Replication of CORBA Objects

Pascal Felber*, Rachid Guerraoui, and André Schiper

Swiss Federal Institute of Technology
Operating Systems Lab.
CH-1015 Lausanne, Switzerland

Abstract. Distributed computing is one of the major trends in the computer industry. As systems become more distributed, they also become more complex and have to deal with new kinds of problems, such as partial crashes and link failures. While many middleware architectures have emerged to answer the growing demand in distributed technologies, most of them do not provide any kind of fault tolerance mechanisms. In this paper, we discuss the addition of object group support to CORBA. We describe three approaches: integration, interception, and service, and we argue is favor of the latter. We present the architecture of an Object Group Service (OGS) that provides for fault tolerance and high availability through object replication. This service enables the application developer to deal with invocations to replicated objects in a completely transparent way. We describe the major components of OGS: messaging, monitoring, consensus, group membership, and group multicast. We finally discuss the implementation of the service and its performance.

1 Introduction

1.1 Context

The last few years have seen the emergence of several programming environments that greatly reduce the complexity of developing distributed software. These environments, regrouped under the term *middleware* as they appear between application programs and operating system services, provide high-level facilities for developing distributed applications without having to deal with low-level details, such as remote communication and object location. They use object-oriented concepts to abstract the complexity of the system and promote modularity and reusability. These environments offer frameworks for integration of heterogeneous distributed components. Examples of these middleware architectures are OMG's CORBA [OMG98a] and Microsoft's DCOM [Ses97].

1.2 Motivations

Existing object-oriented middleware environments essentially deal with point-to-point invocations. While this interaction style complies with the invocation

* Current affiliation is Oracle Corp., 1000 SW Broadway, Suite 1200, Portland, OR 97205.

model of object-based systems, some types of applications need to invoke *several objects* at once. In particular, communication facilities have shown to be necessary for *object replication*.

One-to-many interactions can be provided by group communication. Group communication manages groups of objects and provides primitives for sending messages to all members of a group, with various reliability and ordering guarantees. A group constitutes a logical addressing facility: messages can be issued to a group without having to know the number, identity, or location of individual members. Groups can be used for *replication*: a set of replicas constitutes a group, but are viewed by clients as a single entity in the system.

CORBA does not currently provide any support for replicated objects. Systems that require replication support must do all the necessary design and implementation in the application, with no guarantee of either interoperability or portability. Providing group support in CORBA reduces the burden on designers and implementers of fault tolerant applications. Applications benefit from the power of groups (high availability, fault tolerance, etc.) while preserving the key features of object-oriented middleware environments (simple development process, distribution transparency, component integration, etc.). This work pioneers the ongoing efforts of the OMG to support fault tolerance in CORBA using entity redundancy [OMG98b].

1.3 The Object Group Service

In this paper, we present different approaches followed by existing systems to support replication in the CORBA middleware environment. All these systems use *group communication* mechanisms to provide replication. We classify these approaches according to three categories: the *integration* approach, the *interception* approach, and the *service* approach, and we argue that only the latter complies with the modular, component-based architecture promoted by CORBA.

We present the design of a CORBA *Object Group Service* (OGS), that provides replication support through group communication for standard off-the-shelf CORBA environments. The OGS environment specifies an architecture and a set of interfaces for object groups. The OGS architecture does not define a single monolithic component; it is decomposed into several CORBA services that provide various facilities for reliable distributed computing, and that are used for the actual implementation of group communication. In particular, the *Object Monitoring Service* provides distributed failure detection mechanisms, and the *Object Consensus Service* allows several CORBA objects to solve distributed agreement problems. This decomposition into several independent components promotes modularity and reusability, and extends the use of OGS to areas other than group communication. OGS is currently available for Orbix [ION97] and VisiBroker [Vis98].

1.4 Roadmap

This paper is organized as follows: Section 2 presents background concepts about CORBA, object replication, and group communication. Section 3 discusses related work about object groups and replication support in CORBA. Section 4 describes our *Object Group Service* (OGS), and its different components. Section 5 gives an overview of the different configuration options of OGS. Section 6 presents the performance of the current OGS implementation. Finally, the concluding section summarizes the main characteristics of OGS and relates it to the ongoing standardization efforts for object group support in CORBA.

2 Background

2.1 The Object Management Architecture

The *Object Management Architecture* (OMA) [OMG98a], specified by the *Object Management Group* (OMG), is a conceptual infrastructure for building portable and interoperable software components, based on open standard object-oriented interfaces. Figure 1 shows the five major parts of the OMA reference model.

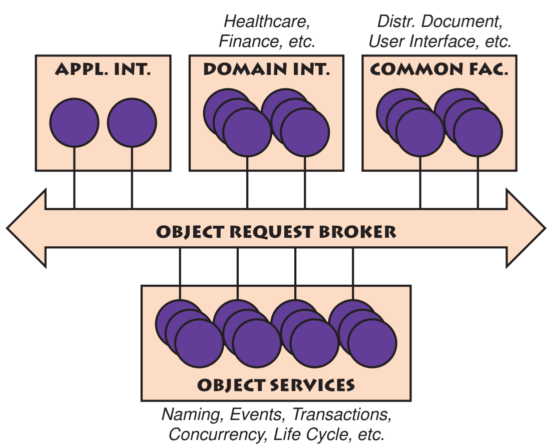

Fig. 1. The OMA architecture

Commercially known as CORBA, the *Object Request Broker* (ORB) is the communication heart of the OMA. It enables heterogeneous objects to transparently invoke remote operations and receive replies in a distributed environment. The ORB also provides the environment for managing objects, advertising their presence, and describing their metadata. 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.

The *Object Services* 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.

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

2.2 Replication

The idea of using redundancy as a mean of masking the failures of individual components dates back to von Neumann [vN56]. With redundant copies, a replicated entity can continue providing services in spite of the failure of some of the copies, without affecting its clients. Redundancy may appear at different points in the architecture, such as redundancy of computational and storage resources; redundancy of communication links between these resources and their clients; redundancy of transient application components.

In distributed systems, the two best known replication policies are *active* and *primary-backup* replication. A replicated object is represented by a set of copies. This set may be static or dynamic. Static replication means that the number and the identity of the copies do not change during the lifetime of the replicated object. Dynamic replication is more powerful since copies may be added or removed at runtime.

2.3 Group Communication

Groups were first introduced in the V-Kernel [CZ85], as a convenient addressing mechanism. They were later extended to handle replication in the Isis system [Bir93]. The key idea of *group communication* is to gather a set of processes or objects into a logical group, and to provide primitives for sending messages to all group members at the same time with various ordering guarantees. A group constitutes a logical addressing facility since messages can be issued to groups without having to know the number, identity, or location of individual members. Groups have proven to be very useful for providing *high availability* through *replication*: a set of replicas constitutes a group, viewed by clients as a single entity in the system. 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 the modification of the group composition 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.

3 Object Groups in CORBA

Although no *standard* support for object groups is available, several systems provide replication of CORBA objects based on group communication. This section introduces the different design alternatives for managing object groups in CORBA, evaluates them, and describes how they have been used by existing systems.

3.1 A Coarse Classification

The CORBA object model defines an object as an entity with a well-defined interface that may be remotely invoked using an *object reference*. An object reference is an *"object name that reliably denotes a particular object. An object reference identifies the same object each time the reference is used in a request, and an object may be denoted by multiple, distinct references"* [OMG98a]. This means that the CORBA specification does not permit an object reference to designate a set of objects, and it does not provide ways for clients to invoke several objects at once using an object reference. CORBA only deals with point-to-point remote invocations.

The absence of mechanism that permits to multicast requests to groups of CORBA objects complicates the design and implementation of many applications that have requirements for reliability and high-availability. Only the CORBA *Object Transaction Service* (OTS) provides fault tolerance to a limited extent through transactional mechanisms, but unlike group-based systems, it does not achieve high availability.

During the last couple of years, several systems have been developed to augment CORBA with groups. We can classify these systems according to three main categories, each of which represents a different approach to group communication in CORBA [FGG97]:

1. The *integration approach* integrates an existing group communication system within an ORB.
2. The *interception approach* intercepts messages issued by an ORB and maps them to a group communication toolkit.
3. The *service approach* provides group communication as a CORBA service beside the ORB.

We now describe these three approaches and existing systems that implement them. A more exhaustive discussion of these approaches can be found in [Fel98].

3.2 Integration Approach

With the integration approach (Fig. 2), the ORB functionality is enhanced by a group communication toolkit. The ORB directly deals with object groups and references to object groups. CORBA requests are passed to the group communication toolkit that multicasts them from clients to replicated servers, using proprietary mechanisms. The group toolkit is “integrated” into the ORB.

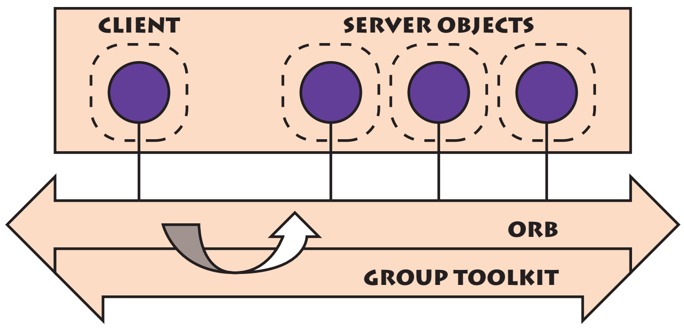

Fig. 2. Integration approach

Two systems are based on the integration approach: Orbix+Isis, [II94] which is a commercial system from Iona Technologies and Isis Distributed Systems[1], and Electra [Maf95], a free CORBA programming environment written in C++. In these systems, only one of the replies is returned by default to the client in order to keep the group invocation transparent.[2] However, a client aware of the group is also able to access all the replies, if necessary. The basic idea is to extend the IDL language mapping and to generate two types of functions from IDL definitions: (1) standard functions that conform to the language mapping and (2) special functions with sequences of values for *out* and *inout* parameters. The client uses the function with the signature that corresponds to its needs. If the client is not aware of groups, it uses only standard functions.

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 fully CORBA compliant, and results in proprietary systems. In addition, the integration approach adopted by Electra and

[1] Orbix+Isis has been recently discontinued.
[2] This makes sense if groups are used for replication.

Orbix+Isis uses non-standard language-specific constructs,[3] causing application code to be also not portable. Furthermore, most existing group toolkits are not adapted to a CORBA environment: they are designed for process groups instead of object groups, and they do not provide adequate primitives for group-to-group communication, making it difficult to support client replication [GFGM98].

An interesting variant of the integration approach, which has not been explored by existing systems, would consist in providing a second object adapter in addition to the *Basic Object Adapter*. We would then have the basic adapter for standard objects and a dedicated adapter — a *Group Object Adapter* — for group member objects. This approach would be more compliant with the CORBA specification, since it would isolate all group management functionalities in a single non-standard component.

3.3 Interception Approach

With the interception approach (Fig. 3), the ORB is not aware of replication. ORB requests formatted according to the IIOP protocol are intercepted transparently on client and server sides using low-level interception mechanisms; they are then passed to a group communication toolkit that forwards them using group multicasts. This approach does not require any modification to the ORB, but relies on OS-specific mechanisms for request interception.

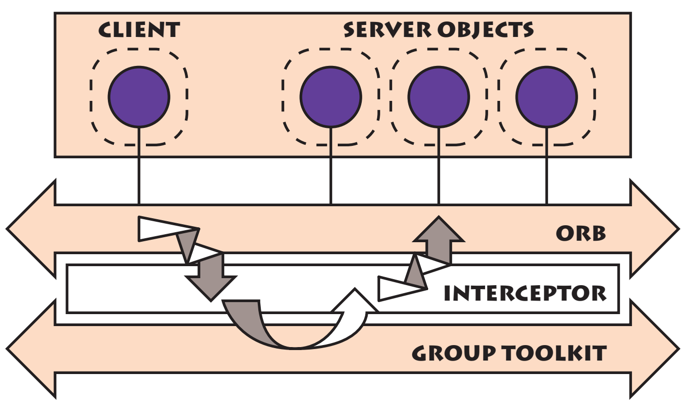

Fig. 3. Interception approach

Eternal [MMSN98] is a system based on the interception approach. It operates on top of the Unix operating system, and works with standard CORBA implementations. Eternal exploits the Unix `/proc` interface to monitor the operating system calls made by an object to establish an IIOP connection over

[3] These constructs are quite different in Electra and Orbix+Isis; application code written for one is not portable to the other.

TCP/IP, and to communicate IIOP messages over that connection. Eternal intercepts the IIOP messages before they reach TCP/IP, and passes them to the Totem group communication system [MMSA+96], which multicasts the messages to the object groups containing the replicas. Any group communication system that provides guarantees similar to those of Totem can be used instead.

The drawback of this approach lies mainly in its low-level interception mechanisms: while these mechanisms can be implemented on most Unix systems, they are not portable to any operating systems.

3.4 Service Approach

The service approach, which has been adopted and developed in the context of OGS, provides explicit group support through a CORBA service [FGG96] (Fig. 4). Unlike the integration approach, a CORBA service is mostly specified in terms of IDL interfaces, and does not depend on implementation language constructs. The ORB is not aware of groups, and the service can be used with any compliant CORBA implementation. The service approach complies with the CORBA philosophy, by promoting modularity and reusability. Note that a service does not have to be centralized: it can be made of several objects, located at different hosts on the network, that work together at providing the complete service. This is important when dealing with fault tolerance, since a centralized service would be a single point of failure.

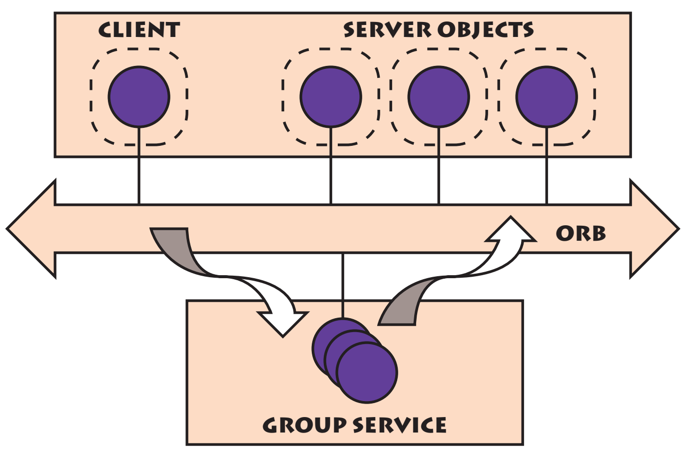

Fig. 4. Service approach

CORBA's open architecture allows us to easily define and implement new services. The process of specifying a new service consists in isolating the requirements, choosing the right abstractions, and specifying the interfaces for these abstractions. The OMG has published guidelines for designing object services and their interfaces [OMG97].

The service approach has also been adopted in the *Distributed Object-Oriented Reliable Service* (DOORS) [CHY+98], currently in development at Bell Labs. Our work on CORBA fault-tolerant services has significantly influenced many of the proposals that have recently been made to the OMG in the context of its undergoing standardization effort towards fault-tolerant CORBA [OMG98b].

4 The Object Group Service

4.1 Overview

The *Object Group Service* (OGS) is a CORBA service that provides group management and communication facilities in a CORBA environment. It is composed of a set of generic IDL-specified interfaces. With OGS, clients can send invocations to object groups without knowing the number and identities of group members. In addition, OGS provides support for *transparent group invocations*, allowing clients to invoke operations on object groups as if they were invoking singleton objects. OGS is based only on standard CORBA mechanisms and is thus portable to any compliant ORB implementation. OGS may be used from any programming language that is supported by CORBA, or from any system that supports the CORBA *Internet Inter-Orb Protocol* (IIOP).

More precisely, the OGS environment specifies an architecture and a set of IDL interfaces for object group support, as well as a set of object services that provide various facilities for reliable distributed computing. OGS uses a component-oriented approach. Basic units of functionality are packaged as CORBA services, which represent distributed components. These services, which include distributed agreement protocols and detection of remote component failure, are used in the implementation of the group communication primitives (see Sect. 4.3). Note that the relationship between these different services are *implementation dependencies*, and are not expressed in the IDL interfaces.

4.2 Transparency

A common goal of object-oriented middleware environments is to hide the low-level mechanisms used for remote invocations and object management as much as possible, to let the developer focus on the application-specific problems. Similarly, when working with object groups, a desirable property is to hide the complex mechanisms used for group communication. This section describes the concept of transparency in the context of OGS and the underlying mechanisms used to provide it.

Transparency in OGS. *Transparency* is the property of a system to be invisible, i.e., the degree to which application programs are unaware of the system. Transparency appears at different levels in the OGS architecture [GFGM98] and may be classified as follows:

– *Plurality transparency:* the application objects have the illusion that they deal with singleton objects, although they interact with object groups.
– *Behavior transparency:* the application objects are not aware of the replication policy and the protocols run by OGS in order to make the system behave consistently.
– *Type transparency:* requests are performed via direct method invocations on the server's interface, rather than by explicit calls to a group communication API (e.g., via a `multicast()` operation), and explicit packing and unpacking of operation parameters into and from messages. Type transparency means that the application does not need to perform type conversions.

Full transparency is achieved if the application program is completely unaware of the group service. OGS provides full client-side transparency, without even requiring the client code to be recompiled [Fel98].

The Benefits of Transparency. Transparency provides a number of facilities which make it useful, if not necessary, to develop distributed applications with object groups. Some of the benefits of transparency are outlined below.

– *Ease of use.* Applications do not need to be written with replication in mind.
– *Less error prone.* Application programmers do not need to write error prone code for inserting and extracting data into and from requests.
– *Reuse of existing code.* Existing client code can be reused without modification. Server code requires only minor modifications. Objects from existing frameworks can be made groupable by using multiple inheritance of interfaces and implementations.
– *Encapsulate behavior in the group.* OGS knows about the operations invoked on object groups, and can associate different semantics to distinct requests on behalf of the object group. The protocols may vary depending on the invoked operation without the client's knowledge.

The Limitations of Transparency. Client transparency is generally considered as a "good thing" for object replication. However, server transparency has several drawbacks for complex systems. When dealing with replica configuration, failure detection mechanisms, or advanced synchronization between replicas, explicit group management support is required. In some cases, transparent replication can make systems suffer reduced performance due to the lack of control over how replication is performed.

4.3 OGS Components

OGS implements group communication using a set of CORBA services. These services are not organized in a layered architecture, but as a set of orthogonal components with *usage relationships* between each other (*component-oriented* approach). Figure 5 presents an abstract view of the major components defined in the OGS architecture. Although this is not clearly visible on the figure, each component is specified independently and they all interact with each other through the ORB. The application may use any of these components directly. These components are:

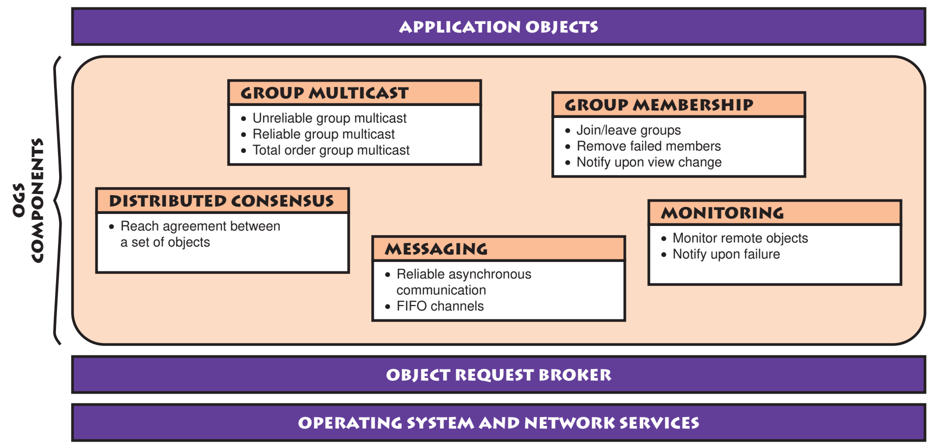

Fig. 5. Overview of the OGS architecture

1. A *Messaging Service* that provides non-blocking reliable point-to-point and multicast communication.
2. A *Monitoring Service* that monitors objects and provides failure detection mechanisms in CORBA.
3. A *Consensus Service* that allows a set of application objects to solve the so-called distributed consensus problem [CT96]. This service is used to implement group multicast and membership protocols.
4. A *Group Service* that provides both *group multicast* and *group membership*.

The dependencies between these components are *implementation-specific*, and do not appear in the IDL interfaces of the services. All these components are described in details in [Fel98]. In the following, we focus on the group service only.

4.4 The Group Service

The *group service* is the core of the OGS environment. It is the service that actually provides object group support, and which the application programmer has to deal with. It implements two functionalities:

– *Group membership* manages the life cycle of object groups. It maintains the updated list of all correct group members. It provides support for joining and

leaving groups, view change notification, and state transfer. A group membership service is generally associated with a failure detection mechanism for detecting group member failures.
- *Group multicast* provides support for sending multicast invocations to all the members of a group, with various reliability and ordering guarantees.

The group multicast and group membership services interact closely. In particular, multicast operations are defined on object groups rather than on sets of unrelated objects, thus involving group membership. Therefore, both services are contained in the set of interfaces forming the group service.

Group Membership. The role of a group membership service is to manage memberships in a distributed system on behalf of the processes that compose it. OGS supports dynamic groups, i.e., the composition of the groups can change over time. New members can join an existing group, explicitly leave it, or may be implicitly removed from the group because of a failure. Objects that wish to join a group do so by contacting the membership service, which updates the list of group members. Once admitted to the group, an object may interact with other group members. Finally, if the object fails or leaves the group, the membership service will again update the list of group members. Dynamic group membership involves two kinds of protocols:

- A *view change protocol*, which is run each time the composition of a group changes. It ensures that every correct member of the group receives a *view change notification*, indicating the new composition of the group as a list of group members with mutually consistent rankings. View changes are totally ordered with each other.
- A *state transfer protocol*, which is an atomic operation occurring during view change, when a new member joins an existing group. It consists in obtaining the state from a current group member, and giving it to the new member. This protocol ensures that the state of all group members is kept consistent upon membership changes. The view change protocol can terminate only after a state transfer is completed.

The group membership interfaces define how OGS and group members interact. They are essentially composed of two types of objects (Fig. 6): (1) service-specific[4] *group administrator* objects that enable group members to change their status in the group (e.g., join and leave the group); and (2) application-specific *groupable* objects that enable OGS to call back to the application for view change and state transfer protocols. The `GroupAdministrator` interface is implemented by the service, and is used as a black box by the application. A group administrator is assigned to a single group at creation time, but there may be several

[4] We distinguish *service-specific* objects, defined and implemented by the service and accessed by the application through their IDL interface, from *application-specific* objects, implemented by the application and supporting some interface defined by the service.

group administrators on the same host or in the same process. The `Groupable` interface must be implemented by application objects that want to be members of a group.

Group Multicast. Group multicast provides primitives for sending invocations to groups instead of singleton objects. OGS provides a rich set of group multicast primitives, adapted to various types of applications. OGS implements groups as open structures, and allows non-member objects to issue multicast invocations to groups. Multicast primitives can be classified according to their degree of *reliability* and their *ordering* guarantees. In addition, OGS provides two communication models: *untyped* and *typed* invocations.

Multicast Reliability. Reliable multicast is a key mechanism for developing replicated applications. In the context of group communication, it means that all correct group members deliver the same set of messages *(agreement)*, that this set includes all messages multicast to the group by correct objects *(validity)*, and that no spurious messages are ever delivered *(integrity)* [HT93].

OGS provides both unreliable and reliable multicast primitives. The unreliable primitives may be useful for read-only operations, but should be used with caution, since it may corrupt the state of the group if all messages are not received by every member. Reliable multicast, in itself, does not ensure that group consistency is preserved; it is generally combined with an ordering guarantee.

Multicast Ordering. In addition to reliability, message ordering is an important concern when dealing with one-to-many communication. The state of an object generally depends on the order in which it receives requests, and some special multicast primitive must be used to maintain a consistent ordering of requests received by all group members. The current version of OGS provides unordered, FIFO, and totally ordered multicast primitives.

Multicast Replies. Just as an invocation to a singleton object may return a value, a multicast invocation may return a set of values (one from each target object). The client may want to wait only for the first reply, for several replies, or even for no replies. OGS provides the following options for the number of replies expected by the client:

- *All-replies multicast invocations* wait for replies from all group members.
- *Majority-of-replies multicast invocations* wait for a majority of replies from the group members.
- *One-reply multicast invocations* wait for the first reply from any group member.
- *Zero-reply (synchronous) multicast invocations* wait for the operation to complete on one group member, but does not return any reply to the client.
- *One-way multicast invocations* do not wait for any reply from the group members, and do not block the client's execution thread.

With either majority-of-replies or all-replies multicast invocations, OGS might need to modify the expected number of replies if a server crashes while the invocation is processed. The zero-reply synchronous invocation style is useful for flow control to avoid that the object group becomes congested; it blocks the sender until the message is processed, allowing the receivers time to decongest.

Untyped Interfaces and Explicit Invocations. The untyped invocation interface of OGS allows clients to send untyped messages to group members. An object that wishes to send a multicast to an object group must explicitly pack the data of the message into a value of type `any`, and pass it to OGS, which will perform the multicast. On the server side, OGS delivers this message by passing the `any` value to group members through their `deliver()` operation.

This invocation interface is flexible since the client can place any kind of data in the message, and can easily specify the semantics associated with the multicast invocation and the expected number of replies. The drawback of this model is that the application programmer must *explicitly* pack all parameters associated with the server's invocation in a message (marshaling), and extract these parameters on the server side (unmarshaling), which is a painful and error-prone task. In addition, this untyped invocation interface exposes group communication to the client application, while a common goal to most group-based systems is to hide groups from clients.

Typed Interfaces and Transparent Invocations. The typed invocation interface of OGS provides group transparency to clients, which can issue invocations to object groups as if they were invoking singleton objects. The client directly invokes operations of the server's interface using static stubs, and OGS delivers multicasts by directly invoking the relevant operation of the server, using static skeletons. OGS transparently filters messages and returns a single reply to the client. Of course, typed communication does require that all servers implement the same IDL interface.

Although less flexible than the untyped invocation interface, this model is much easier to use. The application developer does not need to perform the marshaling and unmarshaling of the request (these operations are performed transparently by OGS), and can benefit from the type safety of CORBA's static invocation interface. The client cannot specify the semantics to associate with a multicast invocation but, as we will see in next section, this may be advantageously replaced by server-specified invocation semantics.

Server-Specified Invocation Semantics. As mentioned in Sect. 4.2, OGS provides server-specified invocation semantics: the server may associate specific semantics to each individual operation of its interface when using the typed version of OGS.[5] For instance, if an operation does not change the state of the server (read-only operation), the server may decide to deliver the invocation without

[5] By default, OGS uses a total order multicast for invocations with unspecified semantics.

ensuring total order. Since the server implements the operations, it knows their properties and the client does not need to be aware of them. Furthermore, the server can optimize client requests based on their semantics; for instance, two update operations do not have to be totally ordered with each other if they are commutative (e.g., they modify disjoint parts of the server's state). This approach is an improvement over the traditional model where a client asks for the strongest ordering guarantees for a message when it is unsure of the exact semantics of the associated operation.

4.5 OGS Components and Interactions

OGS combines support for group membership and group multicast in a single set of interfaces. Figure 6 presents a simplified high-level view of OGS components. OGS interfaces are classified according to three categories, associated with the different views of the service:

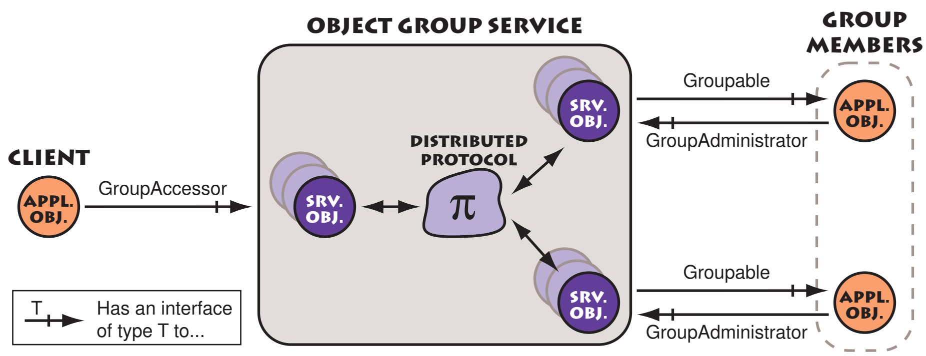

Fig. 6. OGS components overview

1. *Client interfaces* (`GroupAccessor`) allow clients to interact with object groups.
2. *Member interfaces* (`GroupAdministrator`) are a superset of client interfaces, and allow servers to manage the group's life cycle (e.g., join and leave groups).
3. *Service interfaces* (`Groupable`) define interfaces that the member objects must implement for OGS to issue callbacks to them.

Group accessors and administrators are service objects. Performing a multicast to the group initiates a distributed 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.

The IDL interfaces of the group services are described in [Fel98]. OGS interfaces are defined in two modules: the first one, `mGroupAccess`, is used by group clients and defines operations to communicate with object groups; the second module, `mGroupAdmin`, is used by group members and defines operations for administrating object groups. Interfaces from the `mGroupAdmin` module inherit from interfaces defined in `mGroupAccess`. Inheritance is used because the client depends on less functionality than the full interface offers.

Figure 7 illustrates the interfaces and operations of the complete OGS specification in a class diagram. Inheritance relationships clearly show that server-side interfaces inherit from client-side interfaces.

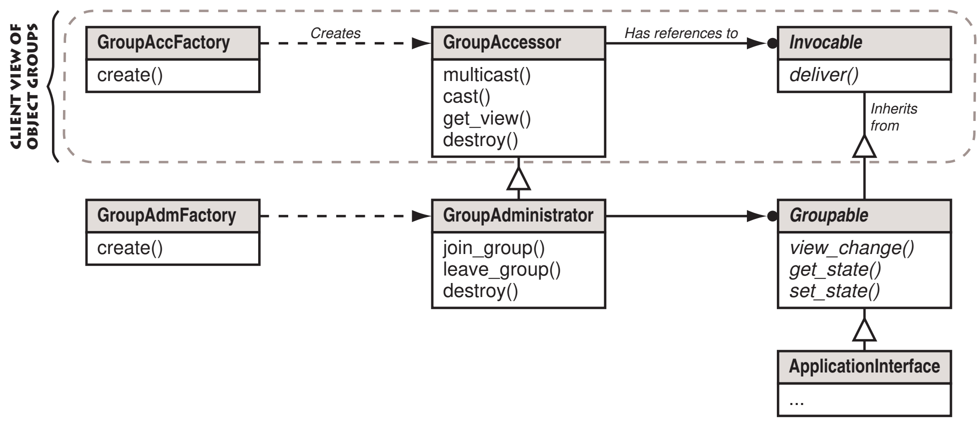

Fig. 7. Class diagram of OGS interfaces

The interaction diagram of Fig. 8 illustrates the view change and state transfer mechanisms on two members. Initially, the group is empty. A first member object joins the group (1), and receives the new view (2). A second member object joins the group (3), which is not empty anymore, leading to both a state transfer protocol (4, 5), and a view change notification (6).

5 OGS Configuration

Invocations to object groups are performed by OGS. Client messages are sent via group accessor objects, and server messages are delivered via group administrator objects. Group accessors and administrators are *service objects* that form the visible part of the OGS runtime system, which is presented in this section. The application developer can configure this runtime system in a number of ways, leading to different degrees of flexibility, efficiency, transparency, or reliability. Examples of OGS programs using various configurations can be found in [Fel98].

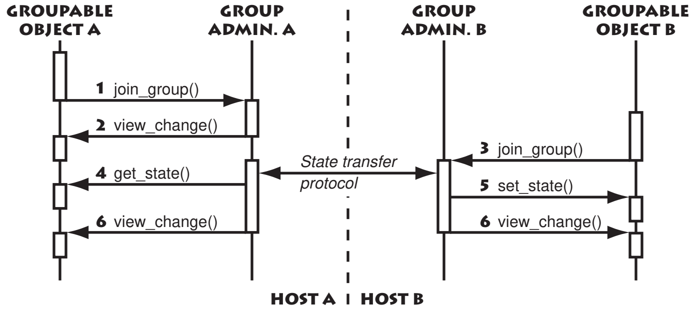

Fig. 8. Interaction diagram of view change and state transfer

To conceal efficiency and flexibility, OGS provides two execution models: a *linkable* model and a *daemon* model. In the first model, 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 model, the service objects are located in another process — the `OGSd` daemon program — which may be on the local or on a remote host.

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 Fig. 9). This execution model is more efficient since inter-process communications are more costly than invocations between objects located in the same process [GFGM98]. However, it enforces the code of the application to be written with the same programming language as the library and to support multi-threading.

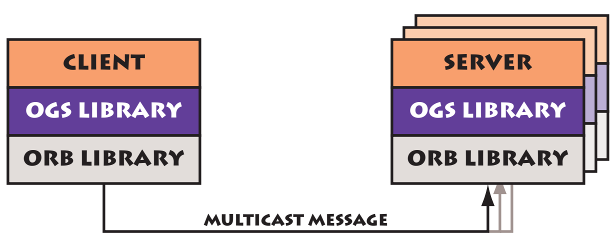

Fig. 9. The co-located execution model

The *daemon* execution model, with two separate processes, 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 another programming language, such as Smalltalk, to use the

C++ or Java service. Figure 10 illustrates the use of the OGS daemon on the client side, while the service is linked with the application on the server side.[6]

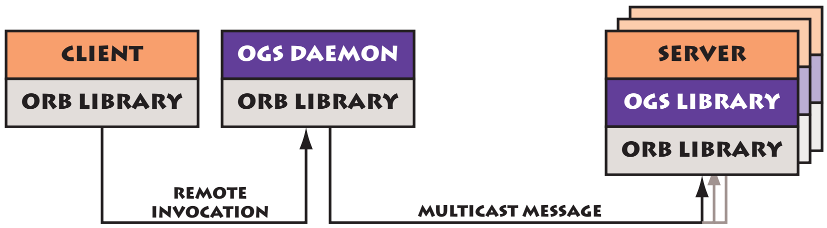

Fig. 10. The remote execution model

The co-located execution model mainly benefits from its efficiency, while the remote execution model provides language heterogeneity. But the choice of the execution model also affects reliability. As matter of fact, with the co-located execution model, the application does not need to care about potential link failures or crashes of service objects, since OGS cannot fail independently from the application.

If the application and the service objects are located on different machines, the application must handle network, machine, and OGS daemon process failures. Therefore, using a remote daemon requires the communication link between the application and the daemon to be reliable (this hypothesis is realistic in a local area network).

6 OGS Performances

This section presents the performance of the OGS implementation on VisiBroker [Vis98] (some performance measurements of OGS on Orbix [ION97] may be found in [FGS98]).[7] We focus on client multicast invocations, i.e., the cost of invocations through OGS, from a client to a group of objects. We compute the overhead of using OGS over using plain CORBA invocations, and we investigate the source of this overhead.

These measurements focus on OGS performance. They do not analyze the intrinsic cost of invocations going through the ORB in detail. Exhaustive performance measurements and analysis of CORBA latency with different ORBs may be found in [GS98].

[6] Note that the servers could also use an OGS daemon.

[7] Due to some limitations in the current version of Orbix, we could not perform all the tests presented in this section with the Orbix version of OGS.

6.1 System Configuration

Our performance measurements have been performed with the C++ version of OGS, compiled with VisiBroker 3.2. 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 (all workstations were running X Windows, as well as several user applications such as netscape or emacs). 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-no-delay* option that sets all sockets to immediately send requests, instead of buffering them and sending them in batches.

6.2 Test Scenarios

Our performance tests evaluate the latency of multicast invocations issued by a client to an object group when no failure occurs. These invocations use the various semantics provided by OGS: total order (using two different algorithms [Fel98]), reliable, and unreliable; and three different modes of invocations: untyped invocations with the OGS library, untyped invocations with the OGS daemon, and typed invocations with the OGS daemon. The group size varies from one to ten members. The client waits for a single reply from the servers, except with the optimistic active replication algorithm, with which the client waits for a majority of replies.

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). The client waits for a reply from each request 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 invocation. We kept the value of the best round. Since there is only one client, invocations are not performed concurrently and OGS cannot benefit from its consensus-based total order algorithm that can order several requests at once. Therefore, this test is not a good measure of the total throughput of OGS with this algorithm.

6.3 Performances Measurements

In Sect. 4.2, we have classified transparency according to three categories: behavior transparency, plurality transparency, and type transparency. We analyze the costs of the OGS architecture according to this classification: we first evaluate the cost of the various multicast primitives of OGS (behavior); plurality transparency is analyzed in details in [Fel98]; we finally evaluate the performance overhead induced by the use of dynamic typing facilities in OGS (type).

The Cost of Behavior. The semantics of multicast invocations depend on the behavior of an object group. Figure 11 illustrates the cost of the different OGS untyped invocation primitives, with the library execution style and different group sizes.

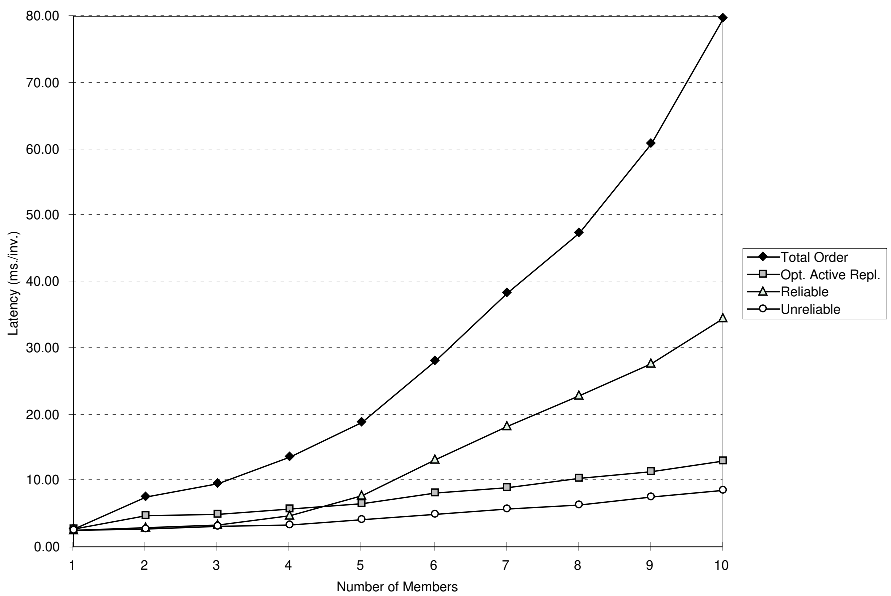

Fig. 11. Comparing OGS multicast primitives

This figure shows that the total order and reliable multicast primitives grow faster than the other primitives. This is due to the fact that the former primitives are based on a simple reliable multicast algorithm; the complexity of the number of messages for this algorithm is $O(n^2)$ for n participants.[8] In contrast, the optimistic active replication algorithm [Fel98] has been optimized so that is does not use the same reliable multicast primitive when there is no failure. Its cost grows thus linearly, similarly to the unreliable multicast primitive.

Note that the cost of an invocation to a single object issued directly through VisiBroker is about three times lower than the same invocation going through OGS to a group with a single member.

The Cost of Typing. Type transparency is an important feature of OGS because it hides groups from the application developer, and 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

[8] Note that this cost could be reduced by using a "smarter" algorithm.

style. Figure 12 compares the latency of untyped totally ordered requests (library and daemon execution styles) with that of typed requests.

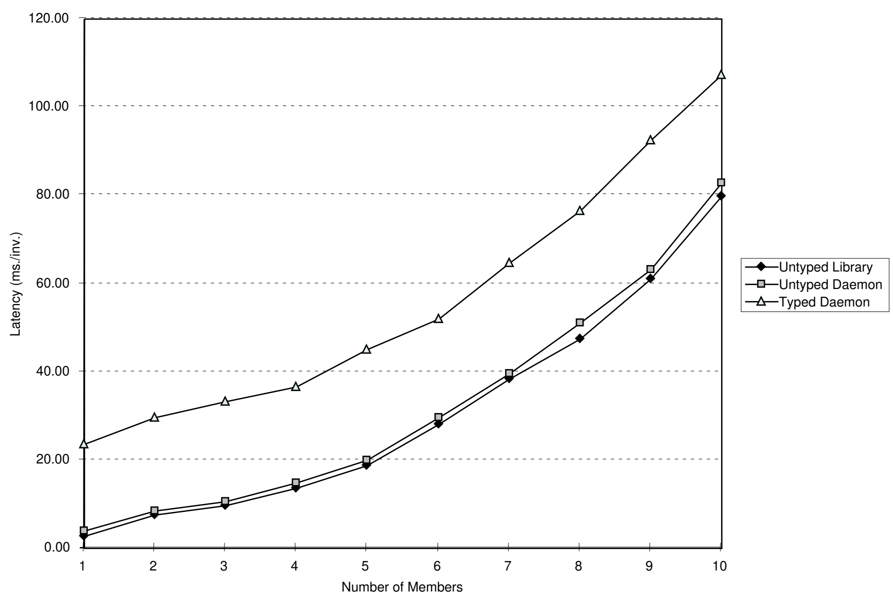

Fig. 12. Untyped vs. typed communication

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 the DSI and the DII for type transparency, and is *independent* of the group size. This is due to the fact that the DSI and the DII are used only once on the client and the server side.

Another source of overhead comes from request management. When profiling OGS, we noticed that a non-negligible part of the time required for remote invocations is spent in constructing requests. We also observed that working with untyped `any` values has a significant impact on performance. 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.

7 Conclusion

Distributed computing is one of the major trends in the computer industry. As systems become more distributed, they also become more complex and have to deal with new kinds of problems, such as partial crashes and link failures. Many of these problems can be solved by replicating critical components.

To answer the growing demand in distributed technologies, several middleware environments have emerged during the last few years. These environments however lack support for one-to-many communication primitives, which are necessary to implement object replication. Augmenting a middleware architecture by adding support for object groups provides straightforward support for replicating distributed objects.

We have presented here an open architecture for object group support in CORBA, based on a component-oriented approach. This architecture is generic and can be applied to other middleware environments than CORBA. The *Object Group Service* (OGS) defines an object group model and specifies a set of interfaces adapted to group communication in middleware environments. OGS is decomposed into several components that conspire to provide higher-level services. This decomposition promotes modularity and reusability, and complies with the CORBA architectural model.

OGS provides support for dynamic group membership and for group multicast with various reliability and ordering guarantees. In addition, OGS proposes several execution styles and various levels of transparency. CORBA objects can easily be made groupable using interface inheritance. Client applications can communicate with object groups in the same way as they do with singleton objects. A prototype implementation of OGS is freely available at `http://lsewww.epfl.ch/OGS/`.

References

[Bir93] K.P. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):36–53, December 1993.

[CHY+98] P.E. Chung, Y. Huang, S. Yajnik, D. Liang, and J. Shih. Doors: Providing fault tolerance for corba applications. In *IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*, September 1998.

[CT96] T.D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

[CZ85] D.R. Cheriton and W. Zwaenepoel. Distributed process groups in the V kernel. *ACM Transactions on Computer Systems*, 3(2):77–107, May 1985.

[Fel98] P. Felber. *The CORBA Object Group Service: A Service Approach to Object Groups in CORBA*. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, 1998. Number 1867.

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

[FGG97] P. Felber, B. Garbinato, and R. Guerraoui. *Special Issues in Object-Oriented Programming*, chapter Towards Reliable CORBA: Integration vs. Service Approach, pages 199–205. dpunkt-Verlag, 1997.

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

[GFGM98] 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)*, 1998.

[GS98] A. Gokhale and D. Schmidt. Measuring and optimizing CORBA latency and scalability over high-speed networks. *IEEE Transactions on Computers*, 47(4):391–413, April 1998.

[HT93] V. Hadzilacos and S. Toueg. *Distributed Systems*, chapter 5: Fault-Tolerant Broadcasts and Related Problems, pages 97–145. Addison-Wesley, 2nd edition, 1993.

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

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

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

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

[MMSN98] L.E. Moser, P.M. Melliar-Smith, and P. Narasimhan. Consistent object replication in the Eternal system. *Theory and Practice of Object Systems*, 4(2):81–92, 1998.

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

[OMG98a] OMG. *The Common Object Request Broker: Architecture and Specification*. OMG, February 1998.

[OMG98b] OMG. *Fault tolerant CORBA Using Entity Redundancy, Request For Proposal*. OMG, Apr 1998. http://www.omg.org/techprocess/meetings/schedule/Fault_Tolerance_RFP.html.

[Ses97] R. Sessions. *COM and DCOM: Microsoft's Vision for Distributed Objects*. John Wiley & Sons, 1997.

[Vis98] Visigenic. *VisiBroker for C++ 3.2 Programmer's Guide*. Visigenic Software, Inc., Mar 1998.

[vN56] J. von Neumann. Probabilistic logics and the synthesis of reliable organisms from unreliable components. In C. E. Shannon and J. McMarthy, editors, *Automata Studies*, pages 43–98. Princeton University Press, 1956.