The Implementation of a CORBA Object Group Service*

Pascal Felber, Rachid Guerraoui, and André Schiper
Ecole Polytechnique Fédérale de Lausanne, Département d'Informatique, CH-1015 Lausanne, Switzerland

The *Object Group Service* (OGS) extends CORBA with the ability to group objects and invoke them as a single entity. Through this abstraction, OGS provides an adequate support for the construction of highly available distributed applications with replicated critical components. OGS was designed and implemented in accordance with the Object Management Group guidelines. It does not rely on any Object Request Broker vendor-specific feature and can transparently be used with any CORBA 2.0 Object Request Broker. It is itself made of several CORBA services such as a monitoring service and a consensus service which can themselves be used as stand-alone CORBA services.

1. Introduction

The *Common Object Request Broker Architecture* (CORBA), specified by the *Object Management Group* (OMG), 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., *Naming Service*, *Transaction Service*, and *Event Service* [14]. Nevertheless, neither the ORB nor the existing services provide tools for building highly available applications. This can be considered a major limitation for the use of CORBA in many of today's applications such as finance, process control and telecommunications.

To overcome this limitation, we have designed and implemented an *Object Group Service* (OGS), which provides facilities for CORBA object group communication. The *group* paradigm is very powerful in supporting reliability and high availability through replication: a set of replicas constitutes a group, viewed by clients as a single entity [1]. Through the group abstraction, the failure of a replica is made transparent to the client; in addition, read-only accesses to a replicated object can transparently be performed through the closest replica. 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). 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 state.

Adding the group abstraction and related mechanisms to a CORBA environment is not straightforward. Some tentatives have been made in this direction [10, 11, 12], but as we discussed in [4], most of the solutions adopted are proprietary and do not comply with the OMG approach. We have designed and implemented our *Object Group Service (OGS)* following the OMG service approach for extending the basic CORBA functionalities. This approach has the advantage of requiring no change to the underlying ORB: it is compliant with the CORBA specification and not proprietary. OGS is modular as it is itself designed and implemented as a set of CORBA sub-services which can themselves be used in a stand-alone way. Among those sub-services are a monitoring service, a consensus service and a multicast service. Using advanced CORBA features such as dynamic requests, we implemented OGS in such a way that its use can be made transparent to client objects, which may invoke a group of server objects (e.g., a group of replicas) as if it were a single object.

This paper describes the architecture and implementation of OGS. Section 2 motivates a service approach in extending CORBA with object group communication. Section 3 describes OGS architecture and in particular the sub-services involved in OGS. Section 4 presents the various object invocation semantics provided in OGS

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

and points out some of their implementation features. Section 5 discusses OGS configuration issues such as naming and localization. Section 6 presents a running scenario of OGS. Section 7 gives some performance figures. Finally, Section 8 summarizes the main characteristics of OGS and discusses its availability.

2. The OGS Approach

2.1. The Object Management Architecture: Background

The *Object Management Architecture* (OMA) [13] is a conceptual infrastructure for building portable and interoperable software components, based on open standard object-oriented interfaces. 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.

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 CORBA specification if it correctly executes any CORBA compliant client or server program.

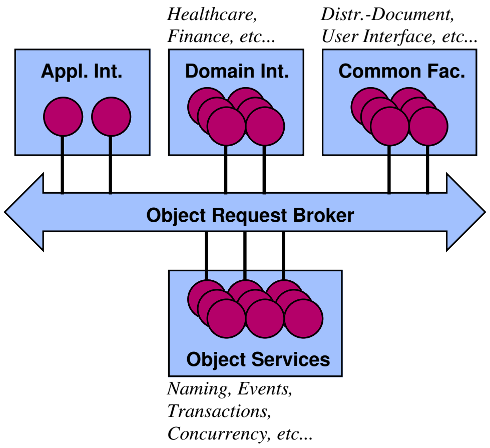


FIG. 1. The OMA Architecture

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

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

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

2.2. A New CORBA Service

Although several CORBA services and facilities have been specified by the OMG, quite nothing has been done concerning reliability and high availability. In fact, a replication service is mentioned as a future CORBA service but no specification has been defined yet. In the following, we motivate the need for designing and implementing a new CORBA service dedicated to group communication, with respect to two alternative approaches: (1) integrating an existing group communication toolkit (e.g., Isis [2]) to CORBA and (2) extending an existing CORBA service.

2.2.1. Reusing a Group Communication Toolkit

As we discuss in [4], there are mainly two ways of integrating a group communication system with CORBA. The first approach consists in building a dedicated Object Request Broker. This approach has been adopted in Orbix+Isis [10] and Electra [11]. 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 and Electra are based on heavyweight proprietary group communication toolkits (respectively Isis and Horus) which do not provide adequate primitives for group-to-group communication, and thus do not support client replication. This implies that replicated CORBA objects are tied to a server role.

An alternative approach consists in intercepting messages issued by an existing ORB and mapping them on a group communication toolkit. This interception approach does not require any modification to the ORB. Eternal [12] uses this mechanism: it intercepts *IIOP* (Internet Inter-ORB Protocol) requests issued by the ORB[1], and maps them onto the Totem group communication toolkit. This approach provides approximately the same degree of transparency as the integration approach. The mapping between object references and replicated servers is managed implicitly by the toolkit, which provides support for replicated clients. Group communication is kept external to the ORB, but depends on low-level mechanisms of the target operating system.

In OGS, we have adopted a *service approach* which consists of designing and implementing (from scratch) group oriented mechanisms on top of an Object Request Broker. This 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 IDL-specified services, such as transactions. As pointed out earlier, an Object Group Service can be considered a fundamental building block for CORBA applications with high availability requirements.

2.2.2. Extending an existing CORBA Service

Upon adopting the service approach, one might wonder whether some of the existing CORBA services can be extended to provide support for high-availability and fault-tolerance. There are mainly two potential candidates: (a) the CORBA Object Transactional Service (OTS), and (b) the CORBA Event Service.

Through the transaction abstraction, OTS provides adequate support to ensure the consistency of multiple replicas despite concurrency and failures. Nevertheless, the current specification of OTS does actually seem to be in contradiction with fault-tolerance. More precisely, OTS is based on the well known two-phase commit protocol, which has two major inconveniences. First, this protocol blocks all objects involved in a transaction if the transaction coordinator crashes. Hence it does not tolerate one site failure. Second, the protocol aborts a transaction if any participant object crashes. This hampers progress if the transaction participants are several replicas of the same object. These issues are discussed in [7].

The Event Service decouples the communication between *suppliers* and *consumers* through *event channels*. Suppliers produce event data and consumers process event data. 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 an intervening object that allows multiple suppliers to communicate with multiple consumers. An event channel is both a consumer and a supplier of events.

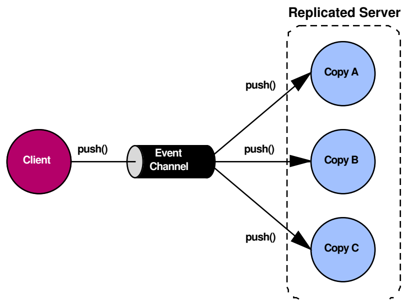


FIG. 2. Using the Event Service for Replication

A natural way of using the Event Service for replication is to use the push model with one event channel: all the copies of a replicated object are consumers of the channel, while clients supply event data on this channel (Figure 2). But, as detailed in [5], the general model introduced by the event service is not adequate as it is for replication and reliable multicast communication, because event channel interfaces define a centralized architecture, quality of service is not sufficient, and the Event Service does not take return values into account.

3. OGS Architecture

3.1. Description

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 can bind to a group using a group name, and issue a single request to all the group members at once. OGS is inherently distributed, and does not depend on any global, critical, or centralized component. The OGS interfaces provide for different levels of reliability. In the following, we first present the external view of OGS and *groupable* objects. Then we describe OGS components and the way they are related.

3.1.1. Typed vs. Untyped Communication

At some point, OGS invokes operations on group member (server) objects. In particular, upon the occurrence of a view change, or when performing a state transfer, OGS has to call back to the group members. Group member objects *inherit* the adequate operations to be invoked in these situations from a predefined IDL interface, enabling OGS to call back to them.

Interface inheritance is adequate when the set of operations that can be invoked on objects by the service is known a priori, as for view change notification or state transfer. But the service has no knowledge of application-specific messages[2] sent by clients to server objects. OGS solves this problem by enforcing all messages to be values of type `Any`, and delivers them to server objects through an operation inherited from the predefined IDL interface. In addition, OGS provides the ability for clients to directly invoke operations of the server interface (typed communication).

Hence, similarly to the CORBA Event Service [14], OGS provides two types of communication: *untyped* and *typed* communication. Untyped communication enables clients to send only values of type `Any`[3] as messages. These messages are received by the servers through their `deliver()` operation. While this message-passing type interface is useful and more efficient in some specific situations, it is generally more convenient for clients to directly invoke an operation of the server interface. Typed communication provides this abstraction; for instance, if the members of an object group support an `Account` interface that defines the `makeDeposit()` operation, a client can directly invoke `makeDeposit()`; OGS intercepts this call and invokes the `makeDeposit()` operation on each member of the group.

Typed communication is an important aspect of 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. 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, and invoke the requested operation on the server interface. Hence, when receiving a typed invocation, the server does not need to be aware that it is a member of a group.

3.1.2. OGS Views

OGS provides several interfaces associated with the different views[4] of the service: (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 OGS to invoke operations on the members of a group. These different views are illustrated in Figure 3.

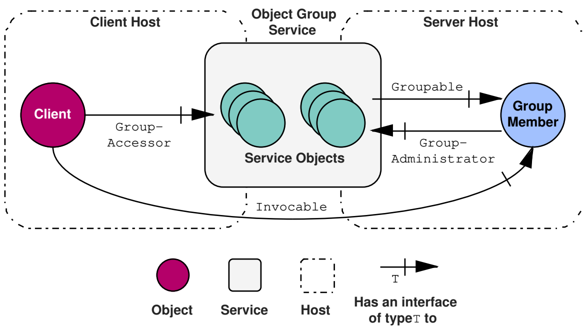


FIG. 3. OGS Views

1. **Client's view** allows a client to get information about groups, to send multicasts to the members of a group, and to send messages to individual members. Clients interact with groups through an interface of type `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()`). When using typed communication, the `GroupAccessorFactory` creates a client-side representative object that implements the server interface using the *Dynamic Skeleton Interface* (DSI).

```
// IDL
module mGroupAccess {
  enum NumReplies { ONEWAY, ZERO, ONE,
                    MAJORITY, ALL };
  enum Ordering { UNRELIABLE, RELIABLE,
                  ATOMIC };

  struct GroupView {
    sequence<Invocable> composition_;
    unsigned long version_;
  };

  exception GroupError { string description; };
  exception NoGroup {};
  exception InvalidGroupName {};

  interface Invocable {
    any deliver(in any msg);
  };

  interface GroupAccessor {
    AnySeq multicast(in any msg,
                     in NumReplies replies,
                     in Ordering order)
      raises(GroupError);
    GroupView get_view()
      raises(GroupError);
  };

  interface GroupAccessorFactory {
    GroupAccessor create(in string g_name)
      raises(GroupError, NoGroup,
             InvalidGroupName);
    Object create_typed(in string g_name,
                        in CORBA::InterfaceDef id)
      raises(GroupError, NoGroup,
             InvalidGroupName);
    void release(in GroupAccessor acc)
      raises(GroupError);
    void release_typed(in Object acc)
      raises(GroupError);
```

```
  };
};
```

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

```
module mGroupAdmin {
  exception NotMember {};
  exception AlreadyMember {};

  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
                               group_name)
      raises(mGroupAccess::GroupError,
             mGroupAccess::InvalidGroupName);
    void release(in GroupAdministrator adm)
      raises(mGroupAccess::GroupError);
  };
};
```

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

```
module mGroupAdmin {
  struct OperationSemantics {
    CORBA::Identifier name_;
    mGroupAccess::Ordering ordering_;
  };
  typedef sequence<OperationSemantics>
          OperationSemanticsSeq;

  interface Groupable : mGroupAccess::Invocable {
    void view_change(in mGroupAccess::GroupView
                     view);
    any get_state();
    void set_state(in any state);
    OperationSemanticsSeq operation_semantics();
  };
};
```

3.1.3. Interacting with OGS

Figure 4 gives an abstract view of the interactions between clients, servers, and OGS. The `GroupAccessor` object (Acc) acts as a client-side representative for the group, and is located on the same host (or in the same process) as the client. The `GroupAdministrator` objects (Adm) interact directly with group members, and are located on server hosts. Performing a multicast to the group initiates some protocol between `GroupAccessor` and `GroupAdministrator` objects (Π), which ensures that the messages are delivered to the members according to some condition (e.g., total order).

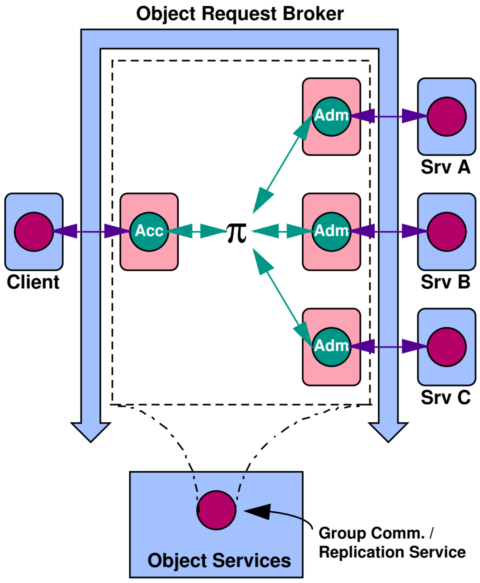


FIG. 4. OGS Architecture

Figure 5 illustrates the interactions between clients, servers (i.e., group members) and OGS. 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), or directly an operation of the server's interface when using typed communication. This initiates the multicast protocol (4'), and the message is eventually delivered to the servers (5).

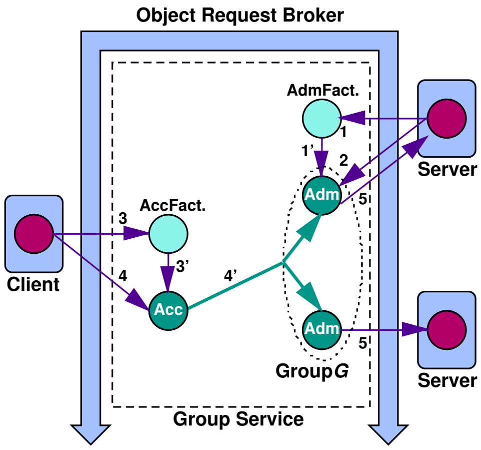
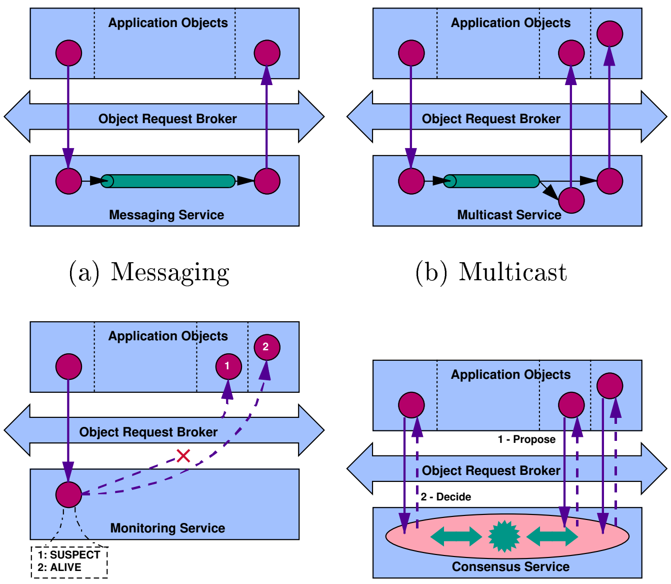


FIG. 5. OGS Interactions

3.2. OGS Components

OGS 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 communication.
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 implement atomicity and total order of multicast invocations.



(a) Messaging (b) Multicast

(c) Monitoring (d) Consensus

FIG. 6. Overview of the Services

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), in charge of group management and group communication.

3.2.1. 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 [5]. 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).

3.2.2. The Object Multicast Service

The *Object Multicast Service* provides a way to send a message to several CORBA objects at once. 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. These multicast primitives do not however provide ordering guarantees, such as total-order, which are implemented at a higher level using the Object Consensus Service. The Object Multicast Service performs remote communications through the Object Messaging Service.

3.2.3. 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 Section 3.2.4). Failure detection does not need to be accurate, i.e., our protocols are correct despite unreliable failure detections [3].

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.

3.2.4. 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 [8]. While most consensus-based environments use ad hoc protocols, it is natural in CORBA's object-oriented context to develop a generic consensus service that can be reused to solve various problems.

The Object Consensus Service is used by 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. Group Communication in OGS

4.1. Invocation Types

OGS provides several types of multicast invocations:

- *Unreliable multicast invocations* enable to invoke several objects at once, but without the guarantee that all servers receive the invocation *if the client crashes while sending the requests.*
- *Reliable multicast invocations* enable to invoke several objects at once with the guarantee that either all correct servers receive the invocation, or none of them does.
- *Total order multicast invocations* enable to invoke several objects at once with the guarantee that all correct servers receive the same set of invocations in the same total order.

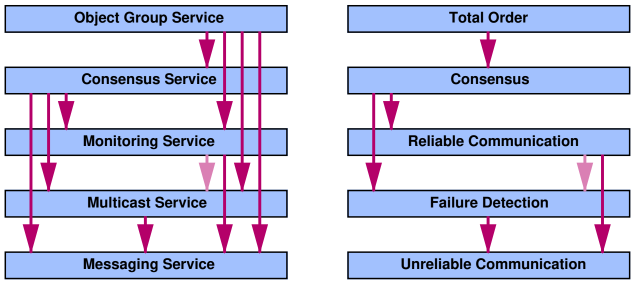
These invocations types can be combined with several types of synchrony:

- *Oneway multicast invocations* do not wait for replies from the server.
- *One-reply multicast invocations* wait for the first reply from the servers.
- *Majority-of-replies multicast invocations* wait for a majority of replies from the servers.
- *All-replies multicast invocations*[6] wait for replies from all servers.

In our model, we consider reliable communication channels. Communications go through TCP-based IIOP, which provides reliable message delivery[7]. All multicast invocations are initiated by an unreliable multicast sent by the client to the servers. Depending on the real invocation semantics, a protocol is then initiated by the servers. Therefore, the servers can decide upon the protocol to use for a specific message. For instance, a simple protocol for reliable multicasts is to let each server forward each message to each other server the first time it receives it; this pessimistic protocol ensures that all correct servers eventually receive the message even if the client fails after having sent it to a subset of the servers.

We use the failure detection mechanism for implementing an optimistic reliable multicast protocol. Since communication channels are reliable, the only scenario in which a multicast invocation might not reach all servers is when the sender fails while issuing the messages. Therefore, for ensuring reliable delivery, the servers will forward a message if and only if they suspect the sender.

Furthermore, the failure detection mechanism is used by the consensus protocol, which in turn is used for ensuring total order of messages and view changes. The basic idea is to buffer total order messages and to launch a consensus protocol for agreeing on the set of messages to deliver, and on their respective ordering. Once the consensus has decided, all servers will deliver the messages. A consensus might order several messages at once, thus increasing the throughput of the system. Figure 7 shows the dependencies between the different components and protocols of the OGS.



(a) Protocols (b) Services

FIG. 7. Service and Protocol Dependencies

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

OGS requires no extension to the current CORBA specifications. Furthermore, all 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, or any ORB that provides non-blocking oneway 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.

4.2. Invocation Semantics

Whereas traditional group communication toolkits let the client choose the semantics of multicasts (e.g., reliable, total order, etc.), our environment provides server-defined invocation semantics[8]. This approach enforces encapsulation as the client does not need to know the ordering guarantees that actually depend on the operation implementations. In all cases the server indeed knows the maximal ordering guarantees needed for a specific operation, and in most cases it knows the minimal guarantees required by the client. For instance, if an operation does not change the state of the server (read-only operation), there is no need for a total order multicast; on the other hand, if an operation changes the state of the server (update operation), total order is required. Therefore, the client does not need to be aware of the exact semantics of the operation executed on the server. This is a big improvement over the traditional model where a client asks the strongest ordering guarantees for a message when it is not aware of the exact semantics of the associated operation.

Furthermore, since the server knows the semantics associated with an operation, it can optimize client requests in consequence. For instance, two update operations need not to be totally ordered if they modify disjoint parts of the server's state[9].

Sometimes, nevertheless, the client may require weaker guarantees than those associated with an operation. In this case, the client may give a *hint* to the server in which it specifies the ordering guarantees it requires. The server is still allowed to optimize such a request (e.g., if no concurrent request is executing, the same semantics can be guaranteed using a simpler protocol), but it should not provide stronger ordering guarantees than those asked by the client unless it might lead to an inconsistent state. If the server does not know the semantics associated with an operation, it will use the strongest ordering guarantees available for the invocation (i.e., total order).

4.3. *Typed Communication*

Typed communication is an important aspect of 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). The service transparently filters messages and returns a single reply to the client. Of course, typed communication requires that all servers implement the same IDL interface.

Typed communication is achieved by OGS using two advanced features of the CORBA specification: the *Dynamic Skeleton Interface* (DSI) and the *Dynamic Invocation Interface* (DII). 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.

4.3.1. General Principle

The approach used in OGS for implementing typed multicast communication is similar to the CORBA *request level bridging* [13]. 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).

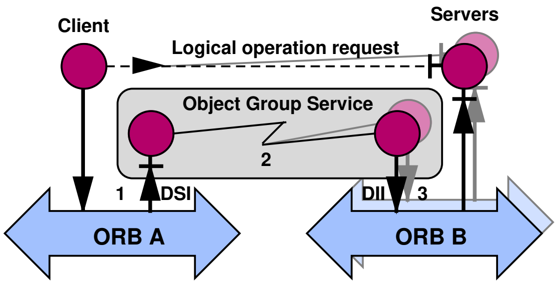


FIG. 8. Service Implementation

The general principle for implementing typed communication, illustrated in Figure 8, is as follows: (1) The original request is passed to an OGS object (`GroupAccessor`) 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) OGS server-side 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) are implemented.

4.3.2. Accepting Requests Using the DSI: the Client Side

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, 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 to receive operation invocations on an IDL interface 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, OGS gives the illusion to the client that it really implements the IDL interface of the server. OGS intercepts invocations, transforms them into an agreed format, and multicasts them to all the group member objects.

On the client side, OGS uses the CORBA Interface Repository (IR) for getting information about the IDL interface of the replicated object. The IR is accessed only once, when creating the typed group accessor, to keep communication overhead as small as possible.

When performing a typed multicast invocation, the service waits for a single reply from the servers, unless the operation is explicitly declared as oneway in the server's IDL interface, in which case a oneway multicast invocation is used.

4.3.3. 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 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. OGS objects on server hosts receive the details of the request to be made as part of the multicast message sent by the client. The message contains information on the object that must be invoked, the operation name, the parameters, etc. OGS translates this into a DII call leading to the invocation of the requested operation on the server interface.

5. Configuration

5.1. *Group Naming*

Each group is associated with a logical name, which acts as a system-wide unique identifier. Ideally, and to prevent naming conflicts, a name will have the form of a Uniform Resource Locator (URL), like `ogs://banks.epfl.ch/swiss`. But it could also be any sequence of characters.

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 bind to a group of which it is not a member, e.g., to multicast a message or to join the group, it has to find a reference to the members of the group to contact them. Therefore, a group naming service is required for maintaining the mapping between group names and references to member objects. This group naming service should (1) be fault-tolerant, (2) provide ways for consistent updates of group information, and (3) be kept up-to-date.

For this group naming service, we rely on implementations of the CORBA *Naming Service* specified in [14] for several reasons: the specification of this service is general enough to meet our needs; with this approach, we follow the CORBA design guidelines which aim at promoting mutual reuse among services, and not to duplicate functionalities; finally, some implementation of the CORBA Naming Service already take fault-tolerance into account.

Furthermore, most current implementations of the Naming Service are robust enough, and are able to recover from failure. In the worst case, a failure of the Naming Service may hinder a new client or member to bind to the group, but it will not disturb objects that are already bound to the group.

Updating the Naming Service is a difficult task. OGS updates the information in the Naming Service each time a view change occurs, but this information is not guaranteed to be up-to-date. In practice, when an object wants to bind to a group, OGS contacts the Naming Service, retrieves the list of the group members, and contacts these members; if at least one current member of the group is reached, the object will manage to bind to the group. It is very unlikely that all the objects listed in the Naming Service have left the group in the meantime[10]; if this was the case nevertheless, OGS would consult the Naming Service again until it succeeds in contacting some current member of the group.

5.2. *OGS Localization*

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

It makes sense to locate the service on the same host as the server; that way, the server will be considered as failed only if the machine crashes. If they are on different hosts, either the crash of any of the machines or a network partition between them will be considered as a failure of the server.

The current implementation of OGS can run as a separate process on the client and the server's host (the `OGSd` daemon program), or be linked with application code (the `OGSl` library). The first approach has the advantage of decoupling the service from the application, enabling several applications running on the same host to use the same resources. It also allows user applications written in Java to use the C++ service. The second approach is more efficient since inter-process communications are more costly than invocations between objects located in the same process.

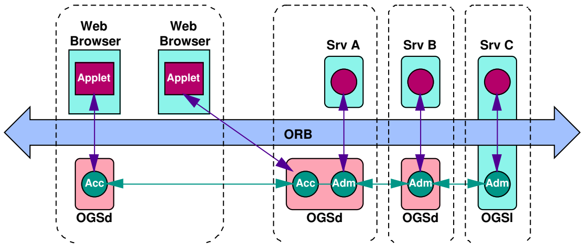


FIG. 9. Service Localization

Figure 9 illustrates a typical configuration of the service components in a heterogeneous environment. Two Java clients interact through a web browser with a group of replicated objects: the first client uses a local OGS, while the second one uses a remote OGS. Three server applications are part of a group: the first two also use a local OGS in a separate process, while the third is directly linked with OGS library; this basically means that the first two servers can be implemented in any programming language supporting CORBA, while the third server has to be written in C++.

6. A Complete Example

This section presents an example use of OGS. The sample application is composed of a replicated counter object that may be incremented and reset (Figure 10). We first present the IDL interfaces of the application objects, then we describe the client and server implementations.

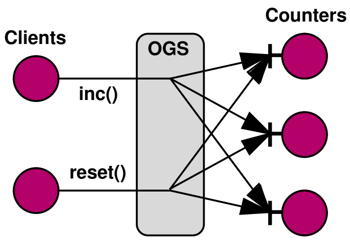


FIG. 10. Counter

6.1. IDL Interfaces

```
// IDL
#include "GroupAdmin.idl"

interface Counter : mGroupAdmin::Groupable {
  readonly attribute long count;

  void inc();
  void reset();
};
```

To get the ability to be a member of a group, the `Counter` interface has to inherit from OGS' `Groupable` interface. This enables the service to call back to the application object for view change notifications, state transfer, and message delivery.

This example illustrates consistency problems that may arise in replication, since the `inc()` and `reset()` operations are not commutative, and if the copies of the replicated counter do not receive the operation invocations in the same global order, the value of the counter may become inconsistent. Therefore, these operation invocations need to be totally ordered with respect to each other.

On the other hand, accessing the current value of the counter does not need to be performed using a total order mechanism: it does not change the state of the counters, and it does not matter if a client receives different values from different servers since it will typically use only the first reply.

6.2. Server Implementation

In addition to its own operations (`inc()`, `reset()`, and `count()`), the `Counter` implements the operations it inherits from the `Groupable` interface. It must also join a group at creation time. The `deliver()` operation is empty since it is only used for untyped message, and our example is based on typed communication. The `get_state()` and `set_state()` are responsible for respectively packing the state into an `Any`, and extracting it from an `Any`. The `operation_semantics()` operation returns a structure detailing the semantics associated to each operation of the `Counter` interface. This information is used by the service to provide the most efficient protocols based on real operation semantics. Part of the C++ code implementing the `Counter` object is given below.

6.2.1. Groupable Operations

```
// C++ (error handling code is omitted)

CORBA::Any* Counter_i::deliver
  (const CORBA::Any& message) {
  return new CORBA::Any();
}

void Counter_i::view_change
  (const mGroupAccess::GroupView& newView) {}

CORBA::Any* Counter_i::get_state() {
  // Pack the state into an any
  CORBA::Any* a = new CORBA::Any();
  *a <<= count_;
  return a;
}

void Counter_i::set_state(const CORBA::Any& a) {
  // Extract the state from an any
  a >>= count_;
}

mGroupAdmin::OperationSemanticsSeq*
    Counter_i::operation_semantics() {
  // Return the semantics associated to each operation
  mGroupAdmin::OperationSemanticsSeq* oss =
    new OperationSemanticsSeq(3);
  oss->length(3);
```

```
  (*oss)[0].name_ = "_get_count";
  (*oss)[0].ordering_ = mGroupAccess::UNRELIABLE;
  (*oss)[1].name_ = "inc";
  (*oss)[1].ordering_ = mGroupAccess::ATOMIC;
  (*oss)[2].name_ = "reset";
  (*oss)[2].ordering_ = mGroupAccess::ATOMIC;
  return oss;
}
```

6.2.2. Counter Operations

```
// C++ (error handling code is omitted)

CORBA::Long Counter_i::count() { return count_; }

void Counter_i::inc() { count_++; }

void Counter_i::reset() { count_ = 0; }
```

6.2.3. Main

```
// C++ (error handling code is omitted)

int main(int argc, char *argv[]) {
  // Obtain initial references to the ORB
  CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
  CORBA::BOA_var boa = orb->BOA_init(argc, argv);

  // Create server application objects
  Counter_var cnt = new Counter_i();

  // Register object implementation
  boa()->obj_is_ready(cnt);

  // Bind to group administrator factory
  mGroupAdmin::GroupAdministratorFactory_var gaf = ...;

  // Create a group administrator
  mGroupAdmin::GroupAdministrator_var ga =
    gaf->create("ogs://counters.epfl.ch/clock");

  // Join the group
  ga->join_group(cnt);

  // Wait for messages
  boa->impl_is_ready();
  return 0;
}
```

6.3. *Client Implementation*

The client's implementation basically consists in creating a typed proxy for the group, narrowing it to the type of the server, and invoking operation on the server's interface that will be actually multicast transparently by the service.

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 advanced features (e.g., to receive more than one reply resulting from a multicast invocation), the client has to invoke explicitly OGS and hence loses the benefits of transparency.

```
// C++ (error handling code is omitted)

int main(int argc, char *argv[]) {
  // Obtain initial references to the ORB
  CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
  CORBA::BOA_var boa = orb->BOA_init(argc, argv);

  // Bind to group accessor factory
  mGroupAccess::GroupAccessorFactory_var gaf = ...;

  // Get a reference to the interface repository
  CORBA::Repository_var ir = CORBA::Repository::_narrow
    (orb->resolve_initial_references
     ("InterfaceRepository"));

  // Get a reference to the interface definition of the server
  CORBA::InterfaceDef_var intf =
    CORBA::InterfaceDef::_narrow(ir->lookup("Counter"));

  // Create a typed group accessor
  CORBA::Counter_var cnt = CORBA::Counter::_narrow
    (gaf->create_typed("ogs://counters.epfl.ch/clock",
     intf));

  // Invoke replicated server
  cnt->reset();
  cout << "Counter is " << cnt->count() << endl;
  cnt->inc();
  cout << "Counter is " << cnt->count() << endl;

  return 0;
}
```

7. OGS Performance

This section presents the performance of two different OGS implementations: on Orbix [9] and VisiBroker [15]. We focus on three different types of measurements:

- Multicast invocations, i.e., the total cost of the different multicast communication primitives.
- One-to-one remote invocations, i.e., the cost of remote invocations through the ORB.
- Request management, i.e., the cost of constructing requests, inserting and extracting complex datatypes into `Any` variables, etc.

Our performance measurements have been performed with Orbix 2.2MT and VisiBroker for C++ 3.0, on three Sun UltraSPARC 1 workstations, with 64 MB of memory and a 8 KB socket queue size, on a local 10 Megabits per-second (Mbps) Ethernet network. The client application is located on the same machine as one of the servers. The tests have been performed with Orbix (using a single thread), and VisiBroker (using multiple threads).

7.1. *Multicast Invocations*

Table 1 presents the number of invocations per second performed on three replicas, using (1) different communication primitives: unreliable multicast, reliable multicast and total order multicast, and (2) two models of synchrony: the client waits for the first reply, and the client waits for replies from all members[11].

TABLE 1. Multicast Invocations (invocations/sec)

ORB	Orbix (ST)	VisiBroker (MT)
Unreliable - 1 reply	70.2194	100.185
Unreliable - all replies	62.5503	20.1594
Reliable - 1 reply	60.7294	62.4234
Reliable - all replies	37.0247	20.2185
Total order - 1 reply	16.3865	25.7907
Total order - all replies	12.1513	19.6833

Table 2 presents the overhead of using a separate process on the client side for invoking replicated objects — rather that collocating OGS with the client —, and of using the typed version of OGS. These tests have been performed with VisiBroker (using multiple threads), waiting only for the first reply.

TABLE 2. Overhead of Two Processes and Typed OGS (invocations/sec)

Model	1 process (untyped)	2 processes (untyped)	2 processes (typed)
Unreliable	100.185	81.4379	36.47
Reliable	62.4234	55.3852	32.5141
Total order	25.7907	22.6333	17.9076

7.2. One-to-one Remote Invocations

The cost of raw remote invocations through the ORB (using IIOP) is presented in Table 3. The test consists in invoking an operation that takes a single `inout` parameter, which is a value of a simple or complex type. The simple data type is a `long` variable, and the complex data type is a structure composed of a `long` variable, and a sequence of object references containing three elements. Results are expressed in number of operations per second, and have been obtained by taking the mean of 10000 executions. Note that a much more comprehensive analysis of CORBA communication overhead can be found in [6].

TABLE 3. One-to-one Remote Invocations (invocations/sec)

ORB	Orbix	VisiBroker
Simple type	354.48	771.462
Complex type	118.229	160.923

7.3. Request Management

When profiling OGS, we noticed that a non-negligible part of the time required for remote invocations was spent in constructing requests, and when working with untyped `Any` values. Unlike other IDL types, `Any` values are augmented by a `typecode` information that contains details about the actual type of the value. This information increases the size of the messages sent on the network. Moreover, validity checks upon data extraction slow down the remote invocation process.

Table 4 presents the cost associated to `Any` management with Orbix and VisiBroker, for (1) inserting simple and complex types into `Any` variables, (2) copying `Any` variables, and (3) extracting simple and complex types from `Any` variables[12]. The simple and complex data types are the same as in previous test. Results are expressed in thousands of operations per second, and have been obtained by taking the mean of 10000 executions.

TABLE 4. Cost of `Any` Management (thousands of operations/sec)

ORB	Orbix	VisiBroker
Insertion (simple type)	62.5849	213.831
Insertion (complex type)	2.71964	6.83963
Copy (simple type)	57.0334	231.107
Copy (complex type)	1.85471	22.8686
Extraction (simple type)	1213.59	4526.94
Extraction (complex type)	0.4197	0.221259

7.4. Evaluation

It turns out from these performance results that OGS with VisiBroker is more efficient than OGS with Orbix in nearly all domains, except when performing multicast invocations and waiting for all responses. This may be due to a high load of one of the servers during the tests.

An interesting result is that the cost of request management is not negligible. Extracting a complex structure from an `Any` value requires a time comparable to that of performing a remote invocation.

Not surprisingly, the typed version of OGS is less efficient than the untyped one, but the difference gets smaller with complex protocols such as total order. This is due to the fact that the DSI and the DII are used only once per request, and add a fixed cost to the invocation time. If the protocol is complex, more messages are generated, without increasing the fixed cost of dynamic request processing.

8. Concluding Remarks

The *Object Group Service (OGS)* provides facilities for CORBA object group communication and hence supports reliability and high availability through replication: a set of replicas constitutes a group, viewed by clients as a single entity. Through the group abstraction, the failure of a replica is made transparent to the client and any read access to a replicated object can transparently be performed through the closest replica.

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.

OGS is now fully specified. It is implemented in C++, but application objects can be programmed in other languages such as Java. OGS currently works with both Orbix [9] and VisiBroker [15] and it can be ported to any other standard CORBA 2.0 ORB. More information on OGS, including performance figures and a binary version of the service itself are available on the web at `http://lsewww.epfl.ch/OGS/`

Notes

1. Requests are intercepted by listening to a specific Unix device.
2. In this context, a *message* is the data associated to a request.
3. A CORBA `Any` variable can contain a value of any type.
4. In this context, a *view* refers to the set of interfaces that are exported by the service, and should not be confused with a *group view* (i.e., composition of a group) provided by the group communication system.
5. Note that a Messaging Service is currently being defined by the OMG. We plan to use it as soon as implementations become available.
6. With majority-of-replies and all-replies multicast invocations, if a server crashes while the invocation is processed, OGS might need to adapt the expected number of replies.
7. The sender receives a notification if the transport subsystem was not able to deliver the message.
8. Note that server-defined invocation semantics are available only for the typed version of OGS.
9. Note that the `OperationSemantics` structure should be augmented to provide information about commutative operations.
10. A special case is when the only member of a group crashes. In this case, the crashed object is not able to remove itself from the Naming Service.
11. Note that the numbers represent the latency of an invocation, and not the throughput of OGS.
12. Note that, upon extraction from a complex type, Orbix returns a pointer to an internal buffer while VisiBroker copies the data.

References

[1] K. Birman and R. van Renessee. *Reliable Distributed Computing with the Isis Toolkit.* IEEE Computer Society Press, 1994.

[2] K.P. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):36–53, December 1993.

[3] 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.

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

[5] P. Felber, R. Guerraoui, and A. Schiper. Replicating objects using the CORBA event service. In *The 6th IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS'97)*, pages 14–19, October 1997.

[6] A. Gokhale and D. C. Schmidt. Measuring the performance of communication middleware on high-speed networks. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM), Stanford*, August 1996.

[7] R. Guerraoui, R. Oliveira, and A. Schiper. Atomic updates of replicated objects. In *European Dependable Computing Conference (EDCC'96)*, number 1150 in Lecture Notes in Computer Science, pages 365–382, Taormina, October 1996. Springer Verlag.

[8] 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.

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

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

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

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

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

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

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