Programming with Object Groups in CORBA

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

Abstract

The Object Group Service (OGS) extends CORBA with the ability to gather several objects inside a group and to transparently handle the group membership and the consistent invocations of the group members. OGS does not require any change to the CORBA specification: it is portable on any CORBA 2.0 ORB and does not rely on ORB vendor-specific features. We present the OGS programming model and we discuss various levels of group transparency and their impact on invocation overheads.

1 Introduction

Most object-based distributed architectures heavily rely on *remote method invocation* as a basic abstraction for inter-object communication. This abstraction simplifies distributed programming by making communication with a remote object look like communication with a local object. Its limitation, however, is that it can only be employed for two-ways communication between a client object and a server object, which is not convenient when the application is composed of distributed objects with a high degree of inter-dependence. In this case, the communication should reflect the inter-dependence and take place from one object to a group of objects implementing a given service. The client should be able to send requests to the group as a whole, rather than being required to know the group membership and to communicate with members on a one-to-one basis. This is even more crucial if the server can change its membership or location during its lifetime.

To provide adequate support for object group semantics in the context of the *Common Object Request Broker Architecture* (CORBA) [2], we have designed and implemented an *Object Group Service* (OGS), which extends CORBA with the ability to manage object group membership and communication [2]. Unlike other tentatives for extending CORBA with object group capabilities, our approach does not rely on any specific Object Request Broker implementation nor on any operating system facility (see sidebar *"Related Work"*).

OGS is CORBA compliant and can be made completely non-intrusive for clients of the service. Rather than committing to one level of *group transparency*, OGS provides flexible programming and configuration models allowing the developer to trade between transparency with availability and efficiency (see sidebar *"The Overhead of OGS"*) according to the nature of the application.

2 A CORBA service

The *Object Management Architecture* (OMA) [2], specified by the *Object Management Group* (OMG), is a conceptual infrastructure for building interoperable software components, based on open standard object-oriented interfaces.

Commercially known as CORBA, the *Object Request Broker* (ORB) is the communication heart of the standard. It is a *software bus* that enables heterogeneous objects to transparently invoke remote operations and receive replies in a distributed environment. Each object interface is specified in the declarative OMG *Interface Definition Language* (IDL), which is implementation independent. Clients use *object references* to identify remote objects and invoke operations on them. The *Common Object Services* (COS) are a collection of interfaces and objects supporting basic functionalities useful for most CORBA applications. 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, but nothing has been specified yet concerning object group management and group communication.

Our *Object Group Service* (OGS) manages groups of CORBA objects and provides primitives to communicate atomically 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 group members at once. OGS is inherently distributed, and does not depend on any global, critical, or centralized component [3].

3 OGS Interfaces

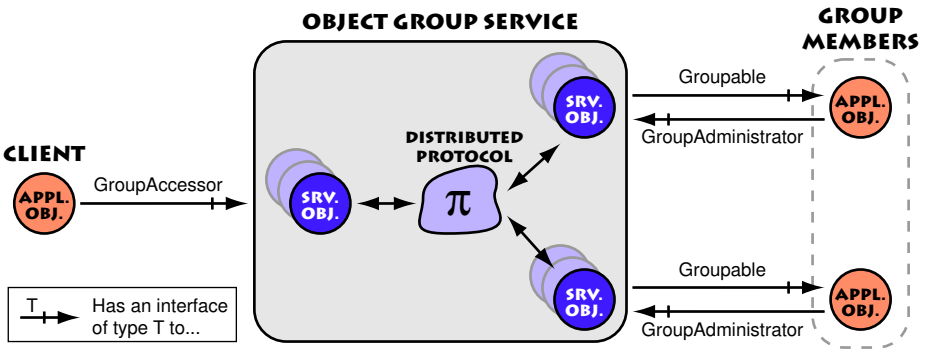

Figure 1: OGS Components and Interfaces

Figure 1 presents a simplified view of OGS components and interfaces. OGS interfaces are associated with the different views of the service: (1) the *client's view*, used to invoke the group, (2) the *member's view*, used by a group member to change its status within the group (e.g., *join* or *leave* the group) and to

communicate with other objects in the group, and (3) the *service's view*, used by OGS to invoke operations on the members of a group.

1. *Client's interfaces* allow a client to get information about groups and send multicasts to the members of a group. Clients interact with groups through an interface of type `GroupAccessor` which acts as a local representative for the group. The `GroupAccessor` interface defines an operation for multicasting messages to the group that it represents (`multicast()`).

2. *Member's interfaces* are a superset of client's interfaces. Objects can additionally join and leave groups using the `join_group()` and `leave_group()` operations of the `GroupAdministrator` interface.

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

Group accessors and administrators are service objects: their interfaces are called *OGS internal interfaces*.[1] Performing a multicast to the group initiates a protocol between group accessors and administrators, which ensures that messages are delivered to the members according to some condition (e.g., total order). The `Groupable` interface must be implemented by application objects: it is an *OGS external interface*.

4 The Developer's View

Client Perspective

OGS provides two types of communication: *untyped* and *typed* communication. Untyped communication enables clients to send only values of type `any`[2] as messages. These messages are received by the group members 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()` with the relevant parameters; OGS intercepts this call using CORBA's *Dynamic Skeleton Interface* (DSI), multicasts it, and invokes the `makeDeposit()` operation on each member of the group using CORBA's *Dynamic Invocation Interface* (DII). In other words, OGS provides support for *non-intrusive* typed communication: OGS allows a client application to invoke a group of replicas just as if it were invoking a non-replicated object.

The object that acts as a group proxy and against which actual client invocations are performed is called a *group accessor*. A group accessor is an object

[1] It is important to notice that the term *internal* does not mean *private*. An *internal* interface is accessible to the application developer, but its implementation is provided internally by the service.

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

that encapsulates the structure and behavior of a group reference, and that remembers and tracks the composition of the group. Ideally, it should be a temporary object, created on-the-fly when a group reference enters the address space of the application. This is however not possible without some ORB support, which is currently available only through vendor-specific extensions (such as Orbix smart proxies). With OGS, a group accessor can be instanciated either explicetely by the application (making the use of OGS non-transaprent), or at deployment time using the `ogsutil` application provided with OGS. This application creates the necessary service objects and registers them in the naming service under names agreed upon with the clients (see Figure 2). The `ogsutil` application first creates a group accessor using an object factory (1, 2), and registers this accessor in the naming service (3). Upon startup, the clients get the reference to the group accessor from the naming service (4), and invoke the object group (5, 6). When using typed communication the group accessor will appear as having the same interface as the server group, making the use of OGS completely transparent to the client.

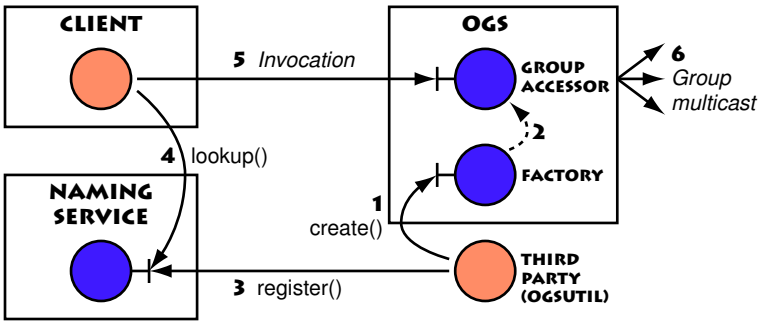

Figure 2: Implicit Service Instantiation

For the purpose of replication, object groups may be handled in a completely transparent way from the client side (see the replicated calculator example in the sidebar *"Putting OGS to Work"*). The client simply looks for a reference to the group in the naming service, and invokes this group, as if it were a singleton object. This enables to increase the reliability of a non fault-tolerant application (through replication) without touching the original code of the clients. One can start by developing a CORBA application without OGS; once the interfaces have been defined, and the application has been implemented and tested without OGS, group support may be added at a later stage with only minimal additions to the servers.

If groups are *not* used for replication, special care has to be taken when defining the IDL interfaces of the application. In particular, if the client wants to access all the replies resulting from an invocation to an object group (e.g., when using groups for parallel processing), the typed version of OGS cannot be used directly. In this situation, though, it is possible to preserve transparency by decoupling the application interfaces so that requests are issued through oneway multicast invocations,[3] and replies are returned explicitly to the caller through

[3]Note that OGS guarantees atomicity of oneway requests: they are either received by all members of a group, or by none.

point-to-point invocation, as shown in the Mandelbrot example in the sidebar *"Putting OGS to Work"*.

Server Perspective

To benefit from group communication, the programmer has to provide application support for OGS. This is performed by having server objects inherit from the `Groupable` IDL interface. This interface defines the following operations that the server objects have to implement to be member of a group.

- *Support for message delivery:* messages sent using the untyped version of OGS are delivered to the server objects through their `deliver()` operation. If the clients use only the typed version of OGS, this operation may be left empty since messages will be delivered through invocations on the server's application-specific interface.
- *Support for view change notification:* when a new object joins a group, or a member object leaves or fails, all member objects are notified through their `view_change()` operation. They receive an ordered list of current group members, which may be used for instance to deterministically assign a different role to each object of the group (see the Mandelbrot example in the sidebar *"Putting OGS to Work"*).
- *Support for state transfer:* when a new object joins a group, it atomically receives the shared state from the other members of the group. This is generally required to preserve the application consistency. The state transfer mechanism is implemented by two operations, `get_state()` and `set_state()`, which are atomically invoked on a current and on the new member respectively.
- *Support for operation semantics:* the server objects can specify the semantics associated to each operation of their interface. This is performed by implementing `operation_semantics()`. A program can for instance achieve better performance by not mandating total ordering of invocations to read-only operations. By default, all operations are associated with the strongest semantics, i.e., totally ordered multicast.

Typical implementations of these operations are described in the sidebar *"Putting OGS to Work"*.

5 The Administrator's View

Invocations to object groups are performed by OGS. This happens by having clients messages go through group accessor objects, and server messages go through group administrator objects. Group accessors and administrators form the visible part of the OGS runtime system. The application administrator can configure this runtime system in a number of ways, leading to different degrees of flexibility, efficiency, transparency, or reliability.

Execution Styles

OGS provides mainly two execution styles: a *linkable* style and a *daemon* style. In the first style, the service objects are co-located with application objects, i.e., they are linked with the application and they execute in the same address space (or process). In the second style, the service objects are located in another process — the `OGSd` daemon program — which may be on the local or on a remote host.

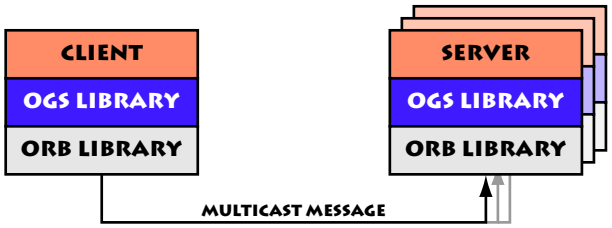

Figure 3: The Linkable Style

The linkable version of OGS is provided as a C++ dynamic library (`OGSl`) to be linked with C++ applications, or as a set of Java classes usable from Java applications (see Figure 3). This execution style is more efficient since invocations between objects located in the same process are less costly than inter-process communications (see the sidebar *"The Overhead of OGS"*). However, it enforces the code of the application to be written with the same programming language as the library.

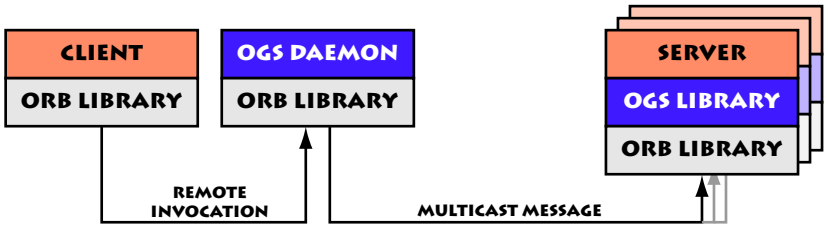

Figure 4: The Daemon Style

The *daemon* execution style, with two separate processes, has the advantage of decoupling the service from the application, enabling several applications running on the same host to share the same OGS instance. It also allows user applications written in another programming language, such as Smalltalk, to use the C++ or Java service. The deamon and linkable execution styles are not exclusive of each other, and different styles can be employed on the client and server sides. Figure 4 illustrates the use of the OGS daemon on the client side, while the service is linked with the application on the server side.

OGS Deployment

Whereas CORBA objects should be independent of their real location, some deployment choices will affect the degree of fault tolerance of an application that uses OGS. This is more a semantic requirement than an architectural requirement since OGS objects, like any CORBA object, can be deployed anywhere.

For instance, when using the daemon execution style, if both the application and the daemon are located on the same machine, only a crash of the OGS daemon process can prevent the application from using the service. If they are on different machines, there is the additional risk of a link or machine failure. When using the linkable style, the application is not exposed to these problems and OGS objects can be considered as always available.

6 Concluding Remarks

When designing and implementing OGS, we draw several observations from our experiences in building support for object groups.

On the conceptual side, one of the major issue comes from group addressing. Clients do not address objects anymore; they address groups of objects. The CORBA object model states that an object reference designates univoquely a single object. Extending this definition to a set of objects requires modifying the CORBA model. This is one of the reason why OGS uses proxy objects to represent actual groups.

In addition, there is a mismatch between addressing and identity. CORBA defines a *strong addressing* model (object location is embedded in CORBA references) and a *weak identity* model (the result of a comparison for equality between two object references is "yes" or "maybe"). When working with references to object groups, we typically need weak addressing (the reference must designate the group as an abstract entity independent of the number or location of its members) and strong identity (for instance to test if an object is member of a group).

Another mismatch comes from the synchronous nature of CORBA incovations versus the asynchrony requirements of multicast protocols. Although CORBA defines deferred synchronous and oneway invocations, there is no guarantee that such an invocation will not block. In contrast, multicast protocols have strong requirements in terms of asynchrony. In the implementation of OGS, we avoid this problem by using multithreading to ensure asynchrony.

On the implementation side, we experienced that the use of typed invocations had a non-negligible cost. Although very powerful, the DSI and DII are expensive in terms of processing time, leading to a tradeoff between transparency and performance (see side bar *"The Overhead of OGS"*). One of the main lessons we have drawn from our experimentations with OGS is the importance of distinguishing different levels of transparency and providing the developer with the ability to access and tune even the very low-level components of OGS.

OGS Status

The design and implementation effort of OGS was initiated in 1994 in the context of the European ESPRIT project OpenDREAMS (project 20843) and and

was continued in the context of the European ESPRIT project OpenDREAMS-II (project 25262). Binary code versions of OGS (both for Orbix 2.3 MT and Visibroker 3.2) are available at `http://lsewww.epfl.ch/OGS/`.

References

[1] OMG. *The Common Object-Request Broker Architecture: Architecture and Specification – CORBAservices: Common Object Services Specification.* http://www.omg.org.

[2] 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.

[3] P. Felber, R. Guerraoui, and A. Schiper. *The Implementation of a CORBA Object Group Service.* Theory and Practice of Object Systems, 4(2), 1998.

Related Work: From Process Groups to CORBA Object Groups

The first system to offer explicit notions of *group* and *multicast communication* is the V system [3]. Its design influenced most existing group-based systems. The Isis system extended the group model of the V system by providing support for fault-tolerance through mechanisms such as process group membership and reliable totally ordered multicast [2]. Inspired by Isis, several group communication systems such as Transis, Totem, and Horus [9] have more recently been developed. These systems mainly differ in the way messages are ordered (e.g., total order/causal order) and in the way atomicity of message delivery is ensured in the case of process or link failures.

Until very recently, CORBA and group-based systems were considered very distinct technologies. Communication in CORBA used to basically rely on inter-object *one-to-one* method invocation [2], whereas group-based systems focused on process groups (vs. object groups) [5]. Extending CORBA with a group abstraction is a new and challenging task. We have distinguished mainly three different approaches to accomplish this task, discussed in details in [1]:

- The *integration approach* consists in embedding group capabilities within an Object Request Broker (ORB). This approach has been adopted in the Electra system [6]. Although appealing for its inherent transparency (an object group is not distinguishable by a client from a singleton object that implements the same interface), this approach does not comply with the CORBA object model and results in proprietary systems.

- The *interception approach* consists in transforming standard IIOP messages issued by an ORB into group communications. Unlike in the *integration* approach, the ORB is not aware of the existence of group mechanisms. Eternal [1] uses this mechanism: the *Eternal Interceptor* captures IIOP messages and the *Eternal Replication Manager* maps them onto the Totem multicast group communication system [9]. Eternal intercepts system calls before they reach the kernel, by performing a low level continual trace of inter-process communications. These system calls are then modified appropriately and passed to the process group interface for communication over Totem.

- The *service approach* provides group communication as a CORBA service beside the ORB. The ORB is not aware of groups and the service can be used with any compliant CORBA implementation. The service approach follows the design of the other functionalities that have been added to CORBA through IDL-specified services, such as persistence and transactions. We followed the service approach to design and implement our *Object Group Service* (OGS) which complies with many of the requirements of the OMG *Request For Proposal* on Fault Tolerant CORBA [8] (currently under specification).

OGS shares some similarities with the CORBA Event Service [2] (e.g., both provide support for collaborative work). Nevertheless, the primary goal of OGS was *reliability* and, unlike the CORBA Event Service, OGS is not based on any

central component. In contrast, event channels are shared CORBA objects in the Event Service and thus constitute single points of failures [4].

References

[1] G. Agha and R. Guerraoui (guest editors). Theory and Practice of Object Systems, 4(2), *Special Issue on High Availability in CORBA*, 1998.

[2] K. Birman and R. van Renesse. *Reliable Distributed Computing with the Isis Toolkit*. IEEE Computer Society Press, 1993.

[3] D. Cheriton and W. Zwaenepoel. *Distributed process groups in the V kernel*. ACM Transactions on Computer Systems, 3(2), 77-107, 1985.

[4] P. Felber, R. Guerraoui, and A. Schiper. *Replicating Objects using the CORBA Event Service*. In Proceedings of the 6th IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS'97), 14-19, 1997.

[5] R. Guerraoui, P. Felber, B. Garbinato, and K. Mazouni. *System Support for Object Groups*. In Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'98), 244-258, 1998.

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

[7] OMG. *The Common Object-Request Broker Architecture: Architecture and Specification – CORBAservices: Common Object Services Specification*. http://www.omg.org.

[8] OMG. *Fault tolerant CORBA Using Entity Redundancy, Request For Proposal*. http://www.omg.org/techprocess/meetings/schedule/Fault_Tolerance_RFP.html.

[9] D. Powell (guest editor). Communications of the ACM, 39(4), *Special issue on Group Communication*, 1996.

The Overhead of OGS

Measuring the overhead of OGS comes down to compare the cost of invocations through OGS (from a client to a group of objects), with the cost of using plain CORBA invocations (of a single object). This overhead varies in function of the group size, the communication protocol used, the locality of objects, and the degree of transparency provided to the application programer. There is a tradeoff between transparency, availability, and efficiency.

In the current OGS implementation, multicast communication is achieved using reliable protocols on top of CORBA's standard Internet Inter-ORB Protocol (IIOP). Our performance measurements have been performed with the C++ version of OGS, compiled with VisiBroker 3.2 [3].

System Configuration

Testing took place on a local 10 Mbit Ethernet network, interconnecting 13 Sun SPARCstations running Solaris 2.5.1 or 2.6, under normal load conditions. Among these workstations, there were four Sun UltraSPARC 30 (250 Mhz processor, 128 MB of RAM), and nine Sun UltraSPARC 1 (170 Mhz processor, 64 MB of RAM). For tests involving up to four hosts, only the UltraSPARC 30 workstations were used. All the client and server applications were located on different hosts, except the OGS daemon which was located on the same host as the client. The tests have been run with the *TCP_NODELAY* option that sets all sockets to immediately send requests, instead of buffering them and sending them in batches.

Test Models

Our performance tests use the various semantics provided by OGS: total order, reliable, and unreliable; and three different modes of invocations (Figure 5): untyped invocations with the OGS library, untyped invocations with the OGS daemon, and typed invocations with the OGS daemon. The arrows in the figure represent the invocation path followed by the requests and the replies. We have implemented two total order algorithm: the first one, based one a consensus algorithm, can order several messages at once and provides a higher throughput when many clients are issuing requests to a group in parallel; the second one, an optimistic algorithm based on a sequencer, provides better latency but cannot order several messages at once and is more costly upon failure of the sequencer. These algorithms are respectively called "Total Order" and "Optimistic Active Replication" in the rest of this section.

Table 1 and Figures 6, 7, and 8 summarize the main results of our experiments. The test program operates as follows: a single client executes several rounds, in each of which it issues a fixed number of synchronous invocations (typically 100).[4] The group size varies from one to ten members. The client waits for a single reply from the servers before issuing the next invocation. The total time of each round is divided by the number of invocations issued during the round to obtain the latency of a single two-ways invocation, as experienced by the client. We kept the value of the best round.

[4]Since there is only one client, invocations are not performed concurrently and hence this test is not a measure of the total throughput of OGS.

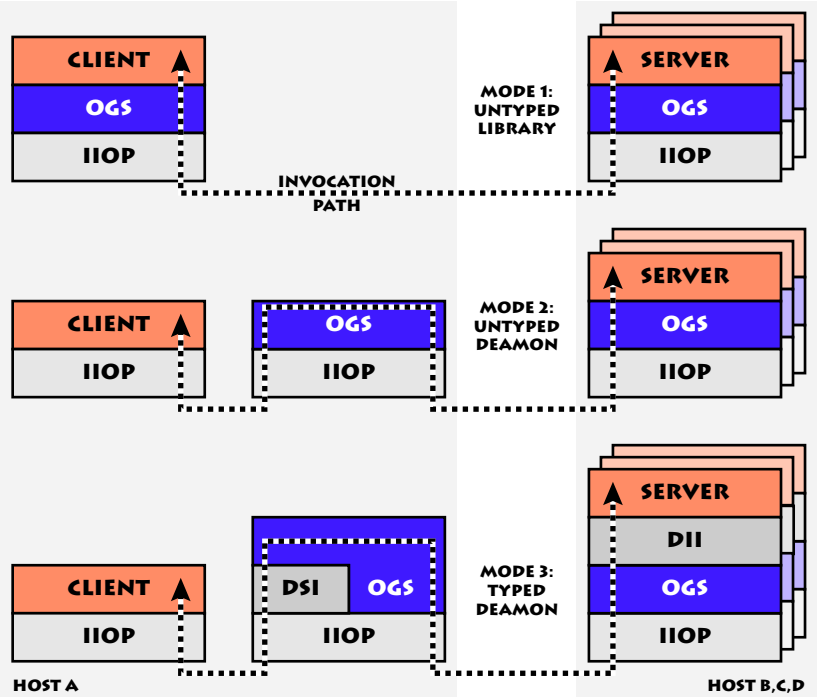

Figure 5: Three Test Models for Client Multicast Invocations

The Cost of Multicast

Figure 6 illustrates the cost of the different OGS untyped invocation primitives, with the library execution style (mode 1 in Figure 5) and different group sizes. This figure shows that the cost of total order and reliable multicast primitives grow faster than the cost of the other primitives. This is due to the fact that the former primitives are based on a simple reliable multicast algorithm, the complexity of which is $O(n^2)$ for n participants. In contrast, our optimistic active replication algorithm has been optimized so that is does not use a reliable multicast primitive [1]. Its cost grows linearly, similarly to unreliable multicast.

The Cost of Indirection

Figure 7 compares the latency of invocations issued through OGS, with a corresponding invocation issued directly through the ORB. It illustrates the additional cost induced by the extra indirection through OGS, as well as the cost of additional group members. OGS invocations are performed using the library execution style with untyped invocations. The invocation sent directly through the ORB is a standard two-way request issued through static stubs and skeletons.

Figure 7 shows that the cost of passing through the service is slightly less than 2 milliseconds per invocation. This overhead is fixed and does not depend on the number of participants: sending an unreliable request to two objects is *not* twice slower than invoking a single object.

Exec. style	*Semantics*	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**	**#9**	**#10**
Untyped OGS / Library	**Total Order**	2.80	7.57	9.61	13.65	18.81	28.07	38.33	47.42	60.97	79.80
Untyped OGS / Library	**Opt. Active Repl.**	2.77	4.74	4.98	5.92	6.58	8.15	8.97	10.42	11.49	13.01
Untyped OGS / Library	**Reliable**	2.70	3.03	3.36	4.72	7.82	13.28	18.22	23.00	27.83	34.60
Untyped OGS / Library	**Unreliable**	2.70	2.90	3.14	3.49	4.24	5.11	5.75	6.46	7.67	8.67
Untyped OGS / Daemon	**Total Order**	3.78	8.60	10.51	14.80	19.91	29.75	39.50	51.02	63.18	82.81
Untyped OGS / Daemon	**Opt. Active Repl.**	3.69	5.89	6.29	7.19	7.51	9.17	10.60	11.91	12.22	13.18
Untyped OGS / Daemon	**Reliable**	3.66	4.14	4.17	6.72	9.32	14.12	19.91	24.60	28.35	36.81
Untyped OGS / Daemon	**Unreliable**	3.69	4.01	4.05	4.89	5.29	5.90	7.07	7.86	9.11	10.05
Typed OGS / Daemon	**Total Order**	23.56	29.75	33.18	36.64	45.15	52.00	64.67	76.62	92.51	107.27
Typed OGS / Daemon	**Opt. Active Repl.**	23.48	24.73	29.65	31.19	32.69	33.86	37.14	42.13	45.15	49.73
Typed OGS / Daemon	**Reliable**	23.24	24.88	28.61	30.38	34.67	39.92	43.21	51.19	60.97	70.99
Typed OGS / Daemon	**Unreliable**	23.33	24.07	26.83	27.79	30.92	32.13	35.65	39.95	43.21	47.15
ORB	**Unreliable**	0.88									

Table 1: Performance of Multicast Invocations with Various Group Sizes and Execution Styles (ms./inv.)

The fixed cost of OGS consists basically of the following actions: OGS accepts client requests and builds a message that it multicasts to all servers; on the server-side, OGS extracts the data from the message, passes it to the server, gets the return value, and builds the reply message for the client; when the first replies arrives, OGS extracts the result and returns it to the client (the other replies are discarded). We experienced that most of this overhead comes from the additional request marshaling and unmarshaling as a result of the indirection through OGS.

The Cost of Type Transparency

Type transparency makes it possible to reuse existing applications without having to modify the client. In the current version of OGS, typed communication is available only for the daemon execution style. Figure 8 compares the latency of untyped totally ordered requests (library and daemon execution styles) with that of typed requests. This figure illustrates that there is a fixed overhead of about 1 millisecond for using the daemon. This corresponds to the latency of a single two-way invocation through the ORB. The typed version of OGS adds an overhead of about 20 milliseconds. This overhead results from the use of CORBA's dynamic interfaces (DSI and DII) for type transparency.

Although these dynamic interfaces are powerful since they permit the developer to receive and send requests on interfaces not known at compile time, they process requests in an interpretive maneer, and they extensively use values of type `any`. 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. Some measurements on VisiBroker 3.2 have shown that the extraction of complex structures from a value of type `any` costs more that a remote invocation through the ORB [1]. Since the DSI and the DII are used only once on the client and the server side, their overhead is *fixed* and does not depend of the group size or the complexity of the protocol.

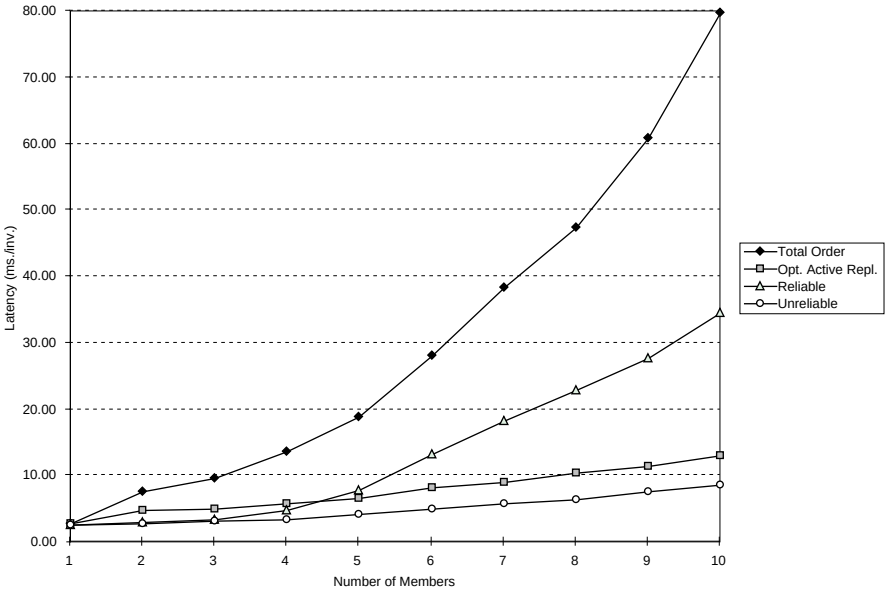

Figure 6: Comparison of OGS Multicast Primitives

References

[1] P. Felber. *The CORBA Object Group Service: A Service Approach to Object Groups in CORBA.* PhD thesis, Swiss Federal Institute of Technology, Lausanne, 1998.

[2] OMG. The Common Object-Request Broker Architecture: Architecture and Specification - CORBAservices: Common Object Services Specification. OMG

[3] Visigenic. *Visibroker for C++ 3.2 Programmer's Guide.* Visigenic Software, Inc., 1998.

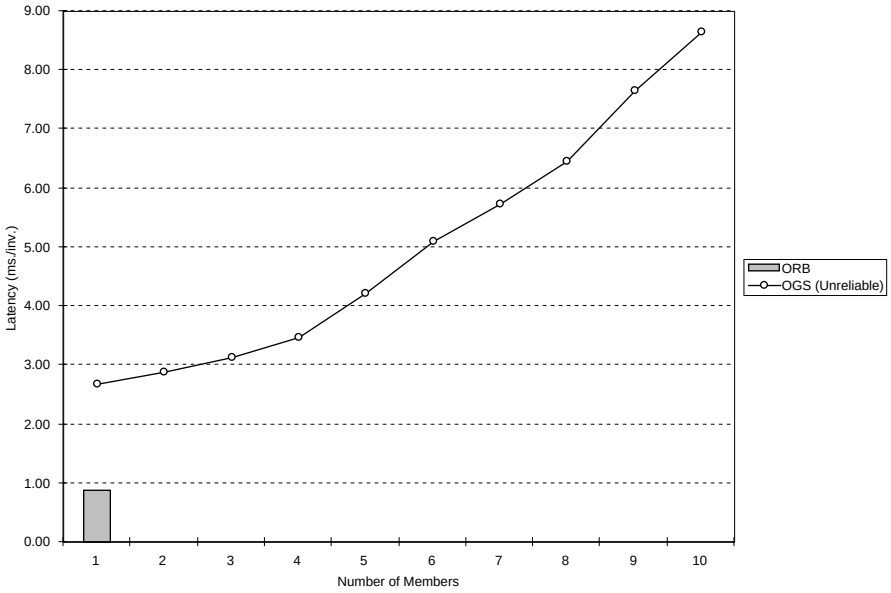

Figure 7: The Cost of Indirection in OGS

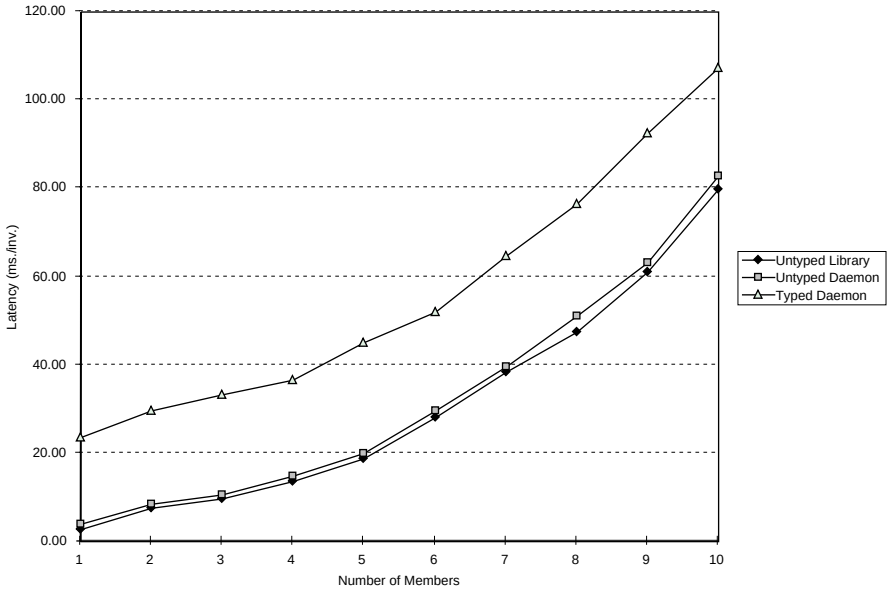

Figure 8: Untyped vs. Typed Communication

Putting OGS to Work

We illustrate below the use of OGS on three application examples: a replicated abstract calculator as an example of a critical component in a financial application, the computation of a Mandelbrot set as an example of a parallel application with load balancing capabilities, and a reliable distributed chat as an example of a collaborative work application.

For the sake of simplicity and space limitation, we only describe the code that is related to OGS. Furthermore, all error-handling code has been removed from our examples. These examples have been tested with VisiBroker 3.2. The complete code of these examples can be found at `http://lsewww.epfl.ch/OGS/`.

Replicated Calculator

The calculator maintains a current value (accumulator), and clients perform arithmetical operations on this value. The available operations are: *add*, *subtract*, *multiply*, *divide*, and *clear*. Some operation pairs are not commutative (e.g., *add* and *multiply*), making total ordering of requests necessary. A practical application of such an abstract calculator is to maintain the balance of a bank account. Additions correspond to deposits, subtractions to withdrawals, and multiplications to interest computation.

The application is composed of two parts: the servers that perform computations and the clients that perform arithmetical operations on the replicated calculator. The shared state of the servers is the calculator's current value. OGS transparently preserves the consistency of the calculator's state by delivering requests in the same order to all replicas.

The IDL interface of the calculator application is given below. It simply consists of a `Calculator` interface, with one attribute and four operations. The `value` attribute represents the current state of the calculator, and should not be modified directly; therefore, it is declared as *read only*. The calculator's interface inherits from OGS' `Groupable` interface.

```
1  // IDL
2  interface Calculator : mGroupAdmin::Groupable {
3    readonly attribute float value;
4
5    float add(in float nb);
6    float sub(in float nb);
7    float mul(in float nb);
8    float div(in float nb);
9    void clr();
10 };
```

The C++ interface of the calculator server is defined below. The implementation of the arithmetical operations are inlined in the C++ interface definition. To keep the code simple, no check is performed for overflow and division by zero. The current value of the calculator is kept in the `value_` member variable.

```
1  // C++
2  class Calculator_i : _sk_Calculator
3  {
4    CORBA::Float value_;
5
6  public:
7    // Operations from interface ''Groupable''
8    virtual CORBA::Any* get_state();
9    virtual void set_state(const CORBA::Any& state);
10   virtual mGroupAdmin::OperationSemanticsSeq* operation_semantics();
```

```
11      // ...
12
13      // Operations from interface ''Calculator''
14      virtual CORBA::Float value()              { return value_; }
15      virtual CORBA::Float add(CORBA::Float x) { value_ += x; }
16      virtual CORBA::Float sub(CORBA::Float x) { value_ -= x; }
17      virtual CORBA::Float mul(CORBA::Float x) { value_ *= x; }
18      virtual CORBA::Float div(CORBA::Float x) { value_ /= x; }
19      virtual void clr()                        { value_ = 0.0; }
20  };
```

The calculator server also implements operations from the `Groupable` interface. In particular, it provides support for state transfer (`get_state` and `set_state`). Since the state of the calculator consists of a single floating point value, the state transfer methods are very simple. In addition, the calculator server specifies the semantics associated to each operation by implementing `operation_semantics`. A read-only operation does not need to be totally ordered if the clients do not care about receiving a slightly out of date value; therefore we use an *unordered* reliable multicast for the `value()` operation. Note that, since the operation semantics defaults to total order, only the operations that have a weaker semantics need to be specified in `operation_semantics()`.

```
1   // C++
2   CORBA::Any* Calculator_i::get_state()
3   {
4     // Pack the state into an any
5     CORBA::Any* a = new CORBA::Any();
6     *a <<= value_;
7     return a;
8   }
9
10  void Calculator_i::set_state(const CORBA::Any& a)
11  {
12    // Extract the state from an any
13    a >>= value_;
14  }
15
16  mGroupAdmin::OperationSemanticsSeq* Calculator_i::operation_semantics()
17  {
18    // Return the semantics associated to each operation
19    mGroupAdmin::OperationSemanticsSeq* oss = new OperationSemanticsSeq(1);
20    oss->length(1);
21    (*oss)[0].name_ = "_get_value";
22    (*oss)[0].ordering_ = mGroupAccess::RELIABLE;   // Unordered!
23    return oss;
24  }
```

Parallel Mandelbrot

The Mandelbrot set is a fractal structure defined on the complex plane, which is traditionally displayed in a 2D picture. The computation of a Mandelbrot set is time-consuming but easy to parallelize. We adopt a client-server approach with the server providing the processing power while the client displays graphically the resulting set. To have the image displayed in "real-time" and to reduce the size of messages, the server transmits the data line by line, as soon as they are completed, to the client.

In this application, OGS is used to distribute the workload among several servers. The area of the Mandelbrot set is separated into bands, each of which is computed on a different server. The set is subdivided into n bands, where n is the number of members in the group. Each member uses its position in the current view to decide which band to compute, and sends lines to the client

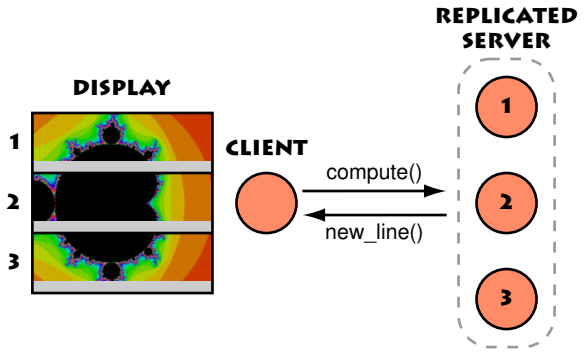

Figure 9: Mandelbrot

which is updated asynchronously. The server is written in C++ and the client in Java. Figure 9 illustrates the architecture of the Mandelbrot application. OGS gives the illusion of one single server whereas the work is actually distributed to several effective servers, making it possible to increase the parallelism degree without the knowledge of the client. Note that in this application the server is replicated in terms of processing, but not state.

The IDL interface of the Mandelbrot application is given below. It is made of a `Mandelbrot` module containing two interfaces: `Client` and `Server`. The server's interface inherits from OGS' `Groupable` interface.

```
1  // IDL
2  module Mandelbrot
3  {
4    const long LINE_SIZE = 400;
5    typedef long Line[LINE_SIZE];
6
7    interface Client {
8      oneway void new_line(in long number, in Line data);
9    };
10
11   interface Server : mGroupAdmin::Groupable {
12     void compute(in Client client,
13                  in long top, in long left,
14                  in long height, in long width,
15                  in long iter, in long zoom);
16   };
17 };
```

The C++ interface of the Mandelbrot server is defined as follows:

```
1  // C++
2  class Mandelbrot_i : _sk_Mandelbrot::_sk_Server
3  {
4    int nb_members_;
5    int position_;
6
7  public:
8    // Operations from interface ''Groupable''
9    virtual void view_change(const mGroupAccess::GroupView& new_view);
10   // ...
11
12   // Operations from interface ''Mandelbrot''
13   virtual void compute(Mandelbrot::Client_ptr client,
14                        CORBA::Long top, CORBA::Long left,
15                        CORBA::Long height, CORBA::Long width,
16                        CORBA::Long iter, CORBA::Long zoom);
17 };
```

The most meaningful operation of the `Groupable` interface for the Mandelbrot server is the `view_change()` operation. This operation is invoked by OGS to give the current view to the application which then decides the area of the Mandelbrot set to compute. This information is stored in the `nb_members_` and `position_` member variables and is updated in the code of the `view_change` operation.

```
1  // C++
2  void Mandelbrot_i::view_change(const mGroupAccess::GroupView& new_view)
3  {
4    nb_members_ = new_view.composition_.length();
5    position_ = new_view.my_index_; // Contains position of this member
6  }
```

The C++ code (not given here) used to compute the Mandelbrot set processes only the area allocated to the local server, and updates the client each time a new line is completed.

Distributed Chat

Our distributed chat application is similar to the well-known *Internet Relay Chat* (IRC) program, but without the centralized server that receives and forwards messages. It allows participants all over the Internet to talk to one another in real-time. Users can join chat channels and send message to these channels. All participants listening to the channel receive the messages. Each participant has a nickname sent along with the messages to identify the originator of the message by other users.

The distributed chat application does not have a pure client/server design. Chat channels are mapped to groups, and participants are both clients and servers of these groups. The member objects are *not* copies of a replicated object; they are distinct entities that collaborate by exchanging messages using group communication.

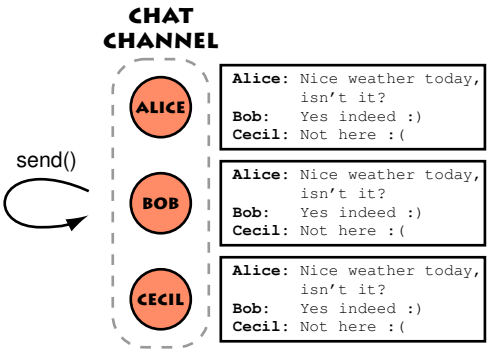

Figure 10: Distributed Chat Application

The general architecture of the distributed chat application is illustrated in Figure 10. Messages issued by a channel member are multicast to all users listening to the channel. The programming model is symmetrical: after a message has been multicast, the originator will receive and deliver its own message like all other group members.

The architecture of this application differs from the other examples, in that there is only one type of object (the chat object), which is both client and server. The IDL interface of the chat application is composed of a single operation used to send a message to the current group, and a read only attribute that stores the nickname of the local participant.

```
// IDL
interface Chat : mGroupAdmin::Groupable {
  readonly attribute string nickname;

  void post(in string sender, in string msg);
};
```

The C++ interface of the chat server is defined as follows:

```
// C++
class Chat_i : _sk_Chat
{
  char *nickname_;

public:
  Chat_i(char *nickname) : nickname_(nickname) {}

  // Operations from interface ``Groupable''
  virtual void view_change(const mGroupAccess::GroupView& new_view);
  // ...

  // Operations from interface ``Chat''
  virtual char *nickname();
  virtual void post(in string sender, in string msg);
};
```

A chat object is stateless. It only receives messages, prints them to the screen, and forgets them. The most meaningful operation of the `Groupable` interface is view change notification: each time the membership changes, the chat object displays the list of participants.

```
// C++
void Chat_i::view_change(const mGroupAccess::GroupView& new_view)
{
  cout << "Participants:" << endl;
  for(int i = 0; i < new_view.composition_.length(); i++) {
    Chat_var chat = Chat::_narrow(new_view.composition_[i]);
    cout << i << ": " << chat->nickname() << endl;
  }
}
```