OMG/CORBA: An Object-Oriented Middleware

Wolfgang Emmerich
Dept. of Computer Science
University College London
London WC1E 6BT, UK
w.emmerich@cs.ucl.ac.uk

Abstract

An increasing number of systems have a distributed software architecture. The main focus of this paper is on OMG/CORBA, a widely recognized middleware standard for heterogeneous and distributed system construction. We discuss CORBA's object model and its representation in the OMG interface definition language (IDL). We show how distributed system components written in different programming languages can be integrated using CORBA's programming language bindings to IDL. We distinguish static and dynamic invocations and sketch the higher-level services that are defined for CORBA. The paper concludes by indicating recent standardization efforts on CORBA undertaken by the OMG.

1 Motivation

An increasing number of organizations can no longer afford to rebuild their corporate IT systems from scratch whenever new technology is to be introduced. They rather have to build systems by integrating legacy and commercial off-the-shelf components with newly built components. As an example, consider a corporate IT system of a bank. It is likely to be based on components for account management, which were created in 70s and 80s and reside on mainframes. The system might include marketing and product database applications as well as loan authorization systems. These might have been constructed using relational database technology running on UNIX servers. More recently, applications supporting stock exchange traders might have been build using OS/2 or Windows-NT workstations. All these applications need to be integrated into a corporate IT system; the loan authorization system must be able to review balances of customer's accounts; when a share package is bought or sold the money needs to be debited or credited to an account; and the marketing department might have to have access the account details, too.

To achieve such an integration three major problem areas need to be addressed. First, integrating existing applications means to build *distributed systems* because the different components cannot easily be migrated from host to host. Second, component *heterogeneity* of the components in programming language, data representations and operating systems needs to be resolved; Finally, components need to define interfaces at appropriate levels of abstraction to advertise the services that they provide.

Components are not re-engineered but rather continue to reside on the hardware platforms they have been constructed for. In general, this leads to the construction of distributed systems. A distributed system consists of components on networked hosts that interact via middleware so that the components appear as a single integrated facility [Emmerich, 2000]. In other words distribution should be transparent to users, and to a large extent [ANSA, 1989] also to the application programmers that build and integrate components. To achieve transparency imposes several challenges: components need to be identified without requiring knowledge about their physical location; programmers should not have to distinguish local from distributed component accesses; communication failures are more likely to occur and should be to users concealed; users of a component should not need to be aware that there are other concurrent users; administrators should be able to decide on component replication without users or application programmers being aware of that.

There are many dimensions in which components of a corporate IT system can be heterogeneous. The heterogeneity of hardware platforms often implies different data representations in memory. Long integers, for instance, are represented as big-endians on IBM-370 and most RISC architectures, while Intel processors use a little-endian representation. Similar differences are found for character set encodings and floating point precision. Therefore, data may have to be transformed from one representation into another upon transmission between components. A different type of heterogeneity is likely to be caused due to components that are connected to different types of networks. Then heterogeneity of network protocols needs to be bridged. Finally, different languages are used for programming components. While Assembler and Cobol were the dominant languages for business applications on mainframes, recently developed components are more likely to be written in C, C++ or Java. The data models underlying these programming languages are considerably different and these differences need to be resolved in order to integrate components written in different languages.

The integration of different components of a distributed system is based on types of services. A service is an operation that is exported by one component, which ensures to perform the service with a certain quality. Services can be used by other components. Services may be parameterized and return results, possibly complex. Services may influence the state of the component, which may or may not be visible to other components. To solve the problem of integration, a homogeneous definition of the different types of components has to be provided. It should determine the services exported, the component state and the relationship the component has to other components.

The three problem areas sketched above are not specific to particular application domains. They rather occur across domain boundaries. It is, therefore, advantageous to support the integration of legacy components and the construction of distributed systems through a distribution middleware. The Common Object Request Broker Architecture [Object Management Group, 1998] (CORBA) is a specification of interfaces and protocols for such a middleware. CORBA is based on the object-oriented paradigm. It has been adopted by the Object Management Group (OMG), a consortium of more than 800 vendor and end-user companies as well as governmental and research institutions.

2 OMG/CORBA

Figure 1 displays the Object Management Architecture of the OMG. It identifies different categories of objects of a distributed object system as well as an object request broker by means of which these objects communicate. CORBAservices represent objects that provide very basic services, which are required for the construction of distributed systems. Examples of these are naming, concurrency control, transactions, event notification, relationships and many more. It is assumed that implementations of the CORBA standard provide most of these services. CORBAfacilities are objects that are useful in the construction of distributed systems. Examples are a help facility or a printing and spooling facility. Domain Interfaces define objects that are useful within a particular application domain. Among others, the OMG is currently standardizing Domain Interfaces for Health care, Telecommunication, Manufacturing and Finance. Finally, Application Objects are built for particular applications. Their construction leverages CORBAservices, CORBAfacilities and the Domain Interfaces using the mechanisms provided by the CORBA object model.

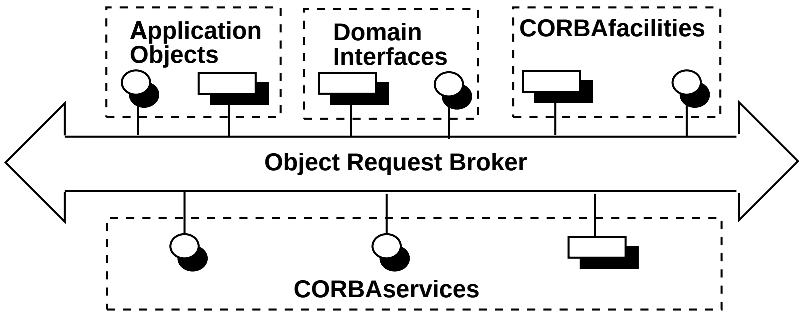

Figure 1: The Object Management Architecture

The CORBA object model determines an informal semantics for object-oriented concepts. The concepts are defined in a way that they can be mapped to a large variety of programming languages. The object-model defines concepts for object and non-object types, operations and attributes exported by objects, type-specific exceptions that may be the object's integrity is violated. The model also includes a mechanism for subtyping by means of which object types inherit attributes and operations of their supertypes.

The CORBA object model is used as a distributed system component model. Distributed system components are implemented by CORBA objects. Component types are implemented by object types. The services offered by components are determined by object type definition. A client component can interact with a server component by means of *object requests*. These are messages that trigger the execution of an operation in a server object. System or type-specific failures that may occur are treated as exceptions that should be caught by the client to react on the failure.

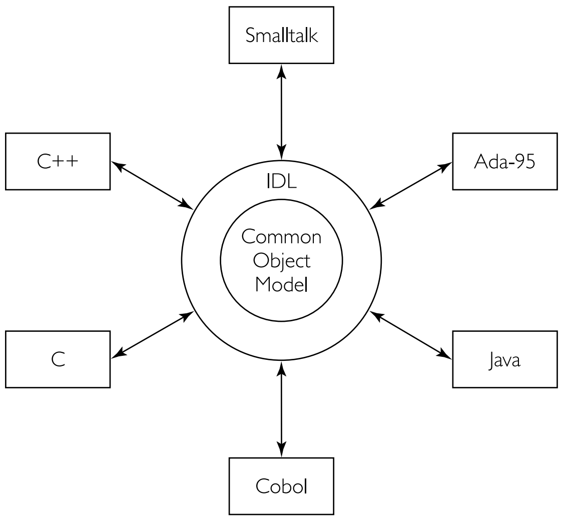

Figure 2: Programming Language Bindings to CORBA/IDL

The OMG Interface Definition Language (IDL) includes constructs for all the concepts of the CORBA object model. IDL is designed to be independent of a particular programming language, though its syntax is oriented towards C++. IDL is not computationally complete. It does not include language constructs to store variables or to express algorithms. As shown in Figure 2, the CORBA defines bindings to: C, C++, Smalltalk, Ada, Java and OO-Cobol. These programming language bindings determine how object types with their attributes, operations and exceptions are implemented in server objects and how clients can make object requests and catch exceptions the server may raise.

```
 1: interface ATM;
 2: interface TellerCtrl {
 3:  typedef sequence<ATM> ATMList;
 4:  exception InvalidPIN;
 5:  exception NotEnoughMoney{short available;};
 6:  readonly attribute ATMList ATMs;
 7:  readonly attribute BankList banks;
 8:  void accept_req(in Requester req,in short amout)
 9:       raises (InvalidPIN, NotEnoughMoney);
10:};
```

Figure 3: Example IDL Interfaces

Figure 3 shows an example of an IDL interface. It defines an interface for a teller machine controller that interconnects distributed ATMs with Bank account databases. Lines in the example are numbered for explanatory purposes only; IDL does not have line numbers. IDL does not support forward references; identifiers have always to be declared before they can be used. Forward declarations can be used to declare the existence of types and the first line is an example forward declaring type `ATM`. It is used in Line 3 to construct type `ATMList` using the `sequence` type constructor. The constructed type is used in Line 6 as an attribute type. Lines 4 and 5 define type specific exceptions. The exception in Line 5 is raised by the operation in Line 8 if a customer's bank account has insufficient funds to satisfy a withdrawal request. That exception has an additional data structure that is used by the teller machine controller to pass information about the maximum amount of money that can be withdrawn.

The availability of a common interface definition language reduces the complexity of integrating n different programming languages from $(n \times (n-1))/2$ bindings between different programming languages to n bindings between a programming language and the interface definition language. Integration is further simplified by the way that CORBA IDL is carefully chosen to be mapped relatively easily to a variety of programming languages.

CORBA object requests, i.e. the invocation of an operation from a remote server object, can be defined statically or dynamically. A static invocation is defined at the time a client object is compiled, while dynamic invocations are determined when the client object executes. The definition of a static request is achieved by using a local procedure call into a server proxy. A dynamic invocation is achieved by using CORBA's reflection mechanisms. In particular, CORBA supports a dynamic invocation interface, which treats object requests as objects themselves. These objects have operations to determine the name of the called operation, to set the request parameters and obtain the operation result. CORBA also supports an interface repository that can be used by client objects to obtain type information of server objects at run-time. Figure 4 shows an example interface repository content and the corresponding interface definition.

```
module SoccerMgmt {
 interface Player;
 interface Team {
  typedef Sequence<Player> PlayerLst;
  exception InvalidNumber;
  attribute PlayerLst members;
  void add(in short number,
           in Player p)
       raises(InvalidNumber)
 };
};
```

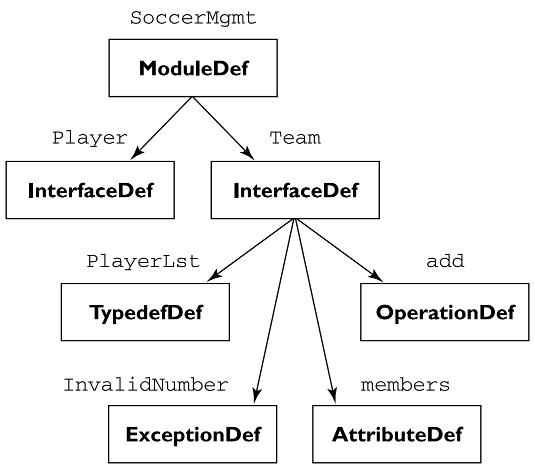

Figure 4: CORBA/IDL Interface and its Interface Repository Representation

As suggested by Figure 4, the interface repository manages persistent abstract representations of the abstract syntax tree of IDL interfaces. IDL type definitions are entered into the interface repository using an IDL compiler and they can be traversed using a set of interface definitions that are defined for the interface repository.

Figure 5 shows the components that are involved in the interaction between object request broker, client and server objects at run-time. Both client and server objects initialize themselves using the ORB interface. The ORB interface also determines the operations that any server object inherits from the pre-defined root of the inheritance hierarchy. The client object issues the request and uses either the static or the dynamic invocation interface. A static request is issued by calling a client stub that is generated from an IDL interface description. Static object requests are synchronous. A dynamic invocation is done using the dynamic invocation interface. The dynamic invocation interface supports both synchronous and deferred synchronous requests. After having issued a deferred synchronous request, control is given back to the client object until a point in time when it polls for the operation result. The object broker uses the object reference that is submitted by the client as part of the request in order to locate the server object. If necessary, the broker activates the object using an object adapter. The broker then invokes the implementation skeleton, which is also generated from the IDL interface definition of the client object. The skeleton finally calls the operation that was requested by the client.

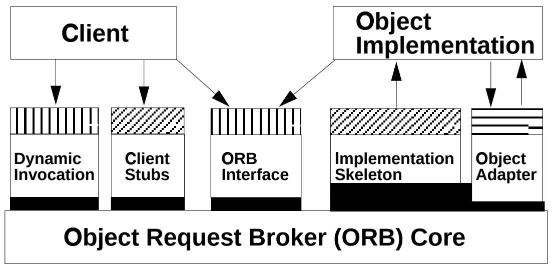

Figure 5: Components involved in Object Requests

Heterogeneity of different data representations that might occur if the broker operates on different hardware platforms is resolved in client stubs and the server skeletons. This is achieved by mapping native atomic data types to a common data representation (CDR) that is specified by the CORBA standard.

The use of a programming language independent interface definition language for which a number of programming language bindings are available achieves programming language interoperability. In order to request an operation execution of a Cobol server object from within a Java applet, for instance, the IDL compiler is instructed to generate Java client stubs for the interface definition and Cobol server skeletons.

The basic mechanisms discussed so far enable heterogeneous and distributed objects to request operation executions from each other. We now discuss CORBAservices that are layered on top of these basic mechanisms. The services aim at solving a number of problems that commonly appear in distributed systems. The services are all specified in IDL and their implementations are distributed objects themselves.

3 CORBA Services

The CORBA services have been adopted by the OMG Technical Committee and its Object Services Task Force between 1993 and 1996 in five stages. In each stage a request for proposals (RFP) was issued that solicited proposals for two to four service specifications. Individual OMG members or member consortia responded to these RFPs by submitting proposals for services. RFP1 solicited the Naming, Event, Life Cycle and Persistent Object services; RFP2 targeted the Relationship, Externalization, Concurrency Control and Transaction services; RFP3 aimed at adopting Security and Time services; RFP4 demanded the Collection, Properties and Licensing services; and RFP5 finally solicited the Query Trading services. We give a brief outline of each of these services in this section.

The *Naming* service supports finding object references in a location transparent way. It facilitates the definition of hierarchically nested naming contexts. Naming contexts support the binding of names to server objects. Names are sequences of the name components in the same way as NFS file names can be composed of directory names. Clients use the naming context to resolve names in order to obtain object references.

Object requests are non-anonymous forms of communication between two different objects. Distributed systems often demand other forms of communication, such as broadcasts where a component interacts with more than one component at the same time or anonymous communication where the client does not have to know its server. The *Event* service supports several of these communication primitives using the concept of channels through which events are communicated. Objects producing events push these into a channel. The channel then informs all those consumer objects that have previously registered an interest with the channel.

The life cycle of distributed objects is considerably more complicated than those of local objects. It is supported by the *Life Cycle* service. Upon object creation the location of the new object has to be determined in addition to the way how objects are constructed and initialized. The Life Cycle service supports that by factories that create objects in their address space. Policies for administrating the location of these factories are supported by an object type for factory finders. During a distributed object's life time, it might have to be migrated from one server to another. Heterogeneity of data representation as well as of the machine code that executes operations may have to be resolved. Moreover, garbage collection algorithms that can be devised within one process are inapplicable in a distributed setting due to the autonomy of the different operating systems involved. The Life Cycle service determines abstract operations for copying, moving and deletion that have to be redefined in all CORBA objects.

Persistence is the property of an object to survive the lifetime of the processes in which they are executed. The persistence of object references is maintained by the object request broker in order to enable the restart of objects that have been temporarily inactive. The persistence of the state of objects, however, is not maintained by the broker but specified in interfaces of the *Persistent State* service. The service supports storing the object's state in a data store. This can be a relational database, an object database or the file system. The Persistent Object service is defined in a way that an object does not have to be aware of the type of data store in which its state is stored. For achieving persistence the attributes that store the object's state (and may not be exposed in IDL) are specified in the persistent state service definition language (PSSDL). A PSSDL compiler creates the code that is necessary for transferring and retrieving the object state into or from a data store.

The *Relationship* service supports to relate different distributed objects. The service supports three different levels of abstraction. The lowest level facilitates the creation, traversal and deletion of relations between two objects. A relation can be established between objects without having to change the related object's type. The middle layer provides algorithms for traversing a graph of related objects. The top layer provides to particular types of relationships, aggregation and reference. By distinguishing these relationships composite objects can be identified. These composite objects are also supported by other services, such as the life cycle service and the externalization service. Deleting a composite object using a life cycle operation will delete all its component objects and moving a composite object to another host will move all its components, too.

The *Externalization* service supports the transformation of composite objects into a stream of bytes and the restoration of the composite object from such a stream. In order to use the service, each component object type has to implement the streamable interface and implement operations `externalize` and `internalize`. The implementations use operations for writing into and reading from a stream that are provided by the service.

CORBA does not impose any restrictions on the degree of concurrency between different clients of a server object. It is sometimes inappropriate that one client interferes with the execution of operations that are requested by another client. To avoid lost updates or inconsistent analysis that may result as a consequence of such interference, server objects may implement a concurrency control scheme. The *Concurrency Control* service provides mechanisms for locking objects in different lock modes. The service defines a lock compatibility matrix. Requesting locks in incompatible modes will delay execution to a point in time when the lock becomes available or a time-out appears.

Failure transparency demands the concealment of faults. In particular, it requires the integrity of object states to be preserved, even in the event of hardware/software failures. This requires that a sequence of object requests is performed in an atomic way, i.e. either completely or not at all; that the sequence leaves the objects involved in a consistent state when it finishes. that it is isolated, i.e. it does not interfere with other concurrent request; and that its effect is durable. Sequences of object requests having these properties are referred to as transactions. The *Transaction* service specifies object-oriented interfaces for the two-phase commit protocol by means of which they can be implemented in a distributed setting.

The network through which the servers that host distributed objects are connected might be insecure. It may be enable intruders to eavesdrop requests, tamper with requests, masquerade requests and replay requests. The aim of the security service RFP was to establish a set of interfaces by which security could be added to an otherwise insecure object request broker. It was however then recognized that security is an pervasive property that affects the core of the object request broker as well as other object services and facilities. The CORBA *Security* specification, however, is still included in the CORBAservices volume. It includes a security reference model for authentication, access-control, auditing and non-repudiation. Moreover, a security architecture is defined and a number of IDL definition for security are specified on which ORB implementors can rely.

In a distributed systems sometimes the need occurs to have reasonably accurate information about the current time. An example is the log kept for security audits. The main purpose of the *Object Time* service is to standardize an interface whose implementation returns the current time in UTC format together with an estimate of inaccuracy. The Object Time service can be implemented in many different ways. Usually it is built on top of an existing time synchronization service, such as the OSF/DCE Time Service.

Objects are often grouped together in lists, sets and the likes. It is advantageous to standardize interfaces to these aggregations of objects. The *Collection* service determines such interfaces. Apart from the basic collection interfaces it also supports the creation of collections through collection factories and the iteration over collections using iterators.

The set of attributes that are exhibited by an object are statically determined by the object's type. Sometimes it is useful to dynamically attach properties to distributed objects even though there is no attribute for them. The *Property* service supports the attachment and detachment of name/value pairs to objects at run-time without a need for changing object types.

In a distributed object system where different autonomous objects request services from each other it becomes important to make sure that only those services are being used that have been paid for. The *Licensing* service supports different models of making services available. This can be done on the basis of time windows that determine start and expiry dates and meters that measure how often a service was used. Licensing policies can be determined for many granularities ranging from individual method invocation to the right to use collections or graphs of related objects.

The *Query* service supports selecting subsets of collections using predicates that are specified at run-time in some query language. Updates of collections are not supported by the query service. The query service deliberately does not impose a particular query language but leaves a certain degree of freedom to support different query languages, such as SQL3 or OQL.

Naming supports obtaining object references of server objects based on the definition of external names. This might not always be the most appropriate way to locate an object. The *Trading* service supports locating objects using a description of the services object offer. The underlying assumption is that both client and server objects use a common conceptual framework in order to describe functionality and quality of services. Server objects use that framework to advertise their services with the trader. Client objects use the framework to inform the trader of the functionality and quality of the services they desire. The trader matches the client request against the server objects that have registered themselves.

4 Summary and Future Adoptions

In this short paper, we have given a concise overview of the distributed object technology supported by the mature part of the OMG/CORBA standard that is widely implemented by CORBA products. We have discussed the object model and its availability in the OMG interface definition language, we have discussed different programming language bindings, the object management architecture and the components that are involved when an object request is make. Finally, we have given a brief overview of the different object services that have been accepted so far.

The literature reports about a number of successful usages of OMG/CORBA for building distributed system architectures. [Emmerich et al., 2001] reports about such a use for integrating different systems of the trading department of a large German bank.

A considerable effort is spent by the OMG now on the definition of Domain Interfaces. Those will standardize interfaces that can be demonstrated to be common within a particular vertical market segment. The OMG has created different task forces for these domains. Among those are task forces for business objects, finance, electronic commerce, telecommunication, health care and manufacturing. More taskforces are going to be started.

The CORBA object model only supports interactions between one client and one server object. Moreover, in order to achieve an integration the client object needs to be changed to invoke a client stub or use the dynamic invocation interface. The CORBA Component Model that is part of the CORBA 3.0 standardization effort [Siegel, 1999] will address these issues and allow more flexible ways of integrating client and server objects. In particular CORBA components can have multiple interfaces and they can publish and subscribe to event-based communication. CORBA components also solve some of the difficulties in achieving enterprise computing, such as the difficulties in implementing two-phase commit transactions or persistence, by providing a container-based programming model, similar to the one known from Enterprise Java Beans [Monson-Haefel, 1999].

Most current CORBA products are only of limited use in real-time and embedded systems because all requests have the same priority. Moreover the memory requirements of current middleware products prevent deployment in embedded systems. These problems have been addressed by various research groups. TAO [Schmidt et al., 1998] is a real-time CORBA prototype developed that supports request prioritization and the definition of scheduling policies. The CORBA 3.0 specification [Siegel, 1999] builds on this research and standardizes real-time and minimal middleware.

References

[ANSA, 1989] ANSA (1989). The Advanced Network Systems Architecture (ANSA). Reference manual, Architecture Project Management, Castle Hill, Cambridge, UK.

[Baker, 1997] Baker, S. (1997). *CORBA Distributed Objects using Orbix*. Addison Wesley.

[Emmerich, 2000] Emmerich, W. (2000). *Engineering Distributed Objects*. John Wiley & Sons.

[Emmerich et al., 2001] Emmerich, W., Ellmer, E., and Fieglein, H. (2001). Tigra – an architectural style for enterprise application integration. In *Proc. of the* 23rd *Int. Conference on Software Engineering, Toronto, Canada*. ACM Press. To appear.

[Monson-Haefel, 1999] Monson-Haefel, R. (1999). *Enterprise Javabeans*. O'Reilly UK.

[Mowbray and Malveau, 1997] Mowbray, T. and Malveau, R. (1997). *CORBA Design Pattern*. Wiley.

[Mowbray and Zahavi, 1995] Mowbray, T. and Zahavi, R. (1995). *The Essential CORBA: Systems Integration Using Distributed Objects*. Wiley.

[Object Management Group, 1998] Object Management Group (1998). *The Common Object Request Broker: Architecture and Specification Revision 2.2*. 492 Old Connecticut Path, Framingham, MA 01701, USA.

[Orfali et al., 1996] Orfali, R., Harkey, D., and Edwards, J. (1996). *Client/Server Programming with CORBA Objects*. Wiley.

[Orfali et al., 1997] Orfali, R., Harkey, D., and Edwards, J. (1997). *Instant CORBA*. Wiley.

[Otte et al., 1996] Otte, R., Patrick, P., and Roy, M. (1996). *Understanding CORBA: The Common Object Request Broker Architecture*. Prentice-Hall.

[Schmidt et al., 1998] Schmidt, D., Gill, C., and Levine, D. (1998). Evaluating Strategies for Real-Time CORBA Dynamic Scheduling. In 19th *International IEEE Real-Time Symposium*. IEEE Computer Society Press.

[Siegel, 1996] Siegel, J., editor (1996). *CORBA Programming*. Wiley.

[Siegel, 1999] Siegel, J. (1999). Component and Object Technology: A Preview of CORBA 3. *IEEE Computer*, pages 114–116.

[Vogel and Duddy, 1997] Vogel, A. and Duddy, K. (1997). *Java Programming with CORBA*. Wiley.