A CORBA Object Group Service

Pascal Felber Rachid Guerraoui André Schiper

Ecole Polytechnique Fédérale de Lausanne

Département d'Informatique

CH-1015 Lausanne, Switzerland

Abstract

This paper describes an *Object Group Service* for the Common Object Request Broker Architecture (CORBA). The service enables to view a group of CORBA objects as a single entity despite concurrent invocations and failures, and thus provides an adequate support for the construction of highly available distributed applications with replicated critical components. The Object Group Service was designed and implemented in accordance with the Object Management Group guidelines. It is itself composed of a set of CORBA objects, and can transparently be used with any CORBA 2.0 multi-thread safe Object Request Broker.

1 Introduction

Increasing need and interest for distributed technologies have led to the emergence of several object-oriented distributed middlewares. One of the most important is the *Common Object Request Broker Architecture* (CORBA), specified by the *Object Management Group* (OMG) [OMG95a]. CORBA provides an object-oriented infrastructure that allows objects to communicate, regardless of the specific platforms and techniques used to implement these objects.

CORBA provides the basic mechanisms for remote object invocation through the *Object Request Broker* (ORB), as well as a set of services for object management, e.g. *Persistence Service* and *Transaction Service* [OMG95b]. Nevertheless, neither the ORB nor the existing services provide tools for building reliable and highly available applications. This can be considered a major drawback for many of today's applications such as process control and telecommunications.

The *group* paradigm was shown to be very powerful in supporting high availability through replication. Several systems have successfully experimented with the support of replication through group communication. Among these systems are Horus [RBC93], Isis [BCJ+90], Phoenix [Mal96], Totem [MMSA+96], and Transis [ADKM92]. Adding group communication to a CORBA environment is not straightforward. Some tentatives have been done in this direction [II94, Maf95, NMMS97], but as we will discuss later in this paper, the solutions adopted are proprietary and do not comply with the OMG approach.

In this paper, we present a way to provide group communication within a CORBA environment following the OMG *service approach*. This approach has the advantage of requiring no change to the underlying ORB. It is compliant with the CORBA specification, and not proprietary. Our *Object Group Service* is modular as it is itself designed and implemented as a set of CORBA objects. The use of the service can be made transparent to client objects, which may invoke a group of server objects (e.g. a group of replicas) as if it was a single object. The Object Group Service is now fully specified. It is implemented in C++ with Orbix [ION95], although it can directly be ported on any standard CORBA 2.0 multi-thread safe ORB.

The remainder of this paper is structured as follows. Section 2 recalls the CORBA model and the group paradigm, then summarizes some related work integrating groups to CORBA. Section 3 presents the architecture of our Object Group Service and Section 4 describes its implementation. Section 5 shows, with a simple example, how to use the Object Group Service in building a highly available application. Section 6 evaluates some design and implementation issues, and presents some performance measures. Finally, Section 7 presents the current status of the service, and the appendix gives its IDL interfaces.

2 Background

CORBA and group technologies have been two distinct research domains during the last few years. The integration of these technologies is a new and challenging task. This section briefly recalls some background on both technologies, and presents some related work.

2.1 CORBA

The *Object Management Architecture* (OMA) [OMG95a] is a conceptual infrastructure for building portable and interoperable[1] software components, based on open standard object-oriented interfaces. A client or server program is said to be *CORBA compliant* if it uses only the constructs described in the CORBA specification. An ORB implementation conforms to the specification if it correctly executes any CORBA compliant client or server program.

The Object Management Architecture

Figure 1 shows the five major parts of the OMA reference model. The *Object Request Broker* (ORB) enables objects to transparently invoke remote operations and receive replies in a distributed environment. The *Object Services* are a collection of interfaces and objects supporting basic functionalities useful for most

[1] *Portability* means here the ability to use an implementation with different ORBs (by simply recompiling it) while *interoperability* means the ability of an implementation to cooperate with other implementations.

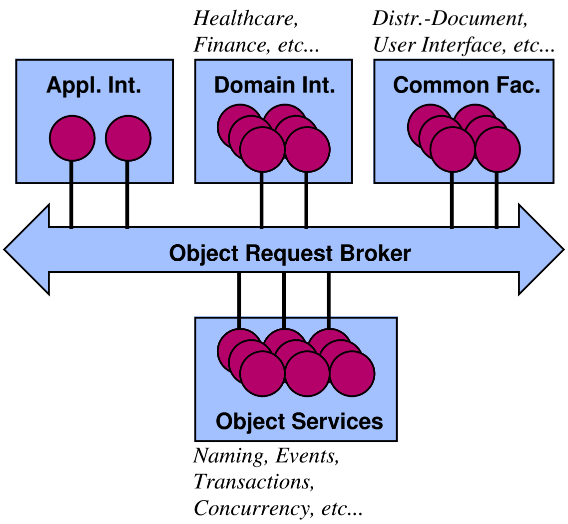


Figure 1: The OMA Architecture

CORBA applications. The *Common Facilities* are a collection of interfaces and objects providing end-user-oriented capabilities useful across many application domains. The *Domain Interfaces* are meant to be used only in specific vertical application domains. Finally, the *Application Objects* are objects specific to end-user applications.

The ORB and the Services

The Object Request Broker can be viewed as an "object bus". CORBA was designed to allow heterogeneous components to interoperate through this bus. Integration of distributed objects is available across platforms, regardless of networking transports and operating systems. Each component 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.

Beside the ORB itself, the CORBA services are of particular interest to us. A 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, and the Transaction Service that lets multiple distributed objects participate in atomic transactions.

2.2 The Group Paradigm

Groups have proved to be very useful for providing high availability through replication: a set of replicas constitutes a group, viewed by clients as a single entity in the system. We describe below the general group model adopted by most group-oriented systems (i.e. Horus [RBC93], Isis [BCJ+90], Phoenix [Mal96], Totem [MMSA+96], and Transis [ADKM92]).

Multicast and Membership

A group is a set of objects that are addressed as a single entity. The key mechanisms underlying the group paradigm are *group multicasts* and *dynamic group membership*. A well-known example of a group multicast primitive is the *total order multicast*, which ensures that the requests issued to a group are received by all the members of the group in the same order. Dynamic group membership allows to change the composition of the group at run-time (e.g. when objects crash or recover).

Figure 2 illustrates how a request from a client is sent to a group of servers. A gray circle represents an object. Clients know about the group through an abstract identifier, which is used to reference the group and to communicate with it. If two clients send the messages m_1 and m_2 to a group G using a *total order multicast* primitive, the underlying system ensures that both messages are received in the same order by all members of G, i.e. all (non crashed) members of G receive either message m_1 before m_2, or m_2 before m_1.

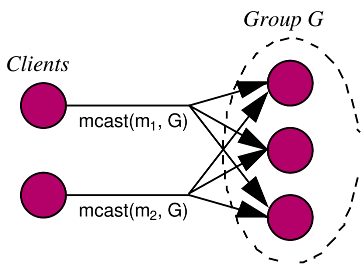


Figure 2: Group Communication

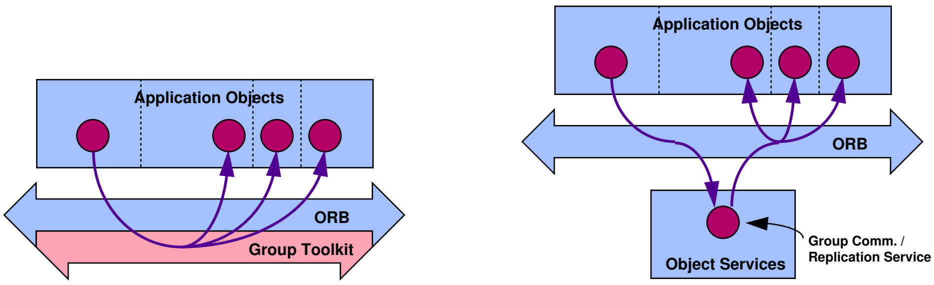
View Change and State Transfer

Group members are notified whenever a member joins or leaves the group so that each member knows the current composition of the group. This mechanism is called a *view change*. Since the members of a group usually share a common state, a new member has to receive the current state from the group when joining it. This is performed by a *state transfer* mechanism that transmits the state from a current member of the group to the new one. Hence, group members must provide operations for "getting" and "setting" their shared state.

2.3 Integrating Group Communication within CORBA: Related Work

In [FGG96], we discussed the problem of adding group communication to CORBA, and we distinguished two main approaches, illustrated in Figures 3(a) and 3(b): the *integration approach*, which consists in integrating an existing group communication system within an Object Request Broker (ORB), and the *service approach*, which provides group communication as a CORBA service beside the ORB. In that paper, we pointed out the advantages and drawbacks of each approach, and expressed our preference for the service approach.

There is a third approach, which we will not discuss in this paper, that consists in intercepting messages issued by an existing ORB and map them to a group communication system. Eternal [NMMS97] is based on this mechanism.



(a) Integration Approach

(b) Service Approach

Figure 3: Service vs. Integration Approach

The integration approach has been adopted in Orbix+Isis [II94] and Electra [Maf95]. Although appealing for its ease of development (there is no need to build a new group system from scratch) and its transparency (an object group is not distinguishable by a client from a singleton object that implements the same interface), this approach is not CORBA compliant, and results in proprietary systems. Orbix+Isis, Electra and Eternal are all based on heavyweight proprietary group communication toolkits (respectively Isis, Horus and Totem) which do not provide adequate primitives for group-to-group communication, and thus do not support client replication [2]. This implies that replicated CORBA objects are tied to a server role.

The service approach complies with the CORBA philosophy, by promoting interoperability and portability. It follows the design of the other functionalities that have been added to CORBA through services, such as persistence and transactions. These services were specified in IDL, and were adopted as CORBA standards. No existing CORBA service supports groups, and although the CORBA Event Service provides features for multi-objects communications, as we will show in Section 6, it cannot be used to support reliable applications. As pointed out earlier, an Object Group Service can be considered a fundamental building block for CORBA applications with reliability requirements.

[2] Eternal provides however a mechanism for client replication.

3 The Object Group Service

3.1 Overview

The *Object Group Service* (OGS) is an *object-oriented* environment providing building blocks for *reliable distributed* applications running in a *heterogeneous* system. The OGS does not require any change (or even extension) to the CORBA specification. It is portable in the sense that it can run on top of any standard CORBA 2.0 ORB, and does not rely on any ORB vendor-specific feature[3]. The OGS is modular as it is itself made of other CORBA (sub-)services, each built as a set of CORBA objects with IDL interfaces. None of these services is exclusive to group communication, and they can all be reused in very different contexts:

1. A *Messaging Service* that provides non-blocking reliable point-to-point channels.
2. A *Multicast Service* that provides reliable multicast communication.
3. A *Monitoring Service* that monitors objects to detect failures.
4. A *Consensus Service* used to guarantee atomicity and total order of multicast invocations.

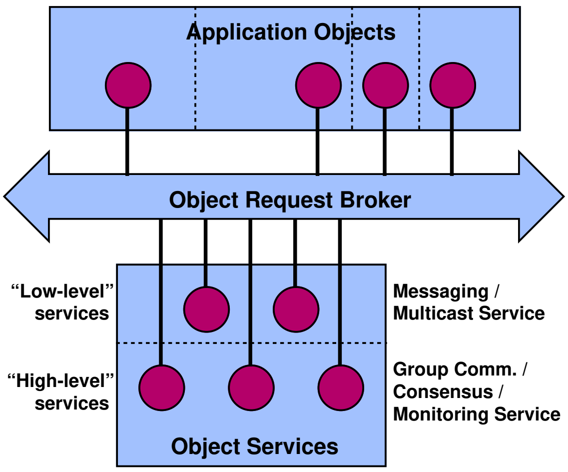


Figure 4: Architecture

Among these services, we distinguish low-level ones (1 and 2), the functionalities of which are close to the ORB, and high-level ones (3 and 4), that perform the group management and communication. Figure 4 illustrates the OGS architecture.

3.2 Description

The OGS manages groups of CORBA objects and provides primitives to communicate with these groups. Clients do not need to know the number, the identity, or the location of the members of a group. A client

[3]Note however that, one of our low-level services requires the ORB to be multi-thread safe (Section 4).

can issue a single request to all the group members at once, by naming the group. The OGS is inherently distributed, and does not depend on any global, critical, or centralized component. The OGS interfaces provide for different levels of reliability.

Invoking Object Groups

Similarly to the CORBA Event Service, the OGS provides two types of communication: *untyped* and *typed* communication. Untyped communication enables clients to send only values of type `any`[4] as messages. These messages are received by the servers through their `deliver()` operation. While this message-passing type interface is useful in some specific situations, it is generally more convenient for client 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 that defines the `makeDeposit()` operation, a client can directly invoke `makeDeposit()`; the OGS intercepts this call and invokes the `makeDeposit()` operation on each member of the group. Typed communication requires a specific binding phase executed by each client willing to get a reference to a group. Once it gets this reference, the client does not distinguish a singleton object implementing a given interface from a group of objects that implement the same interface. All members of the group should obviously support the same interface — this is the case when using group communication for object replication. For more advances features (e.g. to receive more than one reply resulting from a multicast invocation), the client has to invoke explicitly the OGS and hence loses the benefits of transparency.

OGS Views

The OGS provides several interfaces associated with the different views of a group: (1) the *client's view*, used to invoke the group, (2) the *member's view*, used by a group member to modify 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 the OGS to invoke operations on the members of a group.

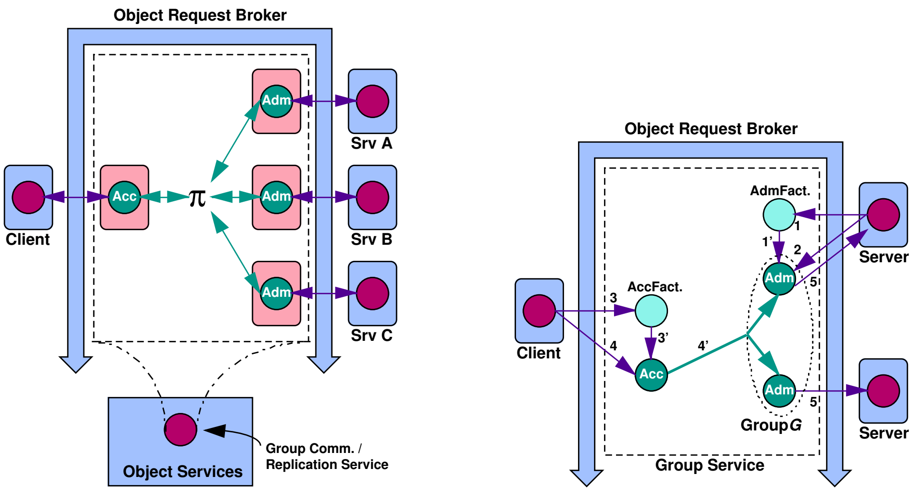
1. **Client's view** allows a client to get information about groups, to send multicasts to the members of a group, and to send messages to individual members. Groups are accessed by clients through an interface of type `GroupAccessor` which acts as a local representative for the group. `GroupAccessor` 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

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

`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 the service to issue callbacks to them. The `Groupable` interface defines operations for receiving messages, for view change notification (`view_change()`), and for state transfer (`get_state()` and `set_state()`).



(a) Architecture

(b) Interactions

Figure 5: Overview of the OGS

Interacting with the OGS

Figure 5(a) gives an abstract view of the interactions between clients, servers, and the OGS. The `GroupAccessor` object (Acc) acts as a client-side representative for the group, and is located in the same process (or on the same host) 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 every member receives the multicast, and that messages are ordered with respect to some criterion (specified by the client).

To join a group, a server first creates a `GroupAdministrator` object using a factory (1, 1') that returns a reference to the newly created object. Using this reference, the server invokes the `join()` operation on the group administrator (2). This launches a state transfer and a view change protocol. To multicast a message, a client first creates a `GroupAccessor` object using a factory (3, 3') that returns a reference to the newly created object. Using this reference, the client invokes the `multicast()` operation on the group accessor (4). This initiates the multicast protocol (4').

3.3 OGS Components

As we pointed out earlier, the OGS consists of a set of (sub-)services, each designed as an independent CORBA service. We overview below each of these services.

The Object Messaging Service

While a standard ORB only provides RPC-like communication primitives, the Object Messaging Service, illustrated in Figure 6(a), provides basic mechanisms for managing *asynchronous* point-to-point messages. It is composed of (1) interfaces that allow clients to invoke servers without blocking the client execution thread, and (2) interfaces that allow clients to specify the required quality of service for sending a message. Qualities of service include *unreliable* and *reliable* communication. With unreliable communication, there is no guarantee that the message will be received by the destination object (the message will be sent only once). With reliable communication, if both the sender and the receiver do not crash, the message is eventually received, assuming that any network failure that may occur is eventually repaired (the message is retransmitted until it is acknowledged).

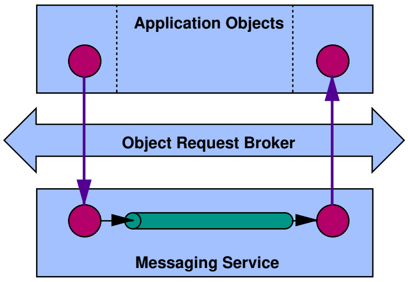
The Object Multicast Service

The *Object Multicast Service* provides a way to send a message to several CORBA objects. Figure 6(b) presents an abstract view of the service. This service provides interfaces for *unreliable* and *reliable* multicast. A reliable multicast ensures that all non-crashed objects eventually receive the message. The multicast primitives do not however provide ordering guarantees, such as total-order. These guarantees are implemented at a higher level using the Object Consensus Service. The Object Multicast Service performs remote communications through the Object Messaging Service.

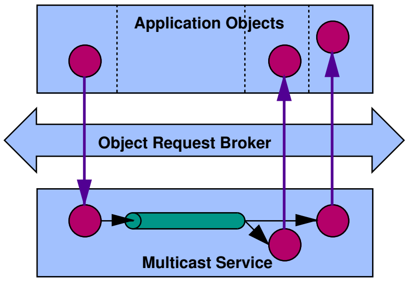
The Object Monitoring Service

Failure detection is necessary to detect unreachable or crashed objects, and is used in particular by the Object Consensus Service (see below). Failure detection does not need to be accurate, i.e. our protocols are correct despite unreliable failure detections [CT96].

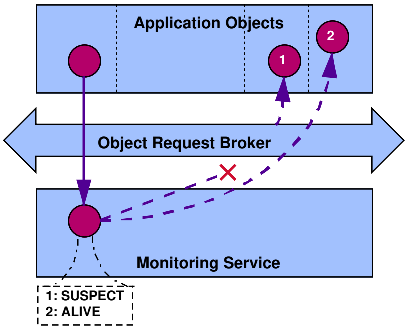
Failure detection is provided by the *Object Monitoring Service.* A failure detector monitors a set of objects in the system, and maintains a list of those that it currently suspects of having crashed. The Object Monitoring Service does not need to have a consistent global state; it is composed of several local failure detector modules that provide different informations depending on their location on the network. For instance, a network partition may lead failure detector modules located in different partitions to provide different informations. Figure 6(c) presents an abstract view of the Object Monitoring Service.



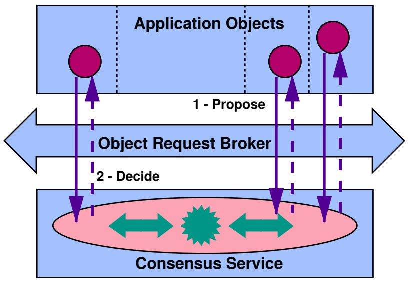
(a) Object Messaging Service



(b) Object Multicast Service



(c) Object Monitoring Service



(d) Object Consensus Service

Figure 6: Overview of the Services

The Object Consensus Service

Informally, a consensus protocol allows several processing elements to reach a common decision, according to their initial inputs, despite the crash of some of them. The consensus problem is a central abstraction for achieving fault-tolerance in distributed systems [GS96]. While most consensus-based environments use ad hoc protocols, it is natural in CORBA's object-oriented context to promote protocol reuse by defining a generic consensus service that can be used in various contexts.

The Object Consensus Service is used by the OGS to ensure that messages that are concurrently multicast by different clients are received by all the members of a group in the same order (i.e. the total order property). Figure 6(d) presents an abstract view of the Object Consensus Service. Participating objects first propose a value to the service (1), which leads to the execution of a distributed consensus protocol; eventually, (2) the service returns the decision to all participating objects.

4 Implementation issues

OGS has been implemented using Orbix. In this section, we do not give an exhaustive presentation of this implementation; for instance, we do not discuss the implementation of the Object Consensus Service, nor the way total order multicast is implemented using this service (see [GFG96] for a complete description). Instead, we present some implementation aspects that we believe to be important here, and that are not described in related publications. In particular, we focus on four fundamental issues: (1) group naming, i.e. the way we associate a symbolic name to a group, (2) typed invocation, i.e. the way we achieve object group transparency, (3) reliable non-blocking communication, i.e. the way we implement the Object Messaging Service, and (4) OGS configuration, in particular the way OGS objects are located in the system.

4.1 Naming Scheme

The OGS maps group names to the set of objects that compose the group. Group names are system-wide unique identifiers. Since group communication provides view change notifications, objects that are part of a group always know the current composition of their group. On the other hand, when an object wants to access a group of which it is not a member (e.g. to multicast a message or to join the group), the OGS must first obtain a reference to the members of the group.

For managing group names and compositions, we use the CORBA *Naming Service* (specified in [OMG95b]) and in particular IONA's implementation of this service. This implementation is robust enough, and is able to recover from failure. In the worst case, a failure of the Naming Service can hinder a new client or member to bind the group, but it will not disturb objects that are already bound to the group. An alternative would consist in using a fault-tolerant implementation of this service, like the one described in [Maf96].

Keeping the Naming Service up-to-date is a difficult task. The OGS updates the information in the Naming Service each time a view change occurs, but this information is not guaranteed to be completely up-to-date. In practice, when an object wants to access a group, the OGS contacts the Naming Service, retrieves the list of the group members, and contacts these members. It is very unlikely that all the objects listed in the Naming Service have left the group in the meantime; if it were the case nevertheless, the OGS would consult the Naming Service again until it succeeds in contacting some current member of the group.

4.2 Typed Communication

Typed communication is an important aspect of the OGS, since it provides group transparency to clients: once a client is bound to a group, it can issue standard invocations as if it were invoking a singleton server (see Section 3.2). Typed communication is achieved by the OGS using two advanced features of the CORBA specification: The *Dynamic Skeleton Interface* (DSI) and the *Dynamic Invocation Interface* (DII). The DSI is used by the service to accept requests that are actually aimed at the server interface. The DII is used to construct the invocations for the server interface. The next sections detail how these features have been used to provide client and server transparency.

General Principle

The approach uses in the OGS for implementing typed multicast communication is similar to the CORBA *request level bridging* [OMG95a]. 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).

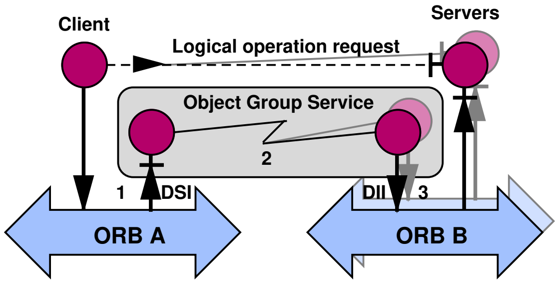


Figure 7: Service Implementation

The general principle for implementing typed communication, illustrated in Figure 7, is as follows: (1) The original request is passed to an OGS object in the client ORB: this object acts as a proxy for the servers. (2) The proxy object translates the request contents to an agreed format and issues a multicast to the server ORBs. (3) The OGS objects receive the multicast and invoke the required operation on the server objects. Any operation result is passed back to the client using point-to-point communications. In the following, we describe how steps (1) and (3) above are implemented. Step (2) is described later as it is related to the implementation of the Messaging Service.

Accepting Requests Using the DSI: the Client Side

The OGS gives the illusion to client objects that they are directly invoking a server, whereas they are actually invoking the service; this greatly simplifies client development. Indeed, the OGS can accept any operation of the server interface, although this interface is not known at compile time. This is achieved through the *Dynamic Skeleton Interface* (DSI).

The DSI allows a server to receive an operation invocation on any object, even if the IDL interface is not known at compile time. The client is not aware that the server is in fact implemented using the DSI; it issues standard invocations to IDL-defined operations. Using the DSI, the OGS gives the illusion to the client that it really implements the IDL interface of the server. The OGS intercepts invocations, transforms them into an agreed format, and multicasts them to all the group member objects.

Constructing Requests Using the DII: the Server Side

An IDL compiler generates the necessary support for clients to invoke remote objects. Using this approach, the IDL interfaces which a client program can use are determined when the client program is compiled. Unfortunately, this is too limiting since it does not make sense for the service to have any compile-time knowledge of the objects that will use it. To overcome this limitation, the CORBA specification defines the *Dynamic Invocation Interface* (DII) that allows an application to issue requests for any interface, even if this interface is unknown at the compile time.

It is important to notice that a server receiving an incoming invocation request does not know whether the client has used a static or a dynamic approach to compose the request. The OGS objects on server hosts receive the details of the request to be made, as part of the untyped multicast message[5] sent by the client. The message contains informations on the object that must be invoked, the operation name, the parameters, etc. The OGS translates this into a DII call leading to the invocation of the requested operation on the server interface.

4.3 Reliable Non-Blocking Communication

CORBA specifies only two types of invocations: two-ways invocations (synchronous or deferred synchronous invocations) and asynchronous one-way invocations (the operation does not return any value and the client does not wait for the invocation to complete). The semantics of these types of invocations are (intentionally) underspecified by the OMG, and two styles of execution semantics are defined [OMG95a]:

- **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 result and the client never synchronizes with the completion, if any, of the request.

The second execution style does not provide any reliability guarantee, since the ORB can just send the invocation message once or even discard it. Furthermore, the specification does not enforce request-only operations to be non-blocking; a compliant ORB can implement one-way asynchronous operations using transport mechanisms that can block the client. Asynchrony means here the client will not wait for the completion of the operation on the server side (which can last for 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*

[5]This message can be a CORBA `any`, a sequence of bytes, or some data in any format internal to the service.

that transports the remote invocation. Hence, a one-way call can block "forever" *at the transport level*, and this may in turn block the application.

These limitations have led us to the specification of the Object Messaging Service, that basically provides reliable non-blocking point-to-point communication. Since this service has standard IDL interfaces, the applications that use it are completely independent of how it is actually implemented. Our first prototype implementation was based on a custom communication library developed in our laboratory, called *Message Passing Library* (MPL). MPL provides connectionless reliable communication over UDP. It has a message-passing type interface, implements flow control, and provides non-blocking communication primitives with several qualities of service and comparable performances with TCP. In the meantime, we have developed another implementation that uses IONA Orbix one-way invocations. This implementation relies on the multi-threading capabilities of Orbix to ensure that the communication is non-blocking (we will come back to this issue in Section 6).

4.4 OGS Configuration

Whereas CORBA objects should be independent of their real location, some of our service objects have to be located on the client and server sites in order to provide the required reliability semantics. This is more a semantic than an architectural requirement, since our services may actually be installed anywhere. The best and most efficient strategy consists in locating these objects in the client and server processes, but it sometimes makes applications more difficult to program and to debug, and it may waste resources if several processes use the same service on the same host.

The current implementation of the OGS can run as a separate process on the client's and the server's host, or can be linked with the application code. The first approach has the advantage of decoupling the service from the application, enabling several applications to run on the same host (and hence use the same resources). The second approach is more efficient since inter-process communications are more expensive than invocations between objects located in the same address space.

5 Using the Object Group Service: Illustration

In order to illustrate the use of the OGS, we consider a simple banking application that manages replicated bank accounts. In our example, an account is an object with two attributes: `name` (name of the account's owner) and `balance` (current amount of money on the account); and two operations: `makeDeposit()` and `makeWithdrawal()`. To simplify, we do not include exception handling, and we assume that there is no overdraft limit on the accounts.

5.1 Defining an IDL Interface

The first step consists in defining an IDL interface for the server objects. Since the `Account` object is replicated, it is meant to be member of a group, and thus must inherit from the `Groupable` interface.

```
// IDL
interface Account {

  readonly attribute string name;
  readonly attribute float balance;

  void makeLodgement(in float f);
  void makeWithdrawal(in float f);
};
```

5.2 Implementing the Server

The `Account` implements the virtual operations inherited from the `Groupable` interface. It must also join a group at creation time. Since we want to use typed communication, the `deliver()` operation of the server object is empty. Part of the C++ code implementing the `Account` object is given below.

```
// Operations of Member

CORBA::Any *Account_i::deliver(const CORBA::Any& message) {
  cerr << "Should not be called" << endl;
}

void Account_i::view_change
    (const mGroupAccess::GroupView& newView) {
  cout << "Received a view change" << endl;
}

CORBA::Any *Account_i::get_state() {
  // Put the state into a CORBA Any
  CORBA::Any *a = new CORBA::Any;
  // ...
  *a <<= state;
  return a;
}

void Account_i::set_state(const CORBA::Any& a) {
  // Get the state from a CORBA Any
  if(a >>= state) {
    // ...
  }
}

// Operations of Account

CORBA::Float Account_i::name() {
  return CORBA::string_dup(name_);
}

CORBA::Float Account_i::balance() {
  return balance_;
}

void Account_i::makeDeposit(CORBA::Float f) {
  balance_ += f;
}

void Account_i::makeWithdrawal(CORBA::Float f) {
  balance_ -= f;
}

// Main

int main(int argc, char *argv[]) {
  // ...
  // Create server application objects
  Account_var serverVar = new Account_i();
  // Bind to local GroupAdministratorFactory
  mGroupAdmin::GroupAdministratorFactory_var fVar = ...;
  // Create a GroupAdministrator
  mGroupAdmin::GroupAdministrator_var aVar = fVar->create(argv[1]);
  // Join the group
  aVar->join_group(serverVar);
  // Wait for client requests
  orb.impl_is_ready();
  return 0;
}
```

5.3 Implementing a Client

The client's implementation consists in creating a representative object for the group, and *narrowing* it into an `Account` object reference (an alternative would have been to use untyped communication and to invoke directly the `multicast()` operation of the `GroupAccessor`). Part of the C++ code implementing the client is given below.

```
int main(int argc, char *argv[]) {
  // ...
  // Bind to local GroupAccessorFactory
  mGroupAccess::GroupAccessorFactory_var fVar = ...;
  // Create a typed GroupAccessor
  CORBA::Object_var taVar = fVar->create_typed(argv[1]);
```

```
  // Narrow it to Account
  Account_var aVar = Counter::_narrow(taVar);
  // Perform any operation on the account
  aVar->makeLodgement(50.95);
  cout << "Account " << aVar->name() << " has " << aVar->balance() << endl;
  return 0;
}
```

6 Discussion

This section evaluates some of our design and implementation choices. We first come back to the motivation of introducing a new service (i.e. the OGS). Then we evaluate our approach regarding interoperability, portability, and transparency issues. Finally, we give some performance measures based on the replication example above.

6.1 Why a New CORBA Service?

As we pointed out earlier in this paper, neither the ORB nor the existing CORBA services provide tools for building highly available applications. One might wonder however whether, instead of introducing a new service (i.e. the OGS), an existing service could not simply be extended to provide reliability guarantees.

At first glance, the CORBA Event Service appears a good candidate for this extension. It decouples the communication between suppliers and consumers through *event channels*. Suppliers can generate events without knowing the identity of the consumers. Conversely, consumers can receive events without knowing the identity of the suppliers. An event channel is a CORBA object with an IDL-defined interface, that acts both as a consumer and as a supplier of event data. The Event Service provides a tempting paradigm for multicast communication and an event channel can be used to model a group, with the members being consumers of the event channel. However, the Event Service does not provide any support for state transfer or dynamic group membership. Furthermore, it is based on a centralized architecture, where a channel is just another CORBA object, and this introduces a single point of failure[6].

Combined with the Persistence Service, the Transaction Service provides adequate support for object *crash recovery*. Nevertheless, crash recovery can only be considered sufficient for certain kind of applications. Applications with high availability requirements, like process control or telecommunications, cannot afford to wait for a critical component to recover. In fact, as specified by the OMG [OMG95b], the Transaction Service relies on the *Two Phase Commit protocol*, which is known to be blocking (i.e. does not ensure high availability) [BHG87].

[6]Note however that some implementations of the Event Service are not centralized. For instance, IONA's implementation (OrbixTalk) uses IP multicasts for efficient multicast communications. A channel is mapped to a specific IP address, which does not introduce a single point of failure. IP multicast is however not well suited for large scale communication.

6.2 Interoperability

In order to take advantage of the full interoperability of the CORBA architecture, remote communication should be implemented using the ORB communication primitives. Indeed, CORBA already provides normalized protocols (e.g. CORBA *Internet Inter-ORB Protocol*, IIOP), and the use of a separate communication channel might affect the interoperability of our service. All the OGS communication is encapsulated in the Object Messaging Service, which uses CORBA one-way invocations, and hence provides full interoperability.

6.3 CORBA Compliance

The OGS requires no extension to the current CORBA specifications. Furthermore, all the OGS components (except the Object Messaging Service) can be directly ported to any CORBA 2.0 compliant ORB. The Object Messaging Service requires however a multi-thread safe ORB (e.g. Orbix), or any ORB that provides non-blocking one-way invocations. With an ORB that does not support non-blocking invocations, the Object Messaging Service has to use either dedicated threads or a private transport mechanism. Note that in the latter case, we lose the interoperability feature provided by standardized protocols (e.g. IIOP); but this is not a problem since our Object Messaging Service is used only for internal communications.

The OGS consists of other CORBA (sub-)services, each built as a set of CORBA objects with IDL interfaces. None of these services is exclusive to the group paradigm, and they can all be reused in very different contexts. By encapsulating the multicast mechanism in a service with IDL-specified interfaces, we gain the ability to program portable applications that need efficient one-to-many communication primitives. For instance, an implementation of the Object Multicast Service may use network features such as IP-multicast to increase the throughput without hampering code portability. Furthermore, the Object Monitoring Service is used by the OGS to implement reliable group communication, but it can be used in a completely different context, e.g. for distributed application management.

6.4 Transparency

At some point, the OGS has to invoke operations on group member (server) objects. In particular, upon the occurrence of a view change, or when performing a state transfer, the OGS has to call back the group members. The group member objects inherit the adequate operations to be invoked in these situations from predefined IDL interface. The OGS is able to invoke any object that support an interface derived from this predefined base interface.

Interface inheritance is adequate when the set of operations that can be invoked on objects by the service is known a priori. Consequently, the format of the messages sent by a client to a server must be defined in the server's IDL interface. Although the OGS supports this model, by enforcing all messages to be `any` values, it provides the ability for clients to directly invoke operations of the server interface (typed communication).

Typed communication is an important aspect of the OGS, since it provides group transparency to clients: once a client is bound to an object group, it can issue standard invocations as if it were invoking a singleton server (see Section 3.2). The service objects on the server hosts receive the details of the request to be made as part of the multicast message sent by the client. The service translates this into a DII call that leads to the invocation of the requested operation on the server interface. The server also does not need to be aware that it is member of a group.

6.5 Performance Evaluation

Number of replicas	1	2	3	5
Unreliable - 1 reply	116.5	82.35	65.85	43.95
Unreliable - all replies	116.5	47.60	33.73	18.73
Reliable - 1 reply	100.0	38.70	40.72	21.10
Reliable - all replies	100.0	33.37	21.72	11.61
Total Order - 1 reply	40.76	17.59	12.62	9.21
Total Order - all replies	40.76	15.66	9.95	6.92

Table 1: Overhead of Replication (invocations/sec)

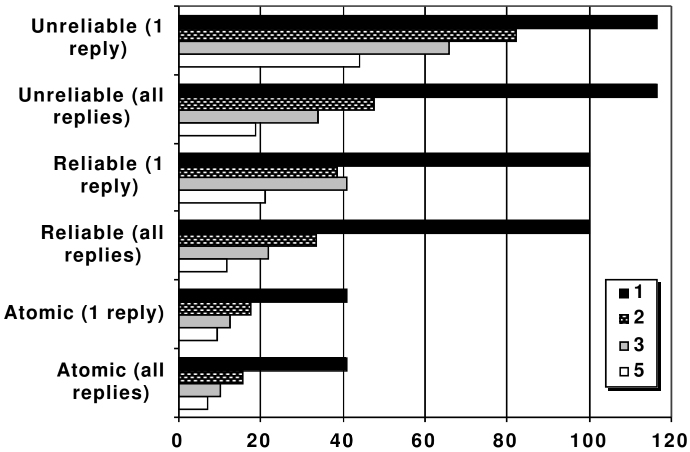


Figure 8: Overhead of Replication (invocations/sec)

We have performed some benchmarks with Orbix running on five Sun SPARCstations 20, with 64 MB of memory and a 8 KB socket queue size, on a local Ethernet network. The test application is based on the replication example of Section 5. We have computed the overhead of replication, by performing a sequence of invocations with 1, 2, 3 and 5 replicas, using different communication primitives: unreliable multicast, reliable multicast and total order multicast; and two models of synchrony: the client waits for the first reply, and the client waits for the replies from all members. The results, expressed in number of invocations per second, are given in Table 1 and in Figure 8.

We obtain good results with a single replica since no extra inter-process communication is required by group communication protocols. The bad performance of total order multicasts can be explained by the overhead of the distributed consensus algorithm used among the group members (up to know we did not consider any optimized decentralized consensus algorithm). As expected, we get better results when waiting only for the first reply than when waiting for all the replies; indeed, in the latter case, the client has to wait for the slowest client. Figure 8 shows that, surprisingly, when waiting only for the first reply, we get better results with three replicas than with two; this could be explained by the third machine having a low load and handling requests quicker than both others.

7 Current status

This work was performed in the context of *OpenDREAMS* (Open Distributed Reliable Environment, Architecture and Middleware for Supervision), an ongoing European ESPRIT project (EP20843), which aims at designing and implementing a CORBA compliant platform to support supervision and control applications. The need for a CORBA Object Group Service was motivated by the high availability requirements of such applications.

Our Object Group Service is fully specified, and our current implementation uses CORBA one-way invocations. It is currently being ported on top of IONA OrbixTalk product, which provides reliable communication based on IP multicast and should provide an efficient implementation support for the Object Multicast Service. We are currently working on code optimizations, and an enhanced implementation will soon be available for public release (more information can be found at `http://lsewww.epfl.ch/projects/`). In the specification of the CORBA services, the OMG mentions a future *Object Replication Service* [OMG95b]. We believe that our Object Group Service constitutes a good candidate for that service.

Acknowledgments

We are very grateful to our partners in the OpenDREAMS ESPRIT project for their fruitful collaboration, and especially Sean O'Sullivan from IONA Technologies, for his very useful comments on the design and implementation of our Object Group Service. We would also like to thank, for their help, Xavier Défago, Silvano Maffeis, Rui Oliveira, Bernhard Ruch, Doug Schmidt, and Steve Vinoski.

References

[ADKM92] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A communication sub-system for high availability. In *22nd Annual International Symposium on Fault-Tolerant Computing*, pages 76–84, July 1992.

[BCJ+90] K. Birman, R. Cooper, T. A. Joseph, K. Marzullo, M. Makpangou, K. Kane, F. Schmuck, and M. Wood. *The Isis System Manual*. Dept. of Computer Science, Cornell University, September 1990.

[BHG87] P.A Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[CT96] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996. A preliminary version appeared in the *Proceedings of the Tenth ACM Symposium on Principles of Distributed Computing*, pages 325–340. ACM Press, August 1991.

[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*, October 1996.

[GFG96] B. Garbinato, P. Felber, and R. Guerraoui. Protocol classes for designing reliable distributed environments. In *Proceedings of the 10th European Conference on Object Oriented Programming (ECOOP), Linz (Austria)*, number 1098 in Lecture Notes in Computer Science. Springer-Verlag, July 1996.

[GS96] R. Guerraoui and A. Schiper. Consensus service: a modular approach for building agreement protocols in distributed systems. In *IEEE 26th Int Symposium on Fault-Tolerant Computing (FTCS-26)*, pages 168–177, June 1996.

[II94] IONA and Isis. *An Introduction to Orbix+Isis*. IONA Technologies Ltd. and Isis Distributed Systems, Inc., 1994.

[ION95] IONA. *Orbix Programming Guide*. IONA Technologies Ltd., Nov 1995.

[Maf95] S. Maffeis. *Run-Time Support for Object-Oriented Distributed Programming*. PhD thesis, University of Zurich, February 1995.

[Maf96] S. Maffeis. A fault-tolerant CORBA name server. In *Proceedings of the 15th IEEE Symposium on Reliable Distributed Systems*, October 1996.

[Mal96] C. P. Malloth. *Conception and Implementation of a Toolkit for Building Fault-Tolerant Distributed Applications in Large Scale Networks*. PhD thesis, Swiss Federal Institute of Technology, Lausanne, September 1996.

[MMSA+96] L.E. Moser, P.M. Melliar-Smith, D.A. Agarwal, R. K. Budhia, and C. A. Lingley-Papadopoulos. Totem: A fault-tolerant multicast group communication system. *Communications of the ACM*, 39(4):54–63, April 1996.

[NMMS97] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Consistency of partitionable object groups in a CORBA framework. In *Proceedings of the 30th Hawaii International Conference on System Sciences*, pages 120–129, January 1997.

[OMG95a] OMG. *The Common Object Request Broker: Architecture and Specification*. OMG, 1995.

[OMG95b] OMG. *CORBAservices: Common Object Services Specification*. OMG, 1995.

[RBC93] R. Van Renesse, K. Birman, and R. Cooper. The HORUS system. Technical report, University of Cornell (NY), 1993.

Appendix: Object Group Service IDL Interface

```
// IDL
// Interface for clients

module mGroupAccess {

  typedef sequence<any> AnySeq;
  typedef string GroupRef;

  interface Invocable;

  struct GroupView {
    sequence<Invocable> members;
    unsigned long version;
  };

  exception GroupError { string description; };
    // Error while performing an operation on the group
  exception NoGroup {};
    // Trying to bind a non-existent group

  interface Invocable {
    oneway void deliver_oneway(in any message);
    any deliver(in any message);
  };

  interface GroupAccessor {
    void multicast_oneway(in any message)
      raises (GroupError);
    AnySeq multicast(in any message)
      raises (GroupError);
    GroupView get_view()
      raises (GroupError);
  };

  interface GroupAccessorFactory {
    GroupAccessor create(in GroupRef groupName)
      raises (GroupError, NoGroup);
    Object create_typed(in GroupRef groupName)
      raises (GroupError, NoGroup);
    void release(in GroupAccessor acc)
      raises (GroupError);
    void release_typed(in Object acc)
      raises (GroupError);
  };
};
```

```
// IDL
// Interface for replicated objects and service

#include "GroupAccess.idl"

module mGroupAdmin {

  exception NotMember {};
    // The object is not member of the group
  exception AlreadyMember {};
    // The object is already member of the group

  interface Groupable : mGroupAccess::Invocable {
    void view_change(in mGroupAccess::GroupView newView);
    any get_state();
    void set_state(in any state);
  };

  interface GroupAdministrator : mGroupAccess::GroupAccessor {
    void join_group(in Groupable member)
      raises(mGroupAccess::GroupError,
              AlreadyMember);
    void leave_group(in Groupable member)
      raises(mGroupAccess::GroupError,
             NotMember);
  };

  interface GroupAdministratorFactory {
    GroupAdministrator create(in mGroupAccess::GroupRef groupName)
      raises(mGroupAccess::GroupError);
    void release(in GroupAdministrator adm)
      raises(mGroupAccess::GroupError);
  };
};
```