The Design of a CORBA Group Communication Service *

Pascal Felber Benoît Garbinato Rachid Guerraoui
Ecole Polytechnique Fédérale de Lausanne
Département d'Informatique
CH-1015 Lausanne (Switzerland)

Abstract

The Common Object Request Broker Architecture (CORBA) is becoming a standard for distributed application middleware, and there are increasing needs for enriching the basic functionalities of CORBA. While mechanisms for persistence, transactions, event channels, etc., have been designed and specified for CORBA, no standard support is provided to handle object replication. In this paper we discuss the issue of augmenting CORBA with group communication, which is considered an adequate paradigm to handle replication. We distinguish two main approaches: the integration *approach and the* service *approach. We argue that the service approach is more appropriate to CORBA as it preserves the modularity of the architecture. We describe a proposal for a group communication service and discuss some implementation issues.*

1 Introduction

Increasing need and interest for distributed technologies have led to the emergence of several object-oriented distributed architectures. One of the most important of these middlewares is the *Common Object Request Broker Architecture (CORBA)*, specified by the *Object Management Group (OMG)* [OMG95a]. It provides an object-oriented infrastructure that allows objects to communicate, independent of the specific platforms and techniques used to implement these objects. CORBA defines *objects* as the unit of distribution, each object providing a set of operations through a single *interface*.

CORBA provides the basic mechanisms for remote invocation through the *Object Request Broker (ORB)*, as well as a set of services for object management, e.g. Persistence Service, Transaction Service, etc [OMG95b]. Nevertheless, neither the ORB nor the existing services provide tools for building reliable and highly available applications. This can be considered as a major drawback for many of today's industrial applications, which require a high degree of reliability and availability. Crashes or network failures are some of the problems that frequently occur and which the programmer has to deal with.

Group communication is considered a powerful paradigm to support fault-tolerance and high availability through replication. Several systems have successfully experimented with the support of reliable and highly available distributed applications through group communication, some of which have an object-oriented architecture. Among these systems are Horus [RBC93], Isis [BCJ+90], Phoenix [Mal96], Totem [AMMS+95], and Transis [ADKM92].

The goal of this paper is not to propose a new group communication protocol of its own, but rather to discuss how to extend a CORBA environment with groups. We present the two main approaches for adding group communication to CORBA: (1) the *integration* approach and (2) the *service* approach. The first approach consists of modifying and extending the Object Request Broker (ORB) with group communication mechanisms. This approach has been adopted in both Orbix+Isis [II94] and Electra [Maf95]. The second approach consists of providing group communication as a service on top of the ORB. We compare the two approaches and we argue that the service approach is a better solution, as it follows the CORBA philosophy by preserving the modularity of the architecture. We present the design of a group communication service compliant with CORBA specifications that makes the use of groups explicit, and we define the IDL interface of its basic components. Then we discuss some implementation issues in building group communication on top of an ORB such as Orbix [ION95].

The remainder of this paper is structured as follows. Section 2 recalls the Group Communication paradigm and the CORBA model. Section 3 presents and compares the two main strategies for adding group communication to a CORBA environment. Section 4 describes the design of a Group Communication Service. Section 5 discusses some implementation issues. Finally, Section 6 summarizes the paper's contributions.

*Research supported by OFES under contract number 95.0078, as part of the ESPRIT Project OpenDREAMS (number 20843).

2 Background

Object-orientation and group communication technologies have been two distinct research domains until the last few years. Integrations of these technologies is a new and challenging task. This section recalls some background on both technologies.

2.1 Group communication

The group communication paradigm allows the provision of high availability through *replication* in a straightforward way (by gathering a set of replicas into a single group). We describe below the general group model as adopted by most group-oriented systems (i.e. Horus, Isis, Phoenix, Totem, Transis). The key mechanisms underlying group communication are *group multicasts* and *dynamic group membership*. A well known example of a group multicast primitive is the *total order multicast*, which ensures that all requests sent to a group will be 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.

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 transparently transmits the current state from an "old" member of the group to the new one. Group members must thus provide operations for "getting" and "setting" their shared state. Group members are also notified whenever a member joins or leaves the group — possibly because of a crash — so that any member knows the current composition of the group. This mechanism is called a *view change*.

When communicating with groups, clients do not need to know the number of group members, nor how the members divide and perform their work. A group of replicas can thus be viewed as an abstract fault-tolerant entity[1].

Figure 1 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 (generally a symbolic name) which they use to reference the group and to communicate with it; they do not need to know the composition of the group, nor the localization of its members. If two clients send messages m_1 and m_2 to a group G using a *total order multicast*, the underlying system ensures that both messages are received in the same order by all members of G, i.e. either message m_1 is received before m_2, or m_2 is received before m_1 at all members of G.

[1] Transactions can also be used for fault-tolerant computing, but they do not provide high-availibility.

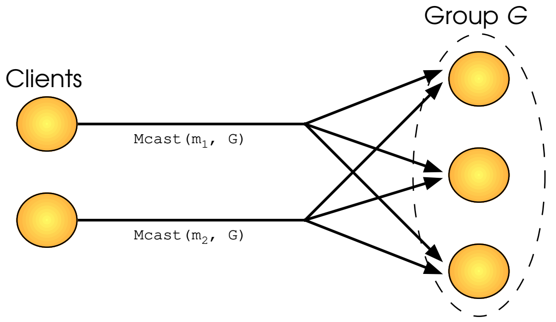

Figure 1. Group Communication

2.2 The CORBA model

The Object Request Broker can be viewed as an "object bus" which allows heterogeneous objects to communicate with each other. Integration of distributed objects is available across networks of systems, independent of networking transports and operating systems. CORBA was designed to allow heterogeneous components to interoperate through the object bus. Each component interface is specified in an *Interface Definition Language (IDL)* that is implementation independent. We summarize below some basic concepts of CORBA that are used in the remainder of the paper.

2.2.1 Basic infrastructure

By providing an object-oriented architecture with interface inheritance, ORB interoperability, and platform independence, CORBA meets some of the most essential requirements of modern computing, which are to maximize the portability, reusability, and interoperability of software[2].

Classical objects only live in a single program and the outside world does not know anything of them. In contrast, CORBA objects can live anywhere on a network. They can be accessed by local or remote clients via operation invocations. The language used to create server objects, and the system they are running on, are totally transparent to clients; the clients only need to know the interface published by the server objects, which acts as a "binding contract" between clients and servers.

Clients use *object references* to identify remote objects and to invoke operations on them. References are valid in the whole system and can thus be passed from one node to another. Furthermore, objects are not tied to a client or server role; they can act in turn as client and as server.

[2] Portability is the ability to use an implementation with different ORBs (by simply recompiling it). Interoperability is the ability of an implementation to cooperate with others.

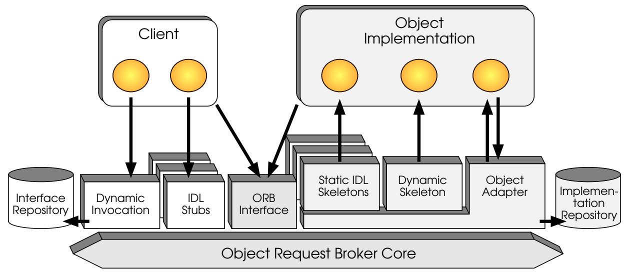

Figure 2. The CORBA Architecture

2.2.2 CORBA invocations

The CORBA architecture is depicted in Figure 2. The *client IDL stubs* provide the static interfaces to object services. They define how clients invoke corresponding services on remote servers. The *Dynamic Invocation Interface (DII)* lets the clients choose at run-time the operation invoked through a set of standard APIs. In contrast to the static stubs, the *DII* is independent of the target object's interface. The *server IDL stubs*, also called *skeletons*, provide static interfaces to each service exported by the servers. The *Dynamic Skeleton Interface (DSI)*, introduced in CORBA 2.0, provides a run-time binding mechanism for servers that do not have static IDL skeletons, allowing them to handle any request dynamically. The *Object Adapter* provides a run-time environment for instantiating server objects, passing requests to them, and assigning them *object references*. The object implementation may call the object adaptor and the ORB while processing a request. The *Implementation Repository* provides run-time information about the classes a server supports, the objects that are instantiated, and their identifiers.

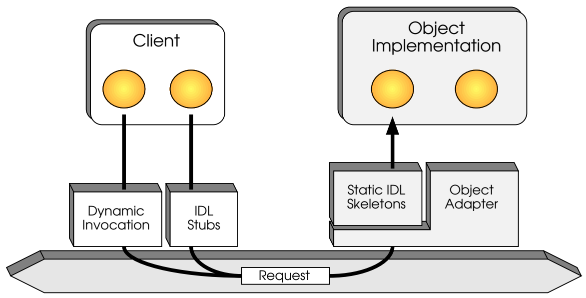

Figure 3. Performing a Request with CORBA

Figure 3 illustrates how a request is performed through the ORB, using either the *DII* or the *client IDL stubs*. Requests are handled at the server site by the *Object Adapter*, and handed to the server's *IDL skeleton*.

2.2.3 CORBA services

The basic communication platform provided by CORBA, that is the ORB, only supports minimal mechanisms for remote inter-object invocation. More specific functionalities (e.g. persistence) are added outside of the ORB as *CORBA services*. A service is basically a set of CORBA objects with their corresponding IDL interfaces, which can be invoked through the ORB by any CORBA client. The main characteristics of the objects composing a service is that they are not related to any specific application but are rather 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 for storing the objects on persistent storage, and the Transaction Service that lets multiple distributed objects participate in atomic transactions. A programmer can create an ordinary object and then make it recoverable, lockable, or persistent by having the object class multiply-inherit from the appropriate service.

3 Adding group communication to CORBA

We will now discuss the two main approaches for adding "object" group communication to a CORBA environment. The first one consists of integrating the notion of group within the ORB core, allowing a client to perform invocations to a group of objects as if it was a singleton object. The ORB performs the invocation to the replicated objects, gathers the replies, and returns them to the client. This approach provides transparency but requires a modification of the ORB core. The second approach consists of providing an external service that offers the basic primitives for managing groups of objects; the ORB knows nothing about groups. This approach is similar to the one adopted by the OMG to enhance CORBA with persistence, transactions, event channels, etc. Achieving complete group transparency is not straightforward with this approach, but the ORB is not modified and the modularity of the architecture is preserved. In the following, we compare the two approaches and argue in favor of the service approach.

3.1 Integration approach

The key idea of integration is that group communication is supported by the ORB core itself. We know of two ORBs which have adopted this approach, namely Orbix+Isis and Electra.

In this model, groups contain objects of the same type[3]. An object reference of type T is mapped to a group of objects of type T. The ORB is modified to distinguish references

[3] The type of a CORBA object is defined by its IDL interface.

to singleton objects from references to groups of objects. Figure 4 illustrates how a request is performed to a group of objects using *Orbix+Isis*. The request is given to the ORB, which recognizes that it is addressed to a group of objects and translates it into a call to an Isis multicast primitive. At the server sites, the relevant operation of each member is invoked by the ORB, which returns the result to Isis. Finally, replies are gathered at the client site, and returned to the client.

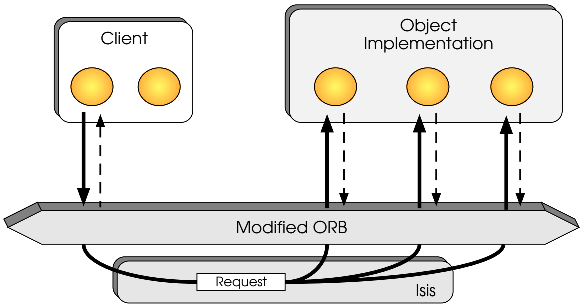

Figure 4. The Orbix+Isis Model

By default, only one of the replies is returned to the client in order to keep the group invocation transparent. However the client is also able to access all the replies, if it is aware of the group, and wants to take advantage of it. The basic idea is to generate two types of functions from IDL definitions: (1) standard functions that conform to the language mapping and (2) special functions with sequences of values for *out* and *inout* parameters. The client uses the function with the signature that corresponds to its needs. If the client is not aware of groups, only the standard functions will be used.

Electra [Maf95] and Orbix+Isis [II94] implement different variations of the integration approach.

- **Electra** extends the *Basic Object Adapter (BOA)*, adding the ability to *create*, *destroy*, *join* and *leave* a group, the notification of *view changes*, and the support for *state transfer*. Since the modification is performed at the BOA level, every CORBA object of the system can potentially be member of a group. All communication is performed using a toolkit that supports group communication (Isis or Horus). Electra has a modular architecture in the sense that it can use different underlying group communication systems. This model is "orthogonal" in the sense that all server objects are handled in the same way. However, it adds unnecessary complexity to objects that are not member of a group, and which only need BOA support.

- With **Orbix+Isis**, objects that can be group members have specific properties. Point-to-point communication goes through Orbix while multicasts are handled by Isis. To be a member of a group, an object must inherit from an abstract base class that extends the object with the notification of *view changes* and the support for *state transfer*. To handle object replication, the programmer can choose between two styles of object groups: *Active Replica* and *Event Stream*. The role of the server is fixed at compile time.

Another variation of the integration approach consists of providing a second object adapter in addition to the *Basic Object Adapter*, which is allowed by the CORBA specification. We would then have the basic adapter for standard objects and a dedicated adapter for member objects.

3.2 Service approach

The basic idea of the service approach is to provide group communication as a service on top of the ORB, and not as part of the ORB itself. The service "plugs" on the object bus, enhancing it with new functionalities. As shown in Figure 5, the *Group Communication Service (GCS)* appears as a new CORBA service, besides other services (e.g. transaction service and persistence service), and also besides any other CORBA object.

The graphical representation of the service in Figure 5 is completely independent of the effective location of the service. A service is viewed as a set of IDL interfaces, and it can be composed of several objects located at different nodes on the network. The service is inherently accessible from anywhere on the bus.

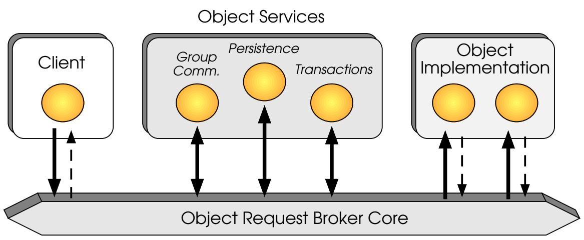

Figure 5. Using a Group Communication Service

Using a service makes groups explicit; one has to invoke the Group Communication Service to join groups, multicast messages, etc. For a client to send a request to a group, the client must invoke a specific *multicast* operation on an object of the service. It is the latter which is then in charge of multicasting the request (with the desired semantics, e.g. total order) and collecting the replies. However, as we will show in Section 5, groups can be hidden at the client side using smart proxies, which can be automatically generated from IDL definitions. On the server side, member objects are required to support a specific set of operations to enable the service to *call back* to them to invoke some operation (view changes, state transfer, etc.). We will detail the service approach in Sections 4 and 5.

3.3 Discussion

The main advantage of the integration approach versus the service approach is transparency. An object group is not distinguishable by a client from a singleton object that implements the same interface. Thus, a singleton object can easily be replaced by an object group to achieve high availability by simply recompiling its clients, as long as the latter exhibit a standard behavior (e.g. they do not need to access the complete set of replies resulting from an invocation to a group). This approach makes it possible to reuse existing clients with very few changes, and to let them access highly available components. In oder to achieve a similar level of transparency in the service approach, the usage of smart proxies is necessary. Any of the two approaches require server objects (i.e. group members) to deal with replication.

Another advantage of the intergration approach is that an implementation can be very efficient since group communication is performed directly by the ORB. Multicasting a request does not involve an intermediary object as in the service approach.

The cost of transparency is that the semantics of CORBA object references are modified. The *Object Model* of the CORBA specification defines an object reference as an *object name that reliably denotes a particular object. An object reference indentifies the same object each time the reference is used in a request, and an object may be denoted by multiple, distinct references* [OMG95a]. With the integration approach, an object reference does not uniquely identify a single CORBA object on the network, but (possibly) a group of objects. Furthermore, the objects referenced can change over time (the reference can even designate an empty group).

The other drawbacks of the integration approach are the loss of portability (implementation is strongly ORB-dependent) and interoperability (both clients and servers have to use the same ORB implementation, which relies on proprietary group communication mechanisms[4]).

The advantages of the service approach over the integration approach mainly concerns portability and interoperability. A service is not ORB-dependent (it is defined only in terms of an IDL interface), and by using the ORB communication primitives it benefits from the ORB's interoperability properties. It requires no modification of the ORB and is easily portable from one ORB to another.

A service offers a more powerful and flexible model since it makes the use of groups explicit, and therefore does not rely on a set of predefined behaviors (e.g. *Active Replica* execution style). It also makes it possible to group objects that support different IDL interfaces. Finally, multiple services can cooperate in the system, e.g. the *GCS* can use a Recovery Service to recover server objects from crash failures, or the Persistent Object Service to manage persistent replicas.

We believe that the service approach is more appropriate to a CORBA environment as it follows the OMG "philosophy": that is, the ORB should provide a minimal mechanism to break the boundaries of languages and address spaces, allowing heterogeneous distributed objects to interoperate. All additional functionality should be provided as services, i.e. specific CORBA objects with IDL interfaces. For instance, concurrency or persistence are needed by many applications, and some programming languages address these issues, but the OMG did not include any support for them in the ORB core. They rather defined external services, keeping the ORB core as simple as possible with the necessary tools common to all CORBA applications. Finally, a service is much more likely to be standardized than an ORB extension (or even a new object adapter).

In the next section, we present the design of a Group Communication Service with a simplified version of its interfaces and we discuss some issues concerning the implementation of this service.

4 The Group Communication Service

4.1 The design of a CORBA service

A CORBA service is most accurately viewed as a set of *IDL (Interface Description Language)* interfaces. It is generally decomposed into several distinct interfaces that provide different views[5] for different types of clients. It can be implemented as a collection of distinct objects that cooperate to provide the complete service, and which are located at different nodes on the network. This is typically the case for our Group Communication Service.

In addition to standard object invocation, services can use *interface inheritance* and *object factories* to provide powerful and reusable services.

- Interface inheritance is useful for defining a set of operations that a client is required to support in order to enable a service issue callbacks; it allows server objects to invoke some operation of the client and pass data asynchronously to it.

[4]Interoperability is generally achieved by using normalized protocols like the *Internet Inter-ORB Protocol (IIOP)* specified in the CORBA 2.0 specification. It is possible to provide an IIOP gateway for an ORB that uses a group communication subsystem, but the gateway represents a single point of failure.

[5]To avoid the confusion between the *view* that a client has from a service, and the *view* (i.e. the composition) of a group provided by the group communication system, we use the term *group view* in the second case.

- Object factories are useful for creating or obtaining references to (possibly remote) objects that support a basic set of operations, without using language-specific constructs. It also allows to pass arguments to an object at creation time, overcoming the lack of constructors in IDL.

4.2 Service structure

The Group Communication Service is intended to work in a distributed environment, and therefore its design does not depend on any global, critical, or centralized service. Clients can send messages to groups of server objects without knowing their identity. Clients and servers support standard IDL interfaces; no extension to CORBA is necessary to define these interfaces. Currently, the GCS only provides for untyped *oneway* communication. The *request/reply* model can easily be implemented on top of the *oneway* model.

When designing the GCS, three families of objects must be distinguished: (1) member objects, (2) service objects, and (3) factory objects.

1. **Member objects** are gathered into a group and have to react to events such as message delivery, view change and state transfer. They must be provided by the programmer and they inherit from generic interfaces that define the operations which the service invokes when one of these events occurs: `deliver()`, `view_change()`, `get_state()`, etc.
2. **Service objects** are associated with a group, and are used to interact with the group. A group is not a CORBA object in itself; it has no localization since it is inherently distributed. Furthermore, we cannot rely on a single centralized object as an intermediary between the clients and the members of a group, since this would compromise fault-tolerance and would constitute a performance bottleneck. Therefore, we use specialized objects as local representative for groups. These service objects run the protocols necessary for message delivery and group management.
3. **Factory objects** facilitate the creation of service objects that are associated to groups at creation time. Providing factory objects with IDL interfaces enhances reuse by being independent of the programming language for creating CORBA objects.

The classes of these three families of objects, as well as their relationships, are shown in Figure 6. They are detailed in the next sections.

4.3 Service interfaces

The Group Communication Service provides several interfaces associated with the different views that objects have

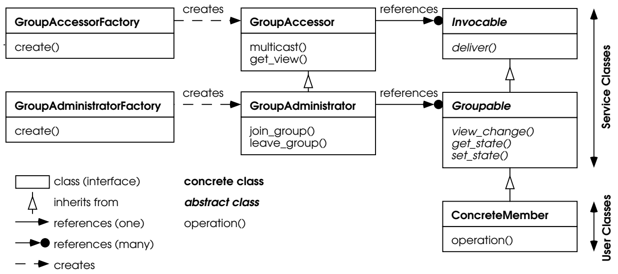

Figure 6. Classes of objects defined by the Group Communication Service

from each other: (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 service to invoke the members of a group.

1. **Client's view** allows a client to get information about groups, to perform requests on them, and to send point-to-point 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 by invoking the `create()` operation on a `GroupAccessorFactory` object, giving it the name of the group as parameter. A `GroupAccessor` object is dedicated to a single group, so that clients must create a distinct object for each group they want to access. The `GroupAccessor` interface defines an operation for multicasting messages to the group it represents (`multicast()`). This interface also defines an operation for getting the current composition of the group (`get_view()`).

```
// IDL
module mGroupAccess {
  // ...
  interface GroupAccessor {
    void multicast(in any message)
      raises (GroupError);
    GroupView get_view()
      raises (GroupError);
  };

  interface GroupAccessorFactory {
    GroupAccessor create(in string groupName)
      raises (GroupError);
  };
  // ...
};
```

GroupView is a structure[6] that contains a list of references to Invocable objects and a version number, which allows to identify outdated group views. An Invocable object is basically an object that can receive a message, i.e. that can be *invoked* through the untyped communication mechanism of the GCS. Once a client gets a reference to an object that supports this interface (e.g. by invoking the get_view() operation of a GroupAccessor object), it can also send point-to-point messages to the associated objects using the regular CORBA invocation mechanisms. Although this type of communication is less useful for clients than multicasts, it increases the system's flexibility especially when using a primary backup replication strategy.

```
// IDL
module mGroupAccess {
  // ...
  interface Invocable;

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

  interface Invocable {
    void deliver(in any message);
  };
  // ...
};
```

In the scheme described above, the group notion is obviously not transparent to clients. However, such transparency can be obtained through the use of smart proxies at the client side. We will discuss the use of this feature in Section 5.3.

2. **Member's view** is a superset of the client's view, and is defined by the GroupAdministrator interface. The GroupAdministrator interface inherits from the GroupAccessor interface and is also dedicated to a single group. Objects can join and leave groups using the join_group() and leave_group() operations of GroupAdministrator objects. GroupAdministrator objects are created by invoking the create() operation on a GroupAdministratorFactory object, passing it the name of the group as parameter.

```
// IDL
module mGroupAdmin {
  // ...
  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 string groupName)
      raises (mGroupAccess::GroupError);
  };
  // ...
};
```

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 inherits from the Invocable interface, and consequently from its ability to receive messages. In addition, it defines operations for view change notification (view_change()) and for state transfer (get_state() and set_state()). The Group Communication Service obtains a reference to Groupable objects as a parameter of the join() operation. Upon a view change, the service invokes the view_change() operation of the Groupable objects, passing them a parameter of type GroupView. This parameter contains a sequence of references to objects that are part of the new group view, but seen as Invocable objects rather than as Groupable objects; this ensures that operations view_change(), get_state() and set_state() will not be invoked by objects other than the Group Communication Service.

```
// IDL
module mGroupAdmin {
  // ...
  interface Groupable : mGroupAccess::Invocable {
    void  view_change(
            in mGroupAccess::GroupView newView);
    any   get_state();
    void  set_state(in any state);
  };
  // ...
};
```

4.4 Interaction between objects of the service

Figure 7 shows the different types of objects and interfaces involved in the Group Communication Service. An arrow with a vertical bar is used to show that the target object supports the interface named next to the arrow, and that clients holding an object reference of this type can invoke the operations defined by the interface. The arrow points from the client to the group member. The service is represented by a blob (misshapen circle), which delimits a conspiracy of one or more objects. For clarification, we have represented two different hosts for the client of the service and the group member. The service itself is partly located on both sites. As depicted in the figure, clients of the

[6] Note that a CORBA structure is not an object, and can thus be passed by value.

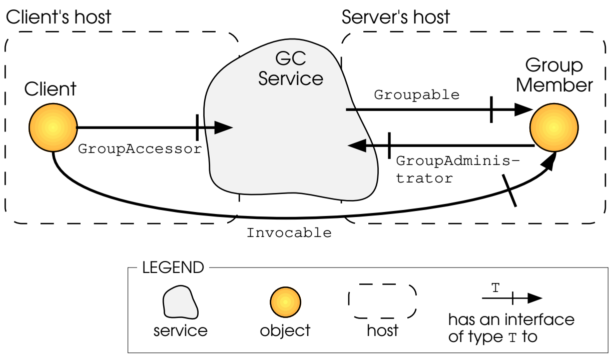

Figure 7. Structure of the Service

service holds an interface of type `GroupAccessor` to the service, while the group member hold an interface of type `GroupAdministrator`. The group member exports two interfaces, one for the service (`Groupable`), and the other for all other objects (`Invocable`).

Figure 8 and 9 illustrate the interactions between objects involved in the Group Communication Service. A box represents an operation active on an object, whose type is indicated on top of each vertical line. Time flows from the top to the bottom. Plain arrows represent invocations, dashed arrows represent the creation of a new object, and double-headed dashed arrows between service objects represent service-specific protocol messages.

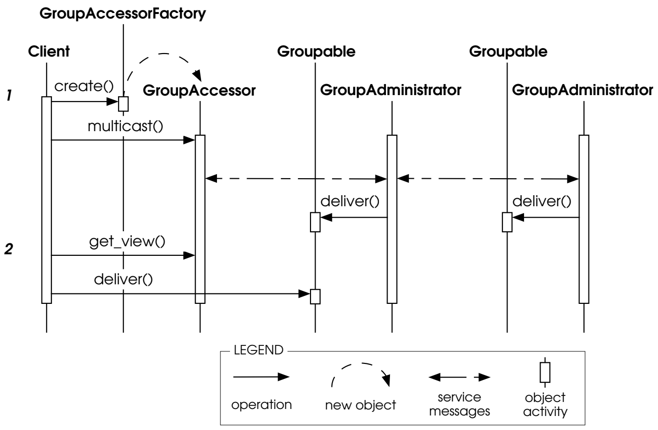

Figure 8. Interaction between two Groupable Objects

Figure 8 shows the interaction between a client and a group. We present two types of interactions: (1) a multicast to the group, and (2) a point-to-point communication with a group member. We assume the group contains two members, i.e. two `Groupable` objects. To perform the multicast, the client first creates a `GroupAccessor` object (used for communication) by invoking the `create()` operation on a `GroupAccessorFactory` object. Then, the multicast is performed using this `GroupAccessor` object (1). To perform a point-to-point communication, the client gets a reference to the `Groupable` objects by invoking the `get_view()` operation. Then the client sends a point-to-point `deliver()` message using a standard CORBA invocation (2).

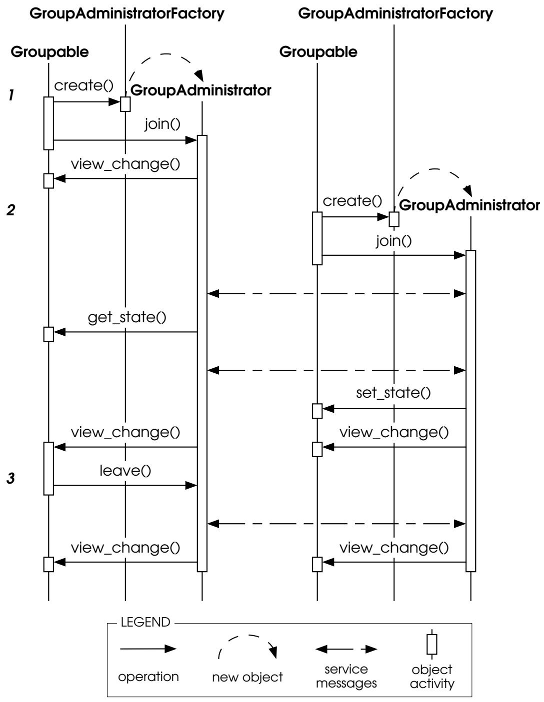

Figure 9. Interaction between a Client and a Groupable Object

Figure 9 shows how `Groupable` objects join and leave a group, and how view changes and state transfers are handled. A first `Groupable` object creates a `GroupAdministrator` object and joins an empty group, which leads to a view change (1). A second object also joins, but since the group is no longer empty, the state transfer protocol is launched before the group view is given to the members (2). Finally, the first object leaves the group, inducing another view change (3). If this object had crashed, the same view change would have been performed.

5 Implementation issues

In this section we discuss some of the implementation choices we are confronted with, when building the Group Communication Service. We discuss these choices with respect to their impact on interoperability, performance, and transparency.

5.1 Interoperability

In order to take advantage of the full interoperability of the CORBA architecture, the remote communication of the GCS should be implemented using the ORB communication primitives. Indeed, CORBA already provides normalized protocols (e.g. *Internet Inter-ORB Protocol (IIOP)*), and the use of a separate communication channel could affect the interoperability of our service. The CORBA specification does not forbid to use a specific group communication toolkit and to provide an IIOP gateway that knows about groups, but the gateway would be a single point of failure and the benefits of group communication would be compromised.

Note, that the service approach does not enforce a communication policy. The service can be implemented on top of a group communication toolkit, as well as on top of the ORB communication mechanisms. But the latter seems to be an interesting solution. For instance, reliable multicast communication can be implemented using CORBA's non-blocking asynchronous invocations. A potential drawback is performance, which could suffer from this implementation choice as shown in the next subsection.

5.2 Performance

The service design principles published by the OMG [OMG95b] recommend to assume good and efficient ORB implementations when designing a service. Nevertheless, we have performed some benchmarks to compare communication time with raw TCP and with the Orbix ORB. The tests have been run on two Sun SPARCstations 20, with 64 MB (megabyte) of memory and a 8 kB (kilobyte) socket queue size, on a local Ethernet network. Two different tests have been performed: the first one uses sequences of octets as invocation parameter whereas the second one uses a string parameter.

The test program has been adapted from the widely available `ttcp` protocol benchmarking tool which measures end-to-end data throughput between two remote processes [SHAS95]. Values are given in megabits per second (Mbps) and are calculated by taking the mean of 20 executions. The size of the messages in the test was between 1 kB and 64 kB. Results are shown in Table 1 and Figure 10.

	1Kb	2Kb	4Kb	8Kb	16Kb	32Kb	64Kb
TCP	3.948	5.246	6.451	7.2	7.784	8.028	8.153
Orbix (seq.)	1.98	2.966	4.07	5.438	6.476	7.134	7.46
Orbix (str.)	2.118	3.206	4.424	5.448	6.106	6.646	6.97

Table 1. Communication Time Results (in Mbps)

Analysis of Table 1 shows that the raw TCP version runs faster that Orbix, especially with small packets where TCP is nearly twice as fast as Orbix. With big messages, the

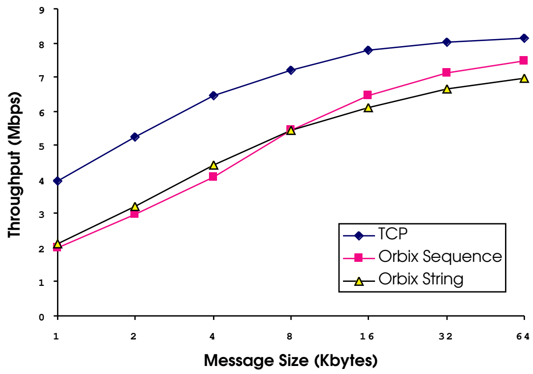

Figure 10. Comparing Communication Time

results are much closer. This can be easily explained by the fixed cost of demultiplexing and memory management in the CORBA implementation. The TCP version directly reads the data stream on a socket while the CORBA version analyses the data, converts it into the expected type (which involves copying) and directs it to the destination object. Since these costs increase only slightly with message size, the difference between the TCP and the CORBA implementation becomes smaller as the message size grows.

Although TCP is used by many group communication systems, some of them rely on custom communication protocols based on UDP and IP multicasts, which should lead to better results. A more exhaustive performance analysis should take this aspect into account, and conduct additional experiments with more than two machines.

5.3 Transparency

Transparency can be achieved by using smart proxies at the client side. Proxies are local representatives for remote objects which hide distribution from clients. Smart proxies are extensions to proxies that perform additional tasks, for instance implementation-specific optimizations; they can be used to cache data at the client site to avoid unnecessary communication. In our case they are used to hide group communications, giving the client the illusion that it communicates with a singleton object.

Although smart proxies are language-dependent — they are made of code — they can be programmed in an ORB-independent way. Furthermore, they can be automatically generated from IDL definitions, freeing the programmer from repetitive and tedious work. A smart proxy must be created for each member object (derived form the `Groupable` interface). A possible implementation of a smart proxy is given in an extended version of this paper [FGG96].

6 Conclusion

The addition of group communication support to CORBA yields an environment that promotes reliability and high availability, as well as interoperability and portability.

In this paper we have discussed the problem of adding group communication to CORBA, focusing on two main approaches: the integration approach which consists of integrating group communications within the Object Request Broker (ORB) core, and the service approach which provides group communication as a service besides the ORB. We have pointed out the advantages and drawbacks of each approach, and we have expressed our preference for the service approach. This approach follows 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, which were specified in the OMG's Interface Description Language (IDL) and adopted as CORBA standards. Furthermore, in the specification of the CORBA services, the OMG mentions a future *Replication Service* intended to provide explicit replication of objects and consistency of replicated copies [OMG95b]. A Group Communication Service constitutes a fundamental building block to support replication.

We have presented the design of a Group Communication Service and we have described its various components. Then we have suggested an IDL specification for its interface. Finally, we have discussed some implementation issues with respect to interoperability, performance and transparency.

Our research effort is embedded in the *OpenDreams* Esprit project, which aims at providing a CORBA compliant platform for reliable industrial applications. We are currently implementing a C++ version of our Group Communication Service, on top of the ORBIX platform. Our service implementation is inspired by the BAST class library [GFG96], using the generic consensus protocol described in [GS96].

References

[ADKM92] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: a communication sub-system for high availability. In *Proceedings of the 22nd International Symposium on Fault Tolerant Computing Systems*, 1992.

[AMMS+95] Y. Amir, L.E. Moser, P.M. Melliar-Smith, D.A. Agarwal, and P.Ciarfella. The totem single-ring ordering and membership protocol. *ACM Transactions on Computer Systems*, 13(4):311–342, November 1995.

[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.

[FGG96] P. Felber, B. Garbinato, and R. Guerraoui. The design of a CORBA group communication service. Technical report, Ecole Polytechnique Fédérale de Lausanne, 1996.

[GFG96] B. Garbinato, P. Felber, and R. Guerraoui. Protocol classes for designing reliable distributed environments. In *Proceedings of European Conference on Object Oriented Programming (ECOOP), Linz (Austria)*. Springer-Verlag, July 1996.

[GS96] R. Guerraoui and A. Schiper. Consensus service: a modular approach for building agreement protocols in distributed systems. In *Proceedings of the 26th International Symposium on Fault-Tolerant Computing Systems, Sendai (Japan)*, 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 (Switzerland), February 1995.

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

[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.

[SHAS95] D. C. Schmidt, T. Harrison, and E. Al-Shaer. Object-oriented components for high-speed network programming. In *Proceedings of the 1st Conference on Object-Oriented Technologies and Systems, Monterey*, June 1995.