part No. 95392-003, Sunsoft Inc, March 96.
- [OHE96] R. Orfali, D. Harkey, J. Edwards: The Essential Distributed Objects. Survival Guide. John Wiley & Sons, 1996
- [OMG92] Object Service Architecture, OMG 92-8-4, 1992.
- [OMG94a] Common Object Services Volume I, OMG 94-1-1, 1994.
- [OMG94b] Persistent Object Service Specification, OMG 94-10-7, 1994.
- [OMG94c] Relationship Service Specification, Joint Object Services Submission, OMG 94-5-5, 1994.
- [OMG94d] Compound LifeCycle Addendum. Joint Object Services Submission. OMG 94-5-6, 1994.
- [OMG94e] Object Externalization Service. OMG 94-9-15, 1995.
- [OMG95a] Common Object Request Broker Architecture and Specification Revision 2.0, OMG 96-3-4, 1995.
- [OMG95b] Object Services RFP 5. OMG TC Document 95-3-25, 1995.
- [OMG95c] Object Management Architecture Guide, 3rd Edition, R.M. Soley (Editor), John Wiley & Sons, 1990.
- [OMG95d] Object Services Architecture, Revision 8.1, OMG 95-1-47, 1995.

- [OMG95e] Object Models, Draft 0.3, OMG 95-1-13, 1995.
- [OMG95f] CORBAservices: Common Object Services Specification, OMG 1995
- [OMG96] Description of New OMA Reference model, OMG 96-05-02, 1996
- [ORBIXa] Orbix, Programmer's Guide. IONA Technologies Ltd. Dublin, 1994
- [ORBIXb] Orbix, Advanced Programmer's Guide. IONA Technologies Ltd. Dublin, 1994.
- [OWeb] IONA Home Page, URL: <http://www.iona.com>
- [Sie96] J. Siegel: CORBA. Fundamentals and Programming. J. Wiley & Sons, 1996