Replicating Objects using the CORBA Event Service?

Pascal Felber Rachid Guerraoui André Schiper

Ecole Polytechnique Fédérale de Lausanne
Département d'Informatique
CH-1015 Lausanne, Switzerland

Abstract

The Common Object Request Broker Architecture (CORBA) is becoming a standard for distributed application middleware, but no support is currently provided to handle object replication. In this paper we discuss the relevance of using OMG's Event Service for replicating objects in a CORBA environment and compare it to other approaches.

1 Introduction

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

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

A natural way to provide fault-tolerance and high availability consists in replicating critical components. Although the OMG mentions a future *Object Replication Service*, we are not aware of any concrete work in this direction.

While the CORBA services design guidelines focus on service reuse and on avoiding duplicated functionality, it is legitimate to wonder whether a new object service should be defined for providing replication, or whether an existing CORBA service already provides an appropriate paradigm.

In this paper we evaluate the use of the Event Service for achieving high availability in the context of CORBA. This service provides an abstraction similar to the group abstraction, which has shown to be very powerful in supporting high availability through replication.

The paper is organized as follows: Section 2 recalls some background on replication. Section 3 summarizes related work on replication in CORBA. Section 4 describes briefly the CORBA Event Service. Section 5 presents how the Event Service can be used for providing replication in CORBA, and Section 6 discusses some architectural issues. Finally, Section 7 presents some concluding remarks.

2 Replication: Background

Replicating components allows to make distributed systems more reliable (the system will survive to the failure of some copies of the replicated component) [GS97]. However, replicating a component involves maintaining all copies consistent with each other, i.e., ensuring that the state of all copies is the same[1].

The two best known replication policies are *active* and *passive* replication. With active replication, all copies of the replicated object handle each request and return a reply; this requires the operations on the replicated object to be deterministic. With passive replication, only one copy (generally called the *primary*) executes the request and updates the other copies upon completion. The failure of a copy is easier to handle with active replication, since it is transparent to the client. On the other hand, the failure of the primary with passive replication is not transparent to the client. Furthermore, passive replication requires specific protocols for the primary to update the state

[1] This requirement may be alleviated for some types of applications that do not require strong consistency.

of the other copies.

A replicated object is represented by a set of copies. This set may be static or dynamic. Static replication requires that the number and the identity of the copies do not change during the lifetime of the replicated object. Dynamic replication is more powerful in that it allows to add or remove copies at run-time.

3 Replication in CORBA

The concept of groups has 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. Several systems have successfully experimented with the support of replication through group communication. Among these systems are Horus [RBC93], Isis [BCJ+90], Phoenix [Mal96], Totem [MMSA+96], and Transis [ADKM92].

In [FGG96], we discussed the problem of adding fault-tolerance to CORBA. We distinguished different approaches, illustrated in Figures 1, 2, and 3: (1) the *integration approach*, which consists in integrating an existing group communication system within an Object Request Broker (ORB), (2) the *interception approach*, which consists in intercepting messages issued by an existing ORB and mapping them on a group communication toolkit, and (3) the *service approach*, which provides group communication as a CORBA service beside the ORB.

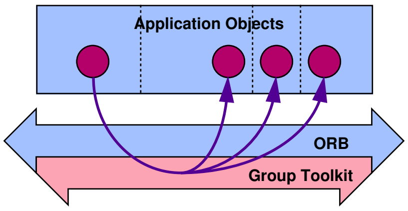

Figure 1: Integration Approach

The integration approach (Figure 1) has been adopted in Orbix+Isis [II94] and Electra [Maf95]. Although appealing for its ease of development (there is no need to build a new group system from scratch) and its transparency (an object group is not distinguishable by a client from a singleton object that implements the same interface), this approach is not CORBA compliant and results in proprietary systems. Orbix+Isis and Electra are based on proprietary group communication toolkits (Isis and Horus respectively), which do not provide adequate primitives for group-to-group communication, and thus do not support client replication. This implies that replicated CORBA objects are tied to a server role.

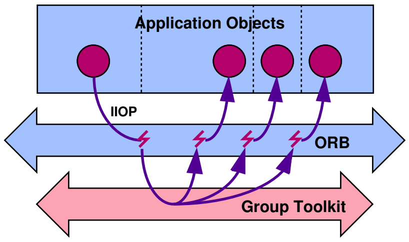

Figure 2: Interception Approach

The interception approach (Figure 2) does not require any modification to the ORB. Eternal [NMMS97] uses this mechanism to provide support for replicated objects in commercial ORBs.

The *Eternal Interceptor* captures *Internet Inter-ORB Protocol* (IIOP) messages and the *Eternal Replication Manager* maps them onto the Totem multicast group communication system. Eternal intercepts system calls before they reach the kernel, by performing a continual trace on the `/proc/pid` file (where `pid` is the process identifier assigned by Unix to each CORBA object). These system calls are then modified appropriately and passed to the process group interface for communication over Totem. Objects that support the same interface become automatically part of the same object group.

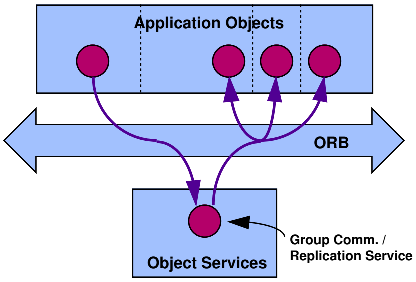

Figure 3: Service Approach

The service approach (Figure 3) complies with the CORBA philosophy, by promoting interoperability and portability. It follows the design of other functionalities that have been added to CORBA through services, such as persistence and transactions. These services are specified in OMG's *Interface Definition Language* (IDL) and have been adopted as CORBA standards. An Object Group Service can be considered a fundamental building block for CORBA applications with reliability requirements.

We have used the service approach to define an *Ob-*

ject Group Service (OGS) [FGG96], which is an *object-oriented* environment providing building blocks for *reliable distributed* applications running in a *heterogeneous* system. It provides primitives for active replication with dynamic groups. The OGS does not require any changes (or even extensions) to the CORBA specification. It is portable in the sense that it can run on top of any standard CORBA 2.0 ORB, and it does not rely on any ORB vendor-specific feature. The OGS is modular as it is itself made of other (sub-)services, each built as a set of CORBA objects with IDL interfaces.

The OGS manages groups of CORBA objects and provides primitives to communicate with these groups. Clients do not need to know the number, the identity, or the location of the members of a group. A client can issue a single request to all the group members at once, by naming the group. The OGS is inherently distributed and does not depend on any global, critical, or centralized component. The OGS interfaces provide for different levels of reliability.

While the service approach appears to be the most appropriate [FGG96], we might wonder whether an existing CORBA service does already provide a suitable paradigm for object replication, i.e., if the IDL interfaces of an existing service can be reused. In particular, the CORBA Event Service provides interfaces that allow to invoke several objects at once.

4 The CORBA Event Service

The CORBA Event Service decouples the communication between *suppliers* and *consumers* through *event channels*. Suppliers produce event data and consumers process event data. Suppliers can generate events without knowing the identity of the consumers. Conversely, consumers can receive events without knowing the identity of the suppliers. An event channel is an intervening object that allows multiple suppliers to communicate with multiple consumers asynchronously. An event channel is both a consumer and a supplier of events.

Event data is communicated from suppliers to consumers by issuing standard CORBA requests. Events themselves are *not* objects because the CORBA object model does not support passing objects by value. There are two approaches to initiating event communication between suppliers and consumers. These two approaches are called the *push model* and the *pull model*.

The push model allows a supplier of events to initiate the transfer of event data to consumers. The pull model allows a consumer of events to request event data from a supplier. In the push model, the supplier takes the initiative; in the pull model, the consumer takes the initiative.

Event channels are standard CORBA objects and communication with an event channel is accomplished using standard CORBA requests. The different communication models defined by the Event Service are illustrated in Figure 4.

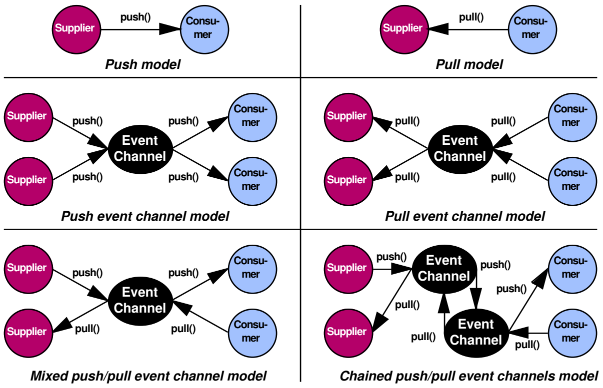

Figure 4: Event Service Communication Models

The basic Event Service model allows only to send data of type `any`[2]. Hence, it provides a message-passing type interface to event communication. The *Typed* Event Service offers an RPC-like communication model. Suppliers call operations on consumers (typed pull model) or consumers call operations on suppliers (typed pull model) using some mutually agreed IDL-described interfaces.

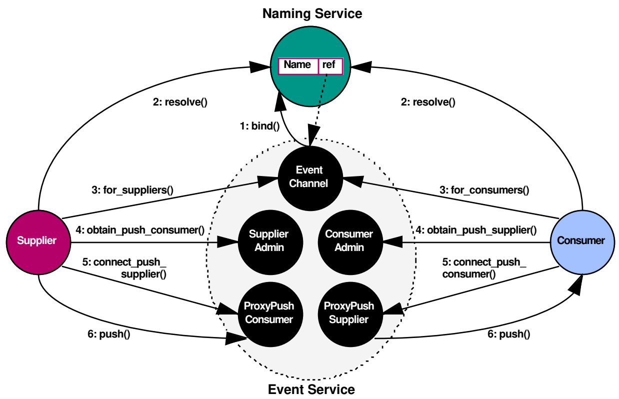

Figure 5: Connecting to an Event Channel

Binding an event channel (i.e., connecting a consumer or a supplier to an event channel) is not as

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

straightforward as it may seem; the following steps, illustrated in Figure 5, are usually required:

An event channel is created and registered with the Naming Service (1). Consumers obtain a reference to the event channel from the Naming Service (2) and register themselves with the event channel: they obtain a reference to a `ConsumerAdmin` interface using the `for_consumers()` operation of the event channel (3); then they obtain a reference to a `ProxyPushSupplier` interface using the `obtain_push_supplier()` operation of the `ConsumerAdmin` interface (4); finally they connect to the event channel using the `connect_push_consumer()` operation of the `ProxyPushSupplier` interface (5).

Supplier registration is symmetrical to consumer registration. Once they are bound to an event channel, suppliers can produce events that are forwarded by the event channel to registered consumers (6).

5 Replication with the Event Service

In this section, we present how the basic Event Service can be used for replication. This approach promotes service reuse, as advocated in the OMG design guidelines: the Event Service is already specified and several implementations are available. Furthermore, the event model is easy to understand and straightforward to use in notification-based applications like news agencies or trading systems.

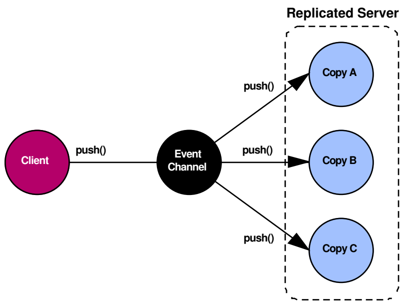

Figure 6: Using the Event Service for Replication

A natural way of using the Event Service for replication is to use the push model with one event channel: all the copies of a replicated object are consumers of the channel, while clients supply event data on this channel (Figure 6).

In particular, the typed version of the Event Service provides for almost transparent server replication: clients register to the event channel as suppliers and obtain a reference to an object that supports the same interface as the replicated server; they can then issue invocations on this interface directly. Clients do not need to know the number or location of the copies, which can change at runtime.

But the Event Service has some limitations that affect its use for replication. These limitations are presented in the following sections.

5.1 Invocation Semantics

The Event Service specification does not enforce implementations to provide guarantees concerning ordering or atomicity [DFGG97]. For instance, with a compliant Event Service implementation, the messages sent by two different suppliers may be received in a different order by different consumers. If the consumers represent different copies of a replicated object, and if the messages are not commutative, the internal state of the copies may diverge, leading to inconsistencies.

Providing strict atomic delivery between all suppliers and all consumers requires additional interfaces [OMG95b]. Different qualities of service may be provided by implementations, but they are not standardized in the Event Service. Messages are not even guaranteed to be delivered in a first-in first-out (FIFO) order.

A solution would consist in providing an implementation that extends the Event Service and overcomes these limitations. But for replication to be valuable using the Event Service, it should work with any implementation of the Event Service that complies with the specification.

5.2 One-way vs. Two-way Invocations

The Event Service only supports one-way communication, i.e., operations on the replicated server must have only *in* parameters. This restriction is too limiting for many distributed applications that require results from invocations.

Return values may be transmitted explicitely using a “reverse” event channel, but this requires clients and servers to be both consumers and suppliers.

5.3 State Transfer

Adding a new replica at run-time requires to transfer state to the new replica. Hence, the replicas must implement operations for getting and setting their state, which are not part of the Event Service interfaces.

A solution to this problem is to send a special event to an existing copy, asking it to push its state to the new copy (using untyped event communication).

A better way to deal with this problem is to combine the Event Service with another existing CORBA service, the Externalization Service, for handling the state transfer. This service defines operations for externalizing and internalizing objects, and provides straightforward support for object migration.

6 Architectural Issues

One of the most important problems is that the architectural design of the Event Service is *centralized*. Although consumers and suppliers use different interfaces for pushing/pulling event data to/from the channel, they have to invoke the same centralized event channel object in order to connect to an event channel. This event channel is a standard CORBA object that may have an interoperable object reference registered in a Naming Service, and may be accessed through IIOP by different consumers and suppliers running on top of different ORBs. IIOP provides no support for multicast communication, and therefore interoperable object references identify centralized objects. This implies that, in the Event Service architecture, the event channel is a single point of failure. Since our aim is to provide fault-tolerance through replication, we can not rely on a solution that introduces a single point of failure. There are basically three ways for decentralizing the Event Service, which are described in the next section: (1) by using IP multicast for event channels, (2) by replicating event channels, and (3) by chaining event channels.

6.1 IP Multicast

In this model, an event channel is represented by an IP multicast address (Figure 7). A supplier generates data by sending it to the multicast address. This approach is very efficient, and was adopted by IONA in their Event Service implementation, called OrbixTalk [ION96]. It is especially suitable for applications that use the *push model* of the Event Service, with *exactly one* event channel between the consumers and the suppliers.

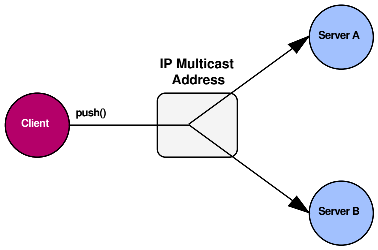

Figure 7: Decentralizing Event Channels using IP Multicast

Although this approach is efficient in LAN environments, it does not scale to multiple router hops since IP routers typically do not forward IP multicast packets and the potential for lost packets increases significantly as multiple routers are traversed. Moreover, the use of IP multicast is completely non-interoperable with other event channel implementations: CORBA IIOP does not support IP multicast and an event channel must be centralized to be designated by an interoperable object reference.

6.2 Replicated Event Channels

With this approach, event channels are replicated, and hence, are no longer a single point of failure (Figure 8). This approach, similar to the architecture of the Isis Distributed News Service [BCJ$^+$90], requires the use of specific protocols, like group communication [Bir93], to keep replicated event channel objects consistent. But systems that use this approach are proprietary and therefore inadequate in many cases.

Having highly available replicated event channels is useful for some kind of applications, especially if they provide message persistence and logging; but it is a heavy-weight solution that is not well adapted for simply providing object replication[3].

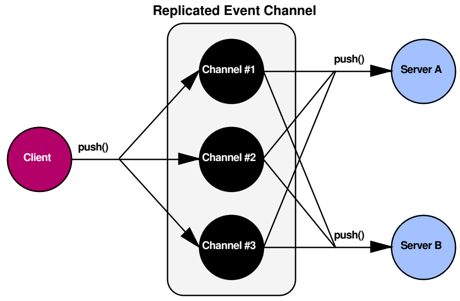

Figure 8: Decentralizing Event Channels using Replication

6.3 Chained Event Channels

Rather than having a single event channel that diffuses messages to all copies of the replicated object, we can introduce several event channels located on the client and on the server site. A client is represented as a *request-supplier* and a *response-consumer*[4], while a server is represented by a *request-consumer* and a *response-supplier* (Figure 9). This model provides two-way communication with no single point of failure. Distinct clients will generate data using distinct request-suppliers and receive replies through distinct response-suppliers.

[3] Actually, replicating event channels for providing replication is a recursive problem!

[4] These event channels reside typically in the same process.

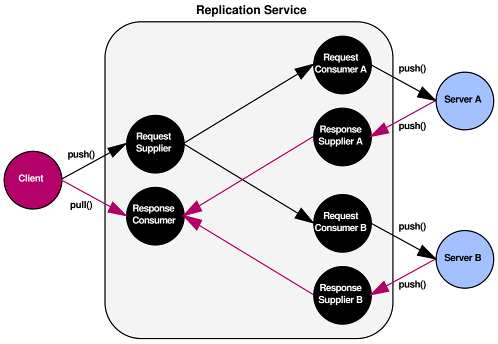

Figure 9: Decentralizing Event Channels using Chaining

The request-supplier object will perform multicast communication, possibly executing some protocol with the request-consumers for guaranteeing atomicity and ordering of messages. The response-consumer will gather multiple replies and return them to the client through a push or a pull mechanism.

This approach is fully CORBA compliant, does not modify the Event Service specification, and does not introduce a single point of failure. But extra protocols are necessary between the channel objects for ensuring atomicity and ordering of messages.

7 Conclusion

Using the Event Service for replication is attractive since the publish/subscribe model is easy to understand and it does not require the definition of a new service. But the general model introduced by the event service is not adequate as it is for replication and reliable multicast communication, because event channel interfaces define a centralized architecture, quality of service is not sufficient, and the Event Service does not take return values into account. An interesting alternative consists in providing an Event Service-like interface to a replication service.

References

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

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

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

[DFGG97] Xavier Défago, Pascal Felber, Benoît Garbinato, and Rachid Guerraoui. Reliability with CORBA event channels. In *The Third USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, pages 237–240, Portland, Oregon, July 1997. USENIX Association.

[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.

[GS97] R. Guerraoui and A. Schiper. Software-based replication for fault tolerance. *IEEE Computer*, 30(4):68–74, April 1997.

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

[ION96] IONA. *OrbixTalk Programming Guide.* IONA Technologies Ltd., Jul 1996.

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

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

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

[NMMS97] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Exploiting the internet inter-ORB protocol to provide CORBA with fault tolerance. In *The Third USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, pages 81–90, Portland, Oregon, July 1997. USENIX Association.

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

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

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