Evaluating CORBA Portability: The Case of an Object Group Service*

Pascal Felber Rachid Guerraoui André Schiper
Ecole Polytechnique Fédérale de Lausanne
Département d'Informatique
CH-1015 Lausanne, Switzerland
Tel: +41 21 693 5269
Fax: +41 21 693 6770
E-mail: Pascal.Felber@epfl.ch
Keywords: CORBA, portability, interoperability, fault tolerance, experience

Abstract

One of the most attractive aspects of the Common Object Request Broker Architecture (CORBA) lies in its open standard nature. Code that is based only on constructs described in the CORBA specification is portable between Object Request Brokers (ORBs), without depending on any single ORB implementation. However, this portability feature is not completely achieved in real-world applications. This paper discusses CORBA portability issues through our experience in implementing a CORBA Object Group Service (OGS) and porting it on different ORBs. In particular, we discuss some implementation choices we made with Orbix and VisiBroker, and we point the impact of these choices on the service portability and interoperability. This sheds some light on whether the current CORBA specification is mature enough to be deployed in industrial systems that have requirements regarding portability and interoperability. We also present how new CORBA specifications solve part of the shortcomings mentioned in this paper.

1 Introduction

1.1 Context

The *Object Management Architecture* (OMA) [OMG98] is now widely considered as a universal conceptual infrastructure for building distributed applications. Following the OMA approach, distributed components are made *portable* as long as they only rely on constructs specified by the *Common Object Request Broker Architecture* (CORBA), and they are made *interoperable* provided that their interfaces are written in the CORBA *Interface Description Language* (IDL) and communicate through the standard *Internet Inter-ORB Protocol* (IIOP). *Portability* means here the ability to use a single implementation of a component with different Object Request Brokers (ORBs), by simply recompiling it. *Interoperability* means the ability of a given component implementation, written in a given language and running on a given ORB, to cooperate with another component, possibly written in a different language and running on a different ORB.

Portability and interoperability are crucial aspects in the development of large scale distributed applications that use heterogeneous components. The degree to which CORBA guarantees portability and interoperability of an application depends however on the nature of the application components. When these components rely on some features not completely specified by CORBA, portability and interoperability might be affected.

1.2 Experience

In this paper, we particularly address CORBA portability, which we discuss through our experience in the development of a CORBA *Object Group Service* (OGS), which we implemented first using Orbix [ION97] and then ported on VisiBroker [Vis97]. This work was performed in the context of the European projects OpenDREAMS and OpenDREAMS II

*Research supported by OFES under contract number 96.0454, as part of the ESPRIT project OpenDREAMS II (project 25262).

(Open Distributed Reliable Environment, Architecture and Middleware for Supervision), which goal is to answer the needs of advanced industrial Supervision and Control Systems (SCS), based on standard middleware specifications. One of the major goal of this project is to keep independency from proprietary systems, and to ensure interoperability and openness of the system. With this respect, portability is one of the major concerns of this project, and this paper gives some answers to the question whether the current CORBA specification is mature enough for being deployed in industrial system, in particular those having requirements regarding portability and interoperability.

Our CORBA Object Group Service (OGS) provides the ability to group CORBA objects and invoke them as a single fault-tolerant entity. Group invocation can be performed with various properties, such as atomicity and total order. Atomicity ensures that all non-crashed members of a group receive an invocation or none of them does, while total order ensures that all group members that receive two invocations, receive them in the same order. The group paradigm was shown to be very useful in supporting highly available applications through replication. The replicas of an object are gathered inside a group and, thanks to atomicity and total order properties, group invocation hides replication by ensuring that all replicas behave as a single copy. OGS complements the existing CORBA services such as the *Object Transaction Service*, the *Object Persistence Service*, the *Event Service*, and the *Life Cycle Service*.

1.3 Portability Issues

Although OGS was specified following the OMG guidelines and our target ORBs (Orbix and VisiBroker) are widely considered as CORBA compliant, it was not possible to write an OGS code that was fully portable from one ORB to another. The reason for this limitation is that, by its very concurrent and distributed nature, OGS relies on specific features such as asynchrony[1] and dynamic invocations, which are not completely specified by CORBA and are usually left to the interpretation of ORB vendors. This led us to various OGS implementation alternatives and posed serious problems when porting the OGS code from one ORB to the other. For instance, asynchrony is a mandatory feature in the implementation of OGS and is not guaranteed by the CORBA specification. Some form of asynchrony can be obtained using event handling or multi-threading, but none of these solutions was straightforward to port from Orbix to VisiBroker. Another example is typed communication which is an important aspect of OGS as it provides a transparent way of invoking object groups. But the *Dynamic Skeleton Interface (DSI)*, which provides the adequate CORBA support for typed communication, is not fully implemented in Orbix. Other specific language issues such as *implementation inheritance*, *object equivalence*, *server-side mapping*, and *scoped names* were also obstacles for full portability. As a consequence, we came out with slightly different implementations of OGS on Orbix and VisiBroker, each version providing different properties and performance.

Although, as we show in the paper, some of the problems we point out will be circumvented by new CORBA specifications (CORBA 2.2 and the CORBA Messaging Service), we believe that it is important to have these problems in mind in future development of new reliable object-based distributed systems. Furthermore, understanding these problems might help in better understanding these new CORBA specifications.

1.4 Roadmap

The rest of the paper is organized as follows. Section 2 briefly presents OGS design and architecture. Section 3 discusses the implementation alternatives that we have faced when dealing with asynchrony and concurrency. Section 4 discusses the issue of providing group transparency through typed communication. Section 5 points out specific language issues which also limited OGS portability. Section 6 discusses the way the new CORBA 2.2 specification and the future Messaging Service circumvent some of the problems pointed out in the paper. Finally, Section 7 concludes the paper.

2 CORBA and OGS: Background

Before discussing the portability issues we have faced when implementing our Object Group Service, we briefly describe in this section the CORBA model and OGS' design and architecture. We mainly remind here the basic characteristics of OGS that are important in our context. More information on OGS can be found in [FGG96] and [FGS98], or directly at `http://lsewww.epfl.ch/OGS/` where binary code versions of OGS are also available.

[1] In our context, and as we will describe later in the paper, asynchrony means the ability of a client to invoke a server without blocking.

2.1 The Object Management Architecture: Background

Figure 1 shows the five major parts of the OMA reference model. (1) The *Object Request Broker* (ORB) enables objects to transparently invoke remote operations and receive replies in a distributed environment. (2) The *Object Services* are a collection of interfaces and objects supporting basic functionalities useful for most CORBA applications. (3) The *Common Facilities* are a collection of interfaces and objects providing end-user-oriented capabilities useful across many application domains. (4) The *Domain Interfaces* are meant to be used only in specific vertical application domains. Finally, (5) the *Application Interfaces* are interfaces specific to end-user applications. The two parts of the OMA reference model that are important in our context are (1) the ORB and (2) the services.

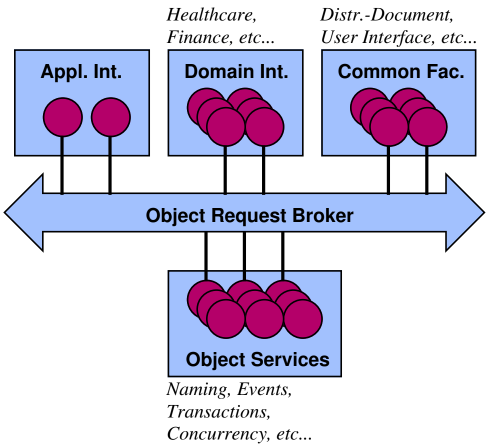

Figure 1: The OMA Architecture

1. The Object Request Broker is viewed as an "object bus", through which heterogeneous objects can interoperate. Integration of distributed objects is available across platforms, regardless of networking transports and operating systems. Each object interface is specified in the OMG *Interface Definition Language* (IDL), which is implementation independent. Clients use *object references* to identify remote objects and invoke operations on them. Objects are not tied to a client or server role: they can act both as client and as server.

2. A CORBA service is basically a set of CORBA objects with their corresponding IDL interfaces, and these objects can be invoked through the ORB by any CORBA client. Services are not related to any specific application but are basic building blocks, usually provided by CORBA environments. Several services have been designed and adopted as standards by the OMG. Among these services are the Life Cycle Service, used for creating and deleting objects, the Persistence Service, used for storing the objects on persistent storage, the Transaction Service that lets multiple distributed objects participate in atomic transactions, and the Event service which allows multiple supplier objects to communicate with multiple consumer objects asynchronously.

Although several CORBA services and facilities have been specified by the OMG, nothing has been done yet concerning high-availability. In fact, a request for proposals (RFP) for Fault Tolerant CORBA is currently being issued. Some systems provide replication of CORBA objects, but they are proprietary and do not follow a service approach [FGG96].

2.2 OGS

OGS manages groups of CORBA objects and provides primitives to communicate atomically with these groups. Groups may be used for replicating objects, and making them highly available and fault tolerant. Clients do not need to know the number, the identity, or the location of the members of a group. A client can bind to a group using a group name, and issue a single request to all group members at once. OGS is inherently distributed, and does not depend on any global, critical, or centralized component. The OGS interfaces provide for different levels of reliability. In the following, we briefly describe OGS components and the way they are related with each other.

OGS Views

OGS provides several interfaces associated with the different views of the service: (1) the *client's view*, used to invoke the group, (2) the *member's view*, used by a group member to change its status within the group (e.g., *join* or *leave* the group) and to communicate with other objects in the group, and (3) the *service's view*, used by OGS to invoke operations on the members of a group. These different views are illustrated in figure 2.

1. **Client's view** allows a client to get information about groups, to send multicasts to the members of a group, and to send messages to individual members. Clients interact with groups through an interface of type

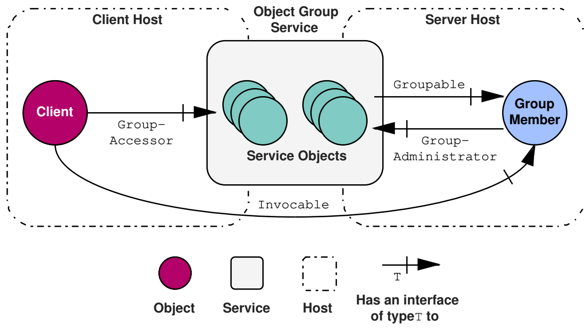

Figure 2: OGS Views

`GroupAccessor` which acts as a local representative for the group. `GroupAccessor` objects are created using a `GroupAccessorFactory` object. The `GroupAccessor` interface defines an operation for multicasting messages to the group that it represents (`multicast()`).

2. **Member's view** is a superset of the client's view, and is defined by the `GroupAdministrator` interface. Objects can join and leave groups using the `join_group()` and `leave_group()` operations of `GroupAdministrator` objects. `GroupAdministrator` objects are created using a `GroupAdministratorFactory` object.

3. **Service's view** is defined by the `Groupable` interface. This interface must be supported by member objects and enables OGS to issue callbacks to them. The `Groupable` interface defines operations for receiving messages, for group composition notification (`view_change()`), and for state transfer (`get_state()` and `set_state()`).

Interacting with OGS

Figure 3 gives an abstract view of the interactions between clients, servers, and OGS. The `GroupAccessor` object (Acc) acts as a client-side representative for the group, and is located on the same host (or in the same process) as the client. The `GroupAdministrator` objects (Adm) interact directly with group members, and are located on server hosts. Performing a multicast to the group initiates a protocol between `GroupAccessor` and `GroupAdministrator` objects, which ensures that messages are delivered to the members according to some condition (e.g., total order).

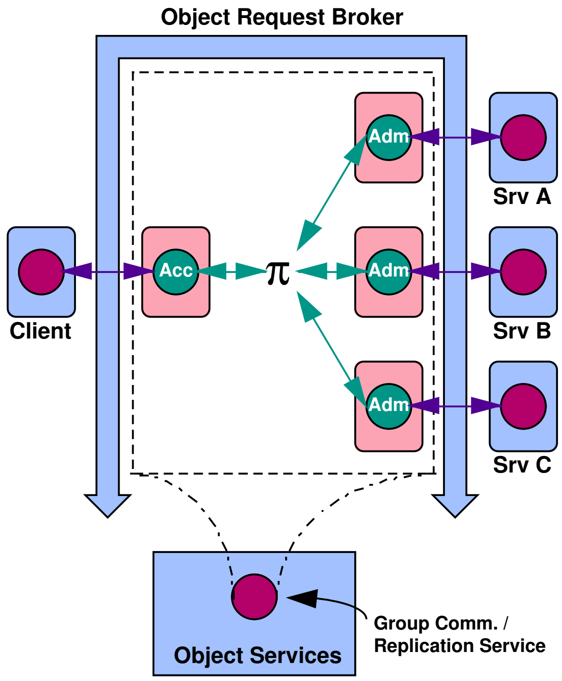

Figure 3: OGS Architecture

3 Asynchrony and Concurrency

This section discusses portability issues related to the asynchronous and concurrent behavior of OGS. We describe first issues related to events and threads and then issues related to invocation semantics.

3.1 Events and Threads

At some point, a CORBA server has to wait for incoming client requests. For doing this, it calls the `impl_is_ready()` function on the Basic Object Adapter (BOA), and the main event loop becomes under control of the ORB. This causes problems with applications using a Graphical User Interface (GUI) such as X Windows, that have their own event loop. Another situation that happens frequently is when a server is accessed by multiple clients at the same time, and wants to handle requests in parallel (a typical example is a WWW server). These problems can be solved by using multiple threads.

Multi-threaded CORBA servers gain the ability to process several requests at the same time. When dealing with requests that require a long processing time (e.g., database queries), multi-threading support is necessary to serve incoming requests while performing time consuming processing. Both Orbix and VisiBroker support multi-threading.

An alternative to multi-threading is to use specific server activation modes, like the *unshared activation mode* specified by CORBA, that starts a new server process for each different request. This solution is more "heavyweight" than multi-threading, and makes it more difficult to synchronize and to share data between different server processes.

The main problem related to multi-threading is mutual exclusion on shared resources. Since all threads of a process share the same data, the programmer must ensure that two concurrent threads do not access the same critical resource at the same time, leading to inconsistencies.

In OGS, multi-threading is especially necessary in one situation: when a client issues a multicast request to the a `GroupAccessor`, the latter launches a distributed protocol and waits for replies from servers before returning. In other words, multicasting is a *blocking* operation that expects replies (i.e., incoming messages) while being processed.

This problem can be solved in a single-threaded system[2] if the ORB provides primitives for processing incoming events in the context of a request. Orbix provides such a primitive (`processEvents()`), but VisiBroker does not. With a multi-threaded model, this problem is solved by starting threads for incoming requests. Mutual exclusion is ensured by mechanisms such as *locks* and *condition variables*. If objects are meant to be co-located, the locks must be recursive so that a thread trying to acquire the lock will not be blocked if it already owns this lock[3]. Special attention must be given to the problem of desequenced requests: OGS provides global ordering of requests, and the threads scheduler must not break this ordering.

As of version 2.1 of the CORBA specification, thread support and management are not specified, and thus are not portable. Although both Orbix and VisiBroker support multi-threading, they provide very different programming models. Both provide object-oriented wrappers for threads, locks and condition variables that allow platform independence (e.g., between Posix and Windows NT threads), but these wrapper classes are not compatible with each other.

The biggest difference between Orbix and VisiBroker threading models appears at thread creation: Orbix provides fine control over thread creation by enforcing the programmer to explicitly create threads in so-called `Filter` objects invoked before request processing. On the other hand, VisiBroker automatically and transparently starts threads for incoming requests, without letting the programmer choose which request has to be started in a new thread, and which has not. The VisiBroker model is easier to use and more efficient, but less powerful. OGS uses its own wrappers for abstracting the differences between both models.

3.2 Communication Semantics

CORBA remote object invocations are based on an RPC-like mechanism and are, by default, synchronous, i.e., the client is blocked until the invocation returns. This behavior is convenient for many applications, but is not well adapted when, for instance, a client has to perform several invocations to different servers, and the servers or the links are slow; in this situation, the client would issue all requests at once and let them execute in parallel while waiting for replies.

In addition, the CORBA specification allows to declare operations to be *oneway*: the operation does not return any value and the client is not interested in waiting for the completion of the invocation. The specification also defines primitives for sending *deferred synchronous* invocations: client thread continues processing, subsequently polling to see if results are available. Currently, this model is only available when using the *Dynamic Invocation Interface* (DII). Note that oneway invocations are *not* asynchronous[4].

The problem with these types of invocations is that their semantics are (intentionally) vague in the specification. Two styles of execution semantics are defined by the object model [OMG98]:

- **At most once:** if an operation request returns successfully, it was performed exactly once; if it returns an exception indication, it was performed at-most-once.
- **Best-effort:** a best-effort operation is a request-only operation, i.e., it cannot return any results and the requester never synchronizes with the completion, if any, of the request.

The second execution style does not provide any guarantee. A *compliant* ORB can just send the invocation message once, or even discard it. If the

[2] With the limitation that only one client accesses the `GroupAccessor` at one time.

[3] This is due to the fact that the target object does not know whether the execution of an operation results from a remote invocation or from a local method call.

[4] In OMG's terminology, asynchronous means that the client ORB does not synchronously invoke the target to obtain a reply. Some other agent is required to separate the client ORB from the target.

message is sent more than once, this can result in more than one invocation on the server and lead to consistency problems if the operation is not idempotent. Furthermore, the specification does not enforce request-only operation to be non-blocking; a compliant ORB can implement oneway operations using transport mechanisms that can block the client. The term *oneway* specifies that the client will not wait for the completion of the operation on the server side (which can last a long time depending on the application), but it does not specify that the client will not wait for the server to acknowledge the reception of the *message* that transports the remote invocation; thus, a oneway call can block "forever" *at the transport level*, which can block in turn the application. The CORBA specification is completely silent on what happens in case of a link failure, if the network is congested, or if the server is extremely busy. Since this behavior is not specified, there is no way to handle it portably.

For instance, an *Internet Inter-ORB Protocol* (IIOP) oneway invocation can block the entire process on a oneway call just because a TCP/IP buffer fills up. According to the CORBA specification [OMG98], *given TCP/IP's flow control mechanism, it is possible to create deadlock situations between clients and servers [...] ORBs are free to adopt any desired implementation strategy, but should provide robust behavior.* Although the client could detect a possible deadlock when performing an IIOP call, this behavior is not guaranteed by the CORBA standard.

A solution for simulating non-blocking invocations with existing ORBs is to use multi-threading at the client side, using a separate thread for each invocation. But the use of threads considerably raises the application's complexity and the probability of programming errors, by adding resource and synchronization problems. In the current OGS implementation, we use multiple threads on the server side, and standard oneway invocations on the client side.

3.3 Locality Constraints

Whereas CORBA objects are independent of their real location, some of our service objects have to be located on the client and server sites for OGS to provide the required semantics, due to the fact that OGS has to deal with failures and partitions. This is more a semantic requirement than an architectural requirement, since our services can actually be installed anywhere; service localization is thus related to *quality of service* (QoS).

It makes sense to locate the service on the same host as the server; that way, the server will be considered as failed only if the machine crashes. If they are on different hosts, the crash of any of the machines, or a network partition between them, will be considered as a failure of the server. The best and most efficient strategy consists in locating OGS objects in the client and server processes (i.e., to co-locate the service objects with the application objects), but it sometimes makes applications more difficult to program and to debug, and it can waste resources if several processes use OGS on the same host.

As a matter of fact, invocations to co-located objects are generally mapped to simple function calls in ORB implementations. While this makes application code more efficient, it creates new problems, such as the need for recursive locks (see Section 3.1), or the different behavior of oneway invocations (with a simple function call, the client will have to wait for the operation's completion). Furthermore, ORB implementations generally do not support invocations to co-located objects that use the DSI.

The current implementation of OGS can run as a separate process on the client and the server's host (the `OGSd` daemon program), or be linked with application code (the `OGSl` library). The first approach has the advantage of decoupling the service from the application, enabling several applications running on the same host to use the same resources. It also allows user applications written in Java to use the C++ service. The second approach is more efficient since inter-process communications are more costly that invocations between objects located in the same process, but it makes application code bigger and transfers the asynchrony problems described in Section 3 to the application (i.e., the user application has to be multi-threaded, or explicitly handle events).

4 Typed Invocations

Similarly to the CORBA Event Service [OMG97], OGS provides two types of communication: *untyped* and *typed* communication. Untyped communication enables clients to send only values of type `Any`[5] as messages. These messages are received by the servers through their `deliver()` operation. While this message-passing type interface is useful and more efficient in some specific situations, it is generally more convenient for clients to directly invoke an operation of the server interface. Typed communication provides this abstraction. For instance, if the members of an object group support an `Account` interface, a client can directly invoke the `makeDeposit()`

[5] A CORBA `Any` variable can contain a value of any type.

operation; OGS intercepts this call and invokes the `makeDeposit()` operation on each member of the group. The service transparently filters messages and returns a single reply to the client. Of course, typed communication requires that all servers implement the same IDL interface.

Typed communication is achieved by OGS using two advanced features of the CORBA specification: the *Dynamic Skeleton Interface* (DSI) and the *Dynamic Invocation Interface* (DII). OGS uses the DSI to accept requests that are actually aimed at the server interface. The DII is used to construct the invocations for the server interface.

The approach used in OGS for implementing typed multicast communication is similar to the CORBA *request level bridging* [OMG98]. Translation from a client request to a multicast (for a set of servers) is performed by application style code outside the ORB. Client and servers mediate through a common protocol between distinct execution environments (possibly different ORBs).

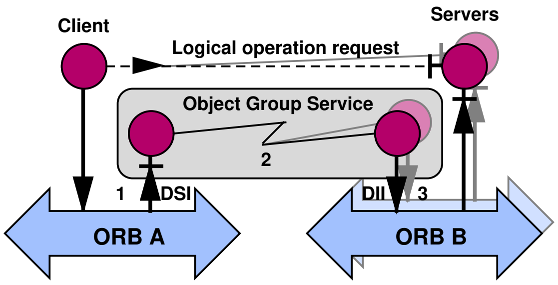

Figure 4: Implementing Typed Communication

The general principle for implementing typed communication is illustrated in Figure 4. A detailed description of this mechanism can be found in [FGS98]. When implementing typed invocation in OGS, we encountered several problems which are described in the rest of this section.

4.1 Dynamic Skeleton Interface

The *Dynamic Skeleton Interface* is not fully implemented in Orbix 2.2. Most notably, the CORBA specification states that a C++ function named `_this()` that returns an object reference to the target object must be available to static and dynamic implementations. But this function is not supported by Orbix, and since dynamic implementation objects do not inherit from the `CORBA::Object` base class, it is not possible to obtain and export a portable object reference to a dynamic implementation with Orbix. As a consequence, typed communication is not provided in the current Orbix implementation of OGS[6].

Furthermore, the mechanism for registering dynamic implementations is left unspecified: Orbix uses a BOA method (`setImpl()`)[7], while VisiBroker automatically registers them, based on the parameters given to the constructor of the dynamic implementation.

4.2 Exceptions

The typed version of OGS uses the DSI on client side to intercept requests performed on the server's interface, and the DII on server side to reconstruct the request and actually invoke the servers. In between, the request is transmitted and the reply is returned by OGS as untyped values, i.e., they are packed in CORBA `Any` variables. Although it is easy to insert the request's parameters and return value into `Any` variables, the CORBA specification does not make it possible to pass back exceptions from the server to the client. On the client side, indeed, returning an exception resulting from a DSI invocation requires giving an `Any` value containing the exception to the dynamic request object. On the server side, the exception returned as result from the DII invocation is a pointer to an object deriving from the `Exception` class, and there is no way of packing this object into an `Any` variable[8]. This implies that the typed version of OGS does not allow to return all exceptions at the moment.

4.3 Any Type Mapping

`Any` values are used intensively in the typed version of OGS, but VisiBroker does not fully comply with the CORBA specification concerning the C++ mapping of the `Any` type. For extraction of complex data types, such as structures, the specification states that the extraction function must be prototyped for passing parameters as pointers rather than values to increase efficiency. As a matter of fact, VisiBroker does not use pointers, preventing developers from writing portable code.

[6]It seems that the latest Orbix release (2.3) solves this problem

[7]Iona states that this function is CORBA compliant in the Orbix reference guide.

[8]Unless the resulting exception is of type `UnknownUserException`, which is not supported by Orbix.

5 Language Issues

This section presents miscellaneous problems encountered in the implementation of OGS, caused by lacks in the CORBA specification and in the C++ language mapping.

5.1 Server-Side Mapping

Although the CORBA specification suggests a C++ server-side mapping, it does not enforce implementations to comply with it. In particular, two models of skeletons are suggested: using inheritance and using delegation (*TIE* approach). Inheritance requires object implementations to inherit from an IDL-generated skeleton class, while delegation lets the skeleton forward requests to the object implementation, without requiring any inheritance relationship.

Orbix and VisiBroker support both models, but use a different syntax and terminology. For instance, with the inheritance model, the skeleton class of interface `M::I` is called `M::IBOAImpl` with Orbix, and `_sk_M::_sk_I` with VisiBroker. Writing portable code requires thus extensive usage of preprocessor macros.

Another problem we faced is the generation by Orbix of an extra parameter for each IDL operation, of type `CORBA::Environment`. This parameter is used principally for signaling exceptions to the client with compilers that do not support native C++ exceptions, and is allowed by the CORBA specification. Since VisiBroker does not add this parameter, we had to use conditional compilation for each IDL-defined operation. In fact, much of the server-side mapping and of the BOA is underspecified.

5.2 Scoped Names

The CORBA specification is loose concerning the C++ mapping of scoped names, which can map to namespaces, nested classes or concatenated identifiers. For instance, if an interface `I` is defined in a module `M`, an implementation can map `I` to a class in the `M` C++ namespace, or to a nested class in the `M` class (`M::I`), or to the unnested class `M_I`.

This lack of precision is meaningful for compilers that do not support the latest C++ features, but it makes user code depend on the implementation choice of ORB vendors, and prevents developers from writing code portable to ORBs that implement different mappings.

5.3 Multiple-inheritance of Implementations

The first Orbix version of OGS used implementation inheritance for derived IDL interfaces. The C++ implementation class was inheriting from both the implementation of its base IDL interface, and from its C++ skeleton class. This programming model works fine with Orbix, but is not fully supported by VisiBroker. We had then to change our implementation to use delegation instead of inheritance.

5.4 Object Equivalence

Another problem relates to the lack of clear semantics concerning object reference comparison, which is often required in a distributed application where several object references must be checked for equality. CORBA only specifies an operation, called `_is_equivalent()`, that compares two object references and returns `true` if they designate the same object, and `false` if the underlying ORB cannot determine whether the object references are equivalent or not, which generally happens when dealing with references from different ORBs; therefore, a compliant implementation can always return `false`, which is obviously useless. A workaround for this problem is to define an explicit operation on the objects that need to be compared, and to invoke it for each comparison; but this involves a remote invocation for each object reference comparison. In general, it is better to design applications so that they do not need to test for object equality.

5.5 Initial Services

CORBA applications can list available services using the `list_initial_services()` function, and they can obtain an initial reference to one of these services using the `resolve_initial_references()` function. This makes it easy for applications to use CORBA services in a portable way. Unfortunately, installing a new service in the list of initial services is implementation specific, and there is no way to add user-defined services. Therefore, getting an initial reference to OGS is not possible using these functions. A solution is to use the *Object Trader Service*, but very few implementations of this service are currently available. The solution we adopted is to simply register OGS in the Naming Service. Note that this problem is related to the service provider, and not to the application developer.

6 New CORBA Specifications

Some very recent OMG specifications solve part of the shortcomings mentioned in this paper. In particular, the new CORBA 2.2 specification adopted in early 1998 [OMG98] defines a new *Portable Object Adapter (POA)* meant to replace the much criticized BOA. This new adapter provides better support for writing portable server code. Asynchronous invocations with several qualities of service will be standardized in a new *Object Messaging Service* [SEI+98]. Unfortunately, these new specifications are unlikely to be included in commercial ORBs before several months[9].

This section briefly presents these new specifications, and explains how they can be used for solving most of the shortcomings described in this paper.

6.1 CORBA 2.2

The CORBA 2.2 specification [OMG98] provides major additions to the previous specification. It also solves many of the portability problems introduced by previous documents, often by stating that previous interfaces, while still legal for reasons of backwards compatibility, are not portable.

One of the major addition of the CORBA 2.2 specification is the introduction of a new object adapter: the *Portable Object Adapter (POA)*. As suggested by its names, one of the major design goal of this new adaptor is to allow programmers to construct object implementations that are portable between different ORBs. The CORBA 2.2 specification also specifies how the POA maps to programming languages. With the POA and the new mappings, it will be possible to develop fully portable servers.

The new POA standardizes threading models. The POA supports two models of threading when used in conjunction with multi-threaded ORB implementations; ORB controlled and single thread behavior. The two models can be used together or independently. The threading model associated with a POA is indicated upon POA creation using a `ThreadPolicy` object. The ORB interface also defines some new thread-related operations. Several operations, which can be used by single-threaded and multi-threaded applications, are included in the ORB interface. An application that is a pure ORB client would not need to use these operations.

[9]Experience shows that new specifications are not implemented in commercial ORBs before at least 1 year from the moment they are adopted by the OMG. Free ORBs tend to adopt new specifications before commercial ORBs.

The POA specification now states that object can be co-located (location transparency), and that invocations to local objects will be mediated by the ORB through the POA the same way as remote invocations. This will solve the problems described in Section 3.3.

The C++ mapping of the new CORBA specification mandates that exceptions can be inserted in and extracted from `Any` variables, thus solving the problem of Section ??.

The CORBA 2.2 specification also standardizes other features that are relevant to future implementations of OGS, although they are not directely related to problems presented in this paper.

The new POA adds support for objects with persistent identities. The POA is designed to allow programmers to build object implementations for objects whose lifetimes (from the perspective of a client holding a reference for such an object) span multiple server lifetimes. This will be very useful for providing recovery of crashed servers.

The POA also allows to associate a single DSI servant[10] with many CORBA objects. This is especially interesting for applications like OGS, since a single service object will be able to mediate requests for multiple replicated servers that support different interfaces.

Another addition of the CORBA 2.2 specification is dynamic management of `Any` values. If an `Any` is passed to a program that does not have any static information for the type of the `Any`, the object receiving the `Any` does not have a portable method of using it. The new specification enables traversal of the data value associated with an `Any` at runtime and extraction of its primitive constituents, as well as the construction of an `Any` at runtime without having static knowledge of its type. These facilities are especially useful for writing powerful generic services, and OGS will directly benefit of it (e.g., for filtering invocations, or for returning exceptions to clients).

Finally, the addition of `Interceptor` objects makes development of services much easier. An interceptor is an object interposed in the invocation (and response) path(s) between a client and a target object. Interceptors may be of two kinds: request-level and message-level, and several interceptors may be chained. OGS will directly benefit from interceptors since they permit services internal to the ORB to be cleanly separated so that OGS can coexist with other ORB services such as security.

[10]A *servant* corresponds to an object implementation.

6.2 Messaging Service

As of now, the Messaging Service is not yet standardized. We base the following description on the joint revised submission to the Messaging Service [SEI+98].

The Messaging specification will provide for two truly asynchronous method invocation models, based on *callback* and *polling*. In the first model, a callback object is registered at the time of the invocation. When the reply is available, that callback object is invoked with the data of the reply. In the polling model, the invocation returns an object which can be queried at any time to obtain the status of the outstanding request. It will be possible to issue asynchronous invocations using static interfaces, thus maintaining strong typing.

Whereas current CORBA specifications do not address the kinds of quality of service (QoS) features associated with messaging systems, the Messaging specification will provide for QoS such as delivery characteristics (lifetime of request and reply, reliability, scope of synchronization with target) and server-side queue management or ordering (temporal based, priority based). The Messaging QoS can be used to request a specific protocol as well as to guide the implicit protocol selection by the ORB. QoS can be independently specified at different levels: ORB, thread, object, or POA.

The Messaging specification will thus circumvent the shortcomings resulting from the lack of a truly asynchronous method invocation model in CORBA.

7 Conclusion

This paper points out some limitations in the portability of CORBA components. We have discussed these issues through our experience in implementing a CORBA Object Group Service (OGS) and then porting it from one ORB (Orbix) to another ORB (VisiBroker). OGS is highly concurrent and distributed, and its implementation relies on specific features such as asynchrony and dynamic invocations, for which CORBA does not provide a comprehensive specification. As a consequence, we could not develop a single portable implementation of OGS. From our experience, version 2.0 of the CORBA specification is not mature enough for developping truly portable applications. This is mainly due to the nature of CORBA: it is a specification only, and there is no reference implementation. While this is a big improvement over proprietary systems, it is also the cause of some problems in the early versions of the specification.

Many of the portability limitations of current implementations can be solved by the adoption of the new CORBA 2.2 specification and the future Messaging Service. Among these are server-side portability, multi-threading support, event management, and DSI support. It is important however to note that it will take some time before commercial ORBs implement the new specification and hence circumvent these limitations.

References

[FGG96] P. Felber, B. Garbinato, and R. Guerraoui. The design of a CORBA group communication service. In *Proceedings of the 15th IEEE Symposium on Reliable Distributed Systems*, pages 150–159, October 1996.

[FGS98] P. Felber, R. Guerraoui, and A. Schiper. The implementation of a CORBA object group service. *Theory and Practice of Object Systems*, 4(2):93–105, 1998.

[ION97] IONA. *Orbix 2.2 Programming Guide.* IONA Technologies Ltd., Mar 1997.

[OMG97] OMG. *CORBAservices: Common Object Services Specification.* OMG, 1997.

[OMG98] OMG. *The Common Object Request Broker: Architecture and Specification.* OMG, February 1998.

[SEI+98] BEA Systems, Expersoft, ICL, IONA, International Business Machines, Nortel, Novell, Oracle, PeerLogic, TIBCO, and Visigenic Software. *Asynchronous Messaging, Joint Revised Submission (orbos/98-01-04).* OMG, January 1998.

[Vis97] Visigenic. *VisiBroker C++ 3.0 Programmer's Guide.* Visigenic Software, Inc., Sep 1997.