System Support for Object Groups*

Rachid Guerraoui, Pascal Felber, Benoît Garbinato and Karim Mazouni †
Computer Science Department,
Swiss Federal Institute of Technology,
CH-1015, Lausanne, Switzerland

Abstract

This paper draws several observations from our experiences in building support for object groups. These observations actually go beyond our experiences and may apply to many other developments of object based distributed systems.

Our first experience aimed at building support for Smalltalk object replication using the Isis process group toolkit. It was quite easy to achieve group transparency but we were confronted with a strong mismatch between the rigidity of the process group model and the flexible nature of object interactions. Consequently, we decided to build our own object oriented protocol framework, specifically dedicated to support object groups (instead of using a process group toolkit). We built our framework in such a way that basic distributed protocols, such as failure detection and multicasts, are considered as first class entities, directly accessible to the programmers. To achieve flexible and dynamic protocol composition, we had to go beyond inheritance and objectify distributed algorithms.

Our second experience consisted in building a CORBA service aimed at managing group of objects written on different languages and running on different platforms. This experience revealed a mismatch between the asynchrony of group protocols and the synchrony of standard CORBA interaction mechanisms, which limited the portability of our CORBA object group service. We restricted the impact of this mismatch by encapsulating asynchrony issues inside a specific messaging sub-service.

We dissect the cost of object group transparency in our various implementations, and we point out the recurrent sources of overheads, namely message indirection, marshaling/unmarshaling and strong consistency.

1 Introduction

By encapsulating a set of entities that cooperate to achieve some common goal, the group abstraction has proven to be very convenient for distributed programming, particularly for achieving continuous availability (fault-tolerance) through replication. Although the group abstraction is intuitive, the underlying techniques required to implement group communication and group membership pose difficult problems. Most of the papers dedicated to object groups mainly describe specific techniques to integrate object group supports in particular systems, e.g., Psync [35], Amoeba [44], Emerald [6], Arjuna [32], and CORBA [33, 10, 1]. The motivations and limitations of various design and implementation choices are however rarely discussed.

The aim of this paper is to discuss, through two experiences, general issues on (1) how to build system support for object groups and (2) how to build such support using standard object mechanisms.

1.1 Objectives

The experiences we describe in this paper had two common objectives:

- *Object group protocols.* The first objective was to build a library of distributed protocols that support object replication through the use of group mechanisms (group membership and group communication). The aim here was to ensure object continuous availability and preserve replica consistency despite concurrency and failures. We considered a distributed system model where nodes (machines) may crash and recover but do not behave maliciously. When a node crashes, all its processes stop performing actions.
- *Group transparency.* Another major objective was to provide the ability to transparently plug a group protocol underneath object invocations, within an application that was written without fault-tolerance in mind. This *group transparency* feature means that one can view a group of replicas as a single object. The main difference is the ability for the group to tolerate failures. An experimented programmer should however be capable of changing the default group protocol with one that better suits the application semantics (e.g., use *passive replication* instead of *active replication* [23]), or even build her/his own group protocol.

1.2 Experiences

Our first experience (1992-1996) started in the context of the *GARF* project, financed by the Swiss National Science

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

†Current affiliation: Union Bank of Switzerland, Zurich.

To appear in OOPSLA'98: the ACM Conference on Object Oriented Programming Systems, Languages and Applications, Vancouver, Canada, October 1998.

Foundation (SPP IF 5003-034344) and resulted in the development of two prototype systems, *GARF* [1] and *Bast* [2], both providing support for Smalltalk object groups. Our second experience (1995-1998) was performed in the context of the European Esprit projects OpenDREAMS (project 20843) and OpenDREAMS II (project 25262), and resulted in the development of a *CORBA Object Group Service*.

1. *Smalltalk object groups.* We started in the first experience by building support for Smalltalk object replication using the Isis group toolkit [5]. We implemented a full prototype, named *GARF* and illustrated its use by replicating a distributed diary manager application in a local area network [15, 16, 24]. The reflective facilities of Smalltalk made it easy to build a simple two-level *mailer/encapsulator* model that separates replication aspects from application functional features. Relying on an underlying process group communication toolkit revealed however a fundamental mismatch between the semantics of process groups and object groups. Roughly speaking, process group toolkits (such as Isis) usually assume a large grain client/server model, whereas object based programs are usually made of fine grain dynamic entities that alternatively play the roles of client and server. The mismatch posed several problems such as *mixed scheduling*, *duplicated invocations* and *group proliferation.* In a first step, we limited the effect of the mismatch by building a specific *adaptor* to interface Smalltalk objects and Isis processes, but we had to make strong assumptions on the nature of the target applications (e.g., no multi-server request atomicity). In a second step, we gave up Isis and we built (from scratch) our own open object oriented distributed protocol framework, named Bast, specifically adapted to support object groups. Bast is open in the sense that distributed protocols such as failure detection and multicast are considered as first class entities directly accessible to the programmer, and it is a framework in the sense that most of those protocols can be customised simply by implementing call-backs. We implemented Bast in Smalltalk, first using inheritance as the only structuring mechanism and later by *objectifying* distributed algorithms through the *Strategy Design Pattern* [14] to promote flexible group protocol composition.

2. *CORBA object groups.* The second experience aimed at supporting CORBA object groups [37], in a language independent manner. The goal was to support the coexistence of groups of objects written in different languages and even within the same group, of objects written in different languages. We designed and implemented a CORBA *Object Group Service (OGS)* along the lines of other CORBA services, such as the *Object Transaction Service* and the *Event Service* [38]. We reused our Smalltalk experience in OGS by relying on a *mailer/encapsulator* communication model to achieve separation of concerns and an object oriented distributed protocol framework to support object groups. OGS was first implemented in C++ using Orbix [28] and was shown to be very effective in supporting object replication in a language independent manner. We later ported *OGS* on Visibroker [43]. Although OGS was specified following the OMG guidelines and our target Object Request Brokers (Orbix and VisiBroker) are widely considered as CORBA compliant, it was not possible to fully port OGS code from one ORB to another. We thus ended up with slightly different implementations of *OGS* for Orbix and Visibroker. The reason of this limited portability is the mismatch between the asynchrony of group protocols and the synchrony of CORBA basic mechanisms. Indeed, most of distributed protocols underlying object groups are asynchronous, whereas CORBA mainly provide *synchronous Object Remote Procedure Calls.* Full asynchrony is not guaranteed by the CORBA specification 2.1, and although some form of asynchrony can be obtained on specific ORB implementations using event handling or multi-threading, none of these solutions is fully portable. We restricted the impact of the mismatch by encapsulating asynchrony issues inside a specific messaging sub-service, which is the only non-portable component of OGS.

Interestingly, in all (non-optimised) implementations that came out of our projects, full group transparency introduces an overhead factor of around 10, when compared with a remote invocation of a single object. This overhead can be viewed as the price to pay for *encapsulating plurality* [6]. We dissect this overhead by separating the various underlying costs: (1) the cost of message indirection (i.e., *invocation transparency*), (2) the cost of marshaling (i.e., *argument transparency*), and (3) the cost of the total order multicast protocol (i.e., *behaviour transparency*).

1.3 Lessons

The aim of this paper is not to describe the architectures or the implementations of the systems that resulted from our experiences, namely GARF, Bast and OGS. Several papers have been published on these systems ([15, 16, 24, 11, 19, 13]). Our objective here is rather to point out general observations that we drew from building those systems. Even though our experiences were mainly directed towards the use of groups for providing continuous availability (fault-tolerance) through object replication, we believe that the observations we point out apply also to other usages of groups, such as load balancing, on-line software life cycle and cooperative work. In fact, we believe that most of those observations concern general developments of object based distributed systems.

1. The mismatch between process groups and object groups applies not only to Isis, but to all process group toolkits we know about (see [40] for a survey on these toolkits).

2. Separating distributed protocols from the algorithms that implement them within two inheritance hierarchies is a nice principle to achieve flexible and dynamic composition. This principle is worth considering in any distributed system of which components exhibit complex dependencies.

3. The difficulty of implementing reliable distributed protocols using standard remote object interaction applies

[1] GARF is the French acronym for *Génération Automatique d'Applications Résistantes aux Fautes.*

[2] In the Egyptian mythology, *Bast* is a cat-goddess.

not only to CORBA, but also to other middleware such as Java RMI [25] or Microsoft DCOM [7].

4. The *mailer/encapsulator* model we used to separate the concerns of replication from other application aspects can be viewed as a pragmatic reflective model for general object based distributed computing.

5. The conclusions we drew about the underlying costs of group transparency and the major role of marshaling overhead, would apply to many other reliable object based distributed systems.

1.4 Roadmap

The rest of the paper is organised as follows. Section 2 discusses the mismatch between object groups and process groups. Section 3 poses the problem of group protocol composition and sketches the solution we adopted in our implementations. Section 4 discusses the mismatch between the asynchronous semantics of group protocols and synchronous standard CORBA invocations. Section 5 describes the mailer/encapsulator model we used to separate the concerns, and the way we implemented it in Smalltalk and CORBA. Section 6 presents some performance measures of our implementations and dissects the cost of object group transparency. Section 7 concludes the paper through some pragmatic considerations.

2 Object groups vs. process groups

This section recalls the background of our first experience and presents a major problem we faced, namely the mismatch between object groups and process groups. We then describe the way we limited the impact of the mismatch by building an object adaptor on top of Isis and we point out alternative solutions.

2.1 Background

The two best known replication strategies are *active* and *passive* replication [23]. With active replication, all copies of the replicated object handle each request and return a reply. Active replication has the advantage that it does not require any specific treatment if a replica crashes (as long as enough replicas are operational). To ensure replica consistency, a total order multicast protocol is used to ensure that all replicas receive concurrent requests in the same order (Figure 1a). With passive replication, only one replica (generally called the *primary*) executes the request and updates the other replicas (Figure 1b). A reliable protocol is used by the primary to update the state of the other replicas and a group membership protocol is used if a replica crashes or recovers. As shown in [23], both replication strategies have their pros and cons and a system that is intended for programming general replicated applications should support both strategies.

Our first experience consisted in building support for Smalltalk object replication. We decided to provide a default active replication behaviour and to enable the programmer to use passive replication instead when desired. While both replication strategies are intuitive, the distributed protocols (e.g., total order multicast and group

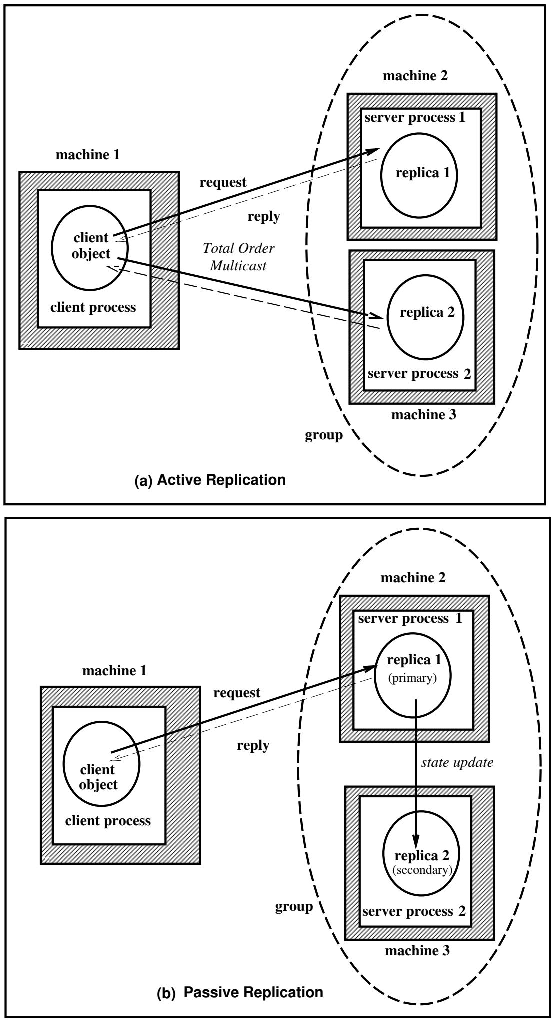

Figure 1: Active and passive replication

membership) required to implement them are quite complex [23]. Instead of building those protocols from scratch, we decided to rely on an existing distributed toolkit, namely Isis, which at the time of our experience (1992), was the only widely known system offering reliable group protocols.

In the following, we point out several effects of the mismatch between the semantics of Smalltalk object groups and Isis process groups. We faced the mismatch when experimenting the use of our first prototype (named GARF [15]) through a replicated distributed diary manager application. Roughly speaking, each member of our group had a (replicated) diary used to store the list of meetings to which the member participates (the application is detailed in [15]).

2.2 The mismatch

Group proliferation

Mapping every object group (i.e., group of object replicas) to a group of processes is a natural way to directly reuse, at the object level, the group mechanisms provided by process group toolkits (Isis in our case). In particular, process group toolkits provide support for creating groups, naming groups, maintaining group membership information and multicasting messages to group members with various delivery guarantees. The *direct* mapping (one object group *corresponds to* one process group [3].) had however two important consequences on our implementation.

1. *Overlapping groups.* Since objects are usually lightweight entities and several objects reside within the same process, we came out with the situation where several Isis process groups were created, even when those groups were actually gathering the same set of processes (the groups overlapped). In our distributed diary manager application for instance, we created a replica of each diary, on every machine of our local network. As a result, several Isis process groups were created: one for each diary. All groups were actually gathering the same set of processes (i.e., the set of Isis processes of our network). This is conveyed by Figure 2 for a scenario with three objects replicated on the same machines: *machine 1* and *machine 2*. Three Isis process groups are created for the same processes, namely *process 1* and *process 2*. Every Isis group of processes introduces specific bandwidth and processing overhead due to the group internal management for group membership and message ordering. In fact, a single group gathering *process 1* and *process 2* would have been sufficient.

2. *Super-groups.* The total order multicast primitive provided by Isis ensures strong server replica consistency in a pure client-server interaction (Figure 1a). Roughly speaking, this primitive guarantees *request atomicity* with respect to concurrency and failures. There are situations however where the client object needs to interact with several servers in an atomic way. In our distributed diary manager application for instance, "scheduling a meeting" requires the interaction with several replicated diaries: a meeting is scheduled if the date is free for *all* involved persons. Hence, scheduling a meeting implies an *atomic* update to several diaries.

[3]An alternative mapping is discussed in Sect.2.3

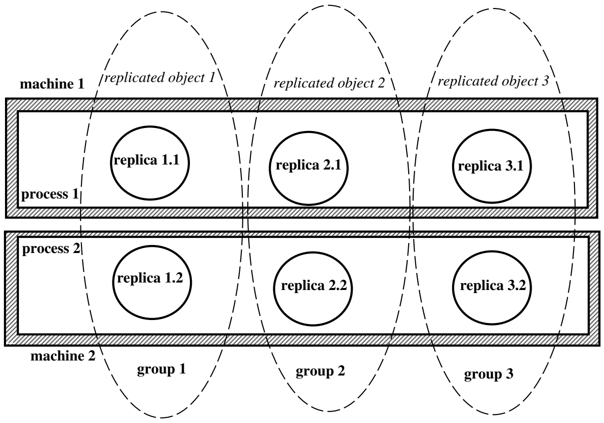

Figure 2: Group proliferation

When every diary is replicated, a meeting implies an atomic activity that involves several replicated objects. The only way to ensure such a *transactional* kind of atomicity is to create, for every *multi-server* request, a *super-group* that contains all replicated diaries [4]. Figure 3 shows a scenario involving two replicated objects. The required semantics is indeed ensured but with a considerable overhead due to the management of big temporary *super-groups*.

Request duplication

When a client invokes an actively replicated server, each replica of the server receives the invocation, performs the requested operation and returns a reply. There is no coordination among the replicas which behave as if they were independent entities. Active replication is straightforwardly supported with a total order multicast primitive when the replicated server plays indeed the strict role of a server. This is not always the case in an object based application as objects may alternatively play the roles of client and server. If a replicated object plays the role of a client for another server, the scenario above will come out with *duplicated* requests. As shown in Figure 4, *request 2* is sent twice to every replica of the second replicated object. This introduces a considerable overhead and can even cause inconsistencies with non-idempotent operations.

Although more subtle, this problem actually occurs also with passive replication. The primary of the first replicated object may issue the request and then crashes and when a new primary is elected, the latter can in turn issue another request.

We had thus to add, on top of Isis, a specific mechanism to *filter* every invocation from a replica in order to discard duplicated requests. The filtering mechanism required a specific distributed synchronisation protocol between the replicas to ensure that exactly one request is issued by the replicated object, even in the case of failures. The mechanism, based on *symmetric proxies*, is detailed and illustrated on the replicated diary manager in [34].

[4]Due to the possibility of deadlocks, a locking based solution does not guarantee continuous availability.

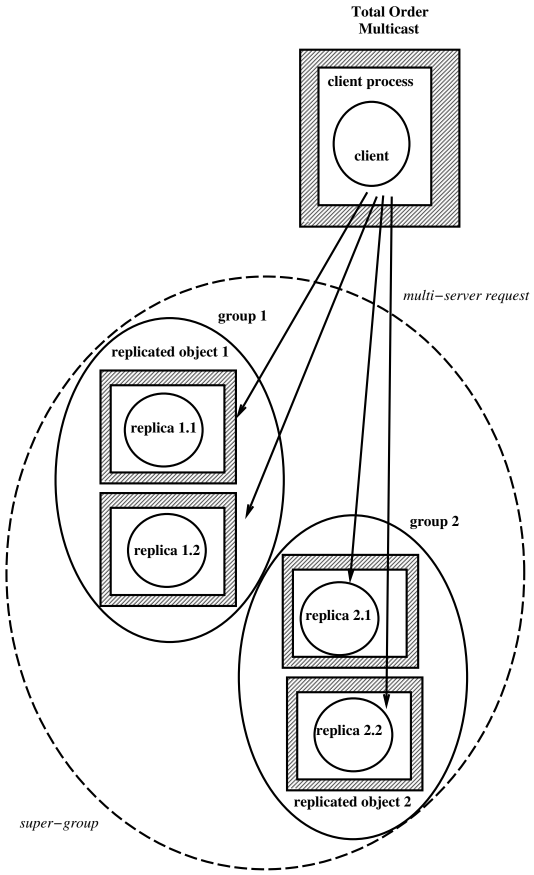

Figure 3: A super-group is created for a multi-server request

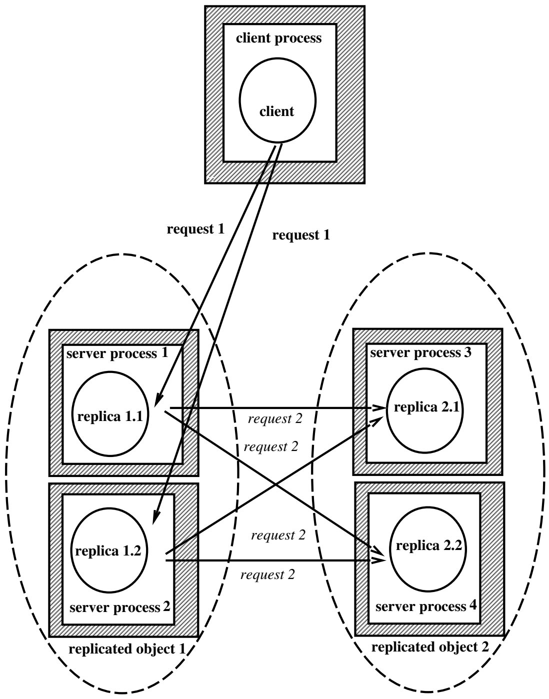

Figure 4: Duplicated requests

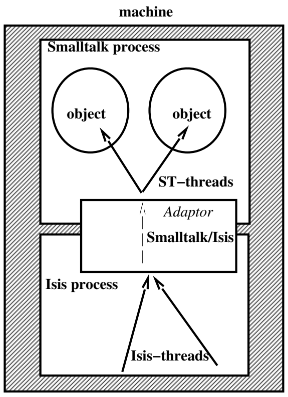

Figure 5: Separating processes

Mixed scheduling

Isis distributed protocols heavily rely on asynchronous communication. This is crucial for distributed protocols such as failure detection and group membership. To support asynchrony, Isis relies on its own multi-threading model. Similarly, Smalltalk heavily depends on its multi-threading facilities (e.g., to run the GUI). The coexistence of the two different threading models had two important influences on our implementation.

1. There was no way of controlling the concurrency between Smalltalk and Isis threads within the same Unix process. We had thus to separate the Smalltalk application from the Isis protocols within two different Unix processes (in every machine). We placed a specific synchronisation adaptor between every Isis process and its associated Smalltalk process (Figure 5).

2. Some of Isis distributed protocols provide ordering guarantees for messages exchanged between the processes. For instance, the Isis total order multicast protocol ensures that all messages sent to a group of processes g, are delivered in the same order by the members of g to the associated Smalltalk processes. This however does not guarantee any ordering between concurrent object invocations inside Smalltalk processes, because of Smalltalk multi-threading. We had thus to explicitly provide a strong serialisation mechanism between the Isis and Smalltalk processes (Figure 5) in order to maintain the message ordering determined by Isis protocols. The serialisation mechanism does not pass a request to the Smalltalk process unless the reply to the previous request has been obtained.

2.3 Alternatives

The mismatch between process groups and object groups is not specifically related to Isis. All process group communication toolkits we know about (e.g., Totem and Horus [40]) were inspired by Isis and would pose the same problems. Both the object adaptor we had to add on top of process groups and the restriction we made about replicated objects (there is no multi-server request atomicity) would have been necessary. At UCSB for instance, Narasimhan, Moser and

Melliar-Smith had to design a similar adaptor, named Eternal [36], on top of their Totem process group communication system to support object replication. The same approach was followed by Maffeis (in the Electra system) to support CORBA object replication with Isis and Horus [33].

Several designers of process group communication toolkits have recognised the mismatch between object groups and process groups. For instance, some experiments with the notion of light-weight group have recently been done in Horus [41]. A light-weight process group is mapped onto an object group, and several light-weight groups are mapped onto one single process group, amortising the cost of many membership changes (Section 2.2.1). This indeed limits the impact of the mismatch (namely overlapping groups) but does not solve all the problems we have faced (e.g., mixed scheduling, request duplication).

Several researchers (e.g., [9]) have argued that group communication toolkits are not adequate to support object replication and that transactional systems provide the ideal support. Indeed, transactions can ensure the consistency of multiple replicas despite concurrency and failures. Nevertheless, existing transactional systems rely on the so-called 2PC protocol [2], which might on one hand block all replicas accessed by a transaction if the transaction coordinator crashes, and on the other hand abort a transaction if a single replica crashes [22]. In the Arjuna project for instance [42], which aimed at supporting object replication using transactions, Little and Shrivastava had to add group communication to their basic transactional kernel [32]. Hence, unlike [9], we do not claim that group communication primitives are useless and that one should better use transactions. We rather claim that, to support object groups, group communication primitives must be build with object semantics in mind. A transactional mechanism should however be integrated within group communication to support multi-server request atomicity.

3 Composing group protocols

This section discusses the issue of composing distributed protocols in an object oriented framework. We had to deal with that issue in a later stage of our first experience, when we decided to build, from scratch, a protocol framework to support Smalltalk object replication (instead of relying on a process group toolkit, i.e., Isis). We named that framework *Bast*. We first summarise below the main characteristics of Bast. Then we discuss the problem we faced when relying on inheritance as the only mechanism to structure distributed protocols. Finally we describe how we by-passed that problem by separating protocols from the algorithms that implement them (i.e., by objectifying the algorithms).

3.1 Background

To support Smalltalk object replication (without the need for an underlying process group system), we built, also in Smalltalk, the Bast object oriented distributed protocol framework. In Bast, we provided group protocols at the object level, together with support for multi-server atomicity, request filtering, and Smalltalk integrated scheduling. Besides circumventing the mismatch between object groups and process groups semantics, we wanted to promote flexible group protocol composition in order to experiment with various replication implementations.

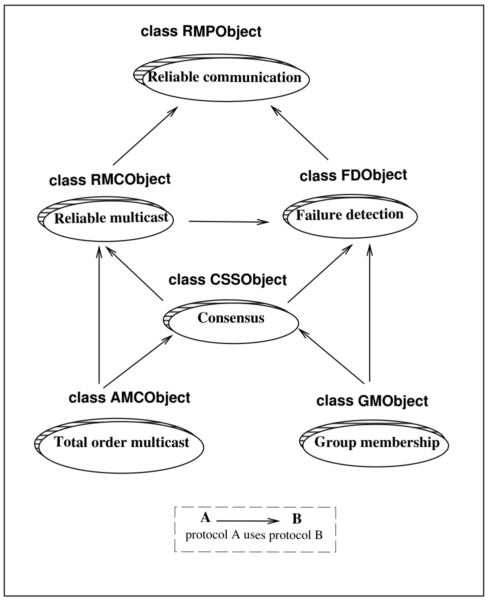

Figure 6: Bast basic protocols

We built Bast as an *open object oriented protocol framework*: Bast protocols are designed as object classes, and structured in a way that a system programmer can customise them by sub-classing existing protocols and implementing call-backs (see [17] for various examples). In contrast to group communication toolkits we know about (e.g., Isis), we did not consider *group membership* to be a basic protocol on which all replicated applications must rely. We rather considered lower level abstractions such as *consensus* and *failure detection* to be first class citizens as well, directly accessible to the programmer. As a consequence, Bast does not impose any reliable distributed programming paradigm. A non-blocking atomic commitment protocol is straightforwardly built using the Bast consensus protocol, and one can mix within the same application, group communication with transactions [21, 17]. Figure 6 presents some of the low level protocols of Bast and their dependencies.

3.2 Structuring protocols with inheritance

Reliable distributed protocols are often challenging to build because of their complex dependency relationships. As shown in Figure 6, the group membership protocol is built on top of consensus and failure detection protocols. Consensus is itself based on failure detection, reliable multicast, and reliable (point-to-point) communication.

In our first implementation of Bast [17], we relied on inheritance as the only mechanism for structuring distributed protocols. Every protocol is represented by a class that inherits from all protocol classes on which it depends. By sub-classing appropriate protocol classes and implementing their call-back operations according to the desired semantics, programmers have the ability to tailor protocols to their needs. However, inheritance alone is not sufficient as far as protocol composition goes, as it does not allow us to easily implement a new algorithm for some existing protocol and to use it in whatever protocol class we want.

To illustrate this problem, consider Figure 6 and suppose some protocol programmer wants to change the failure detection algorithm on which consensus and reliable multicast depend, while leaving it unchanged for group membership. This indeed makes sense as the optimal failure detection mechanism for group membership is not the same as for consensus and reliable multicast. In a group membership protocol, a replica that is suspected is excluded from a group, which is an expensive operation. One should better make sure that the replica is not just slow (and falsely suspected). Hence, even after a long time-out period, before a suspicion is indeed raised, it makes sense to first confirm that suspicion by contacting several replicas (e.g., a majority) and checking the validity of the suspicion. On the contrary, in consensus or reliable multicast, suspecting a replica leads to electing a new leader or forwarding missing messages, which can be desirable to fastly terminate the protocols. In this case, a simple heartbeat protocol (every replica periodically sends an *I am alive* message to all) based on a short time-out period failure detection is sufficient.

With inheritance alone as a code reuse mechanism, one has to implement the new failure detection algorithm both in class `CSSObject` and class `RMCObject`, hampering optimal code reuse.

Assembling the various protocol layers through multiple inheritance can be viewed as an appealing alternative: each class would implement only one protocol, while accessing all required underlying protocols through abstract operations. The latter would then be provided by other protocol classes through multiple inheritance. With this design (sometimes called *mixins*), protocol classes would all be *abstract*. The drawback with this approach is that classes are not more ready-to-use components: before being able to actually create a protocol object, one has to build a new class deriving from all the necessary protocol classes. Furthermore, because protocol layers are assembled through sub-classing, it is very difficult to compose them at runtime.

3.3 Objectifying the algorithms

To achieve flexible protocol composition, we *objectified* distributed algorithms, i.e., we separated protocol layers from their implementations (i.e., the algorithms), themselves manipulated as first-class objects. As we show in [18], objectifying distributed algorithms comes down to recursively apply the *Strategy design pattern* introduced by Gamma et al. [14]. Following the terminology of [14], a protocol is a *context* and an algorithm is a *strategy*. This approach makes protocol composition very flexible and even possible at run time. Protocol dependencies are expressed only at the specification level, through subclass relationships. The algorithms are structured in a separate inheritance hierarchy. This can lead to choose/build the algorithm that is best adapted to every protocol. Back to the failure detection example above, one can have a majority based failure detector algorithm with for the group membership protocol, and a simpler failure detection algorithm (with a short time-out value) for the consensus and reliable multicast protocols.

Although applied in a different context, this approach is similar to the one followed in the design of the *CONDUIT+* framework of network protocols [26], and the *x-Kernel* library of communication protocols [39].

4 Asynchronous protocols vs. synchronous invocations

Our Bast framework enabled us to experiment with various group protocols and algorithms. The main limitation of Bast however is its restriction to single-language object groups, namely Smalltalk. We decided to reuse the experience gained from Smalltalk object replication and build support for general CORBA object replication. In the following we recall the context of our CORBA experience, then we discuss the difficulty we faced in building (reliable) group communication protocols using standard CORBA communications, namely the mismatch between the asynchrony of group protocols and the synchrony of standard remote CORBA object invocations. We describe the way we dealt with that mismatch and we discuss alternative approaches.

4.1 Background

The *Common Object Request Broker Architecture* (CORBA), proposed by the *Object Management Group* (OMG) [37], is a middleware specification that defines the basic mechanisms for remote object invocation through an *Object Request Broker* (ORB), as well as a set of services for object management, e.g., *Persistence Service*, *Event Service*, and *Transaction Service* [38]. The basic ORB specification does not contain any aspect related to fault-tolerance and the only CORBA service which offers some degree of reliability is the *Transaction Service*. However, as pointed out in Section 2, the protocols underlying that service preserve consistent long-term data by using recovery mechanisms upon failures, but do not ensure continuous availability [5].

Our objective consisted in building support for CORBA object groups following the OMG guidelines, i.e., we added group support as a new CORBA service, besides existing CORBA services and without requiring any modification to the ORB specification. This approach follows the design of the other functionalities that have been added to CORBA through services, such as persistence and transactions. These services were specified in CORBA IDL and were adopted as CORBA standards. With this objective in mind and after our first experience with Smalltalk object replication, we designed a set of IDL interfaces to describe our Object Group Service (OGS), itself designed as a set of sub-services (Figure 7). We implemented it in C++ using two commercial off-the-shelf ORBs that comply with the CORBA 2.1 specification: *Orbix* and *Visibroker*.

Although we strictly followed the OMG guidelines in service design and implementation, it was not possible to obtain a fully portable code for OGS (from Orbix to Visibroker). The major reason was that distributed group protocols, such as failure detection and group membership, are inherently asynchronous and, as we discuss below, there is no standardised CORBA support for (reliable) asynchronous communication.

4.2 The mismatch

In the CORBA specification, remote object invocations are by default synchronous. A client invokes operations on a lo-

[5] OMG explicitly requires the use of the 2PC protocol, which is known to be non-fault-tolerant [2]. In fact, the OMG is currently issuing a *Request For Proposal* concerning fault-tolerant CORBA. We will come back to that in the last section of this paper.

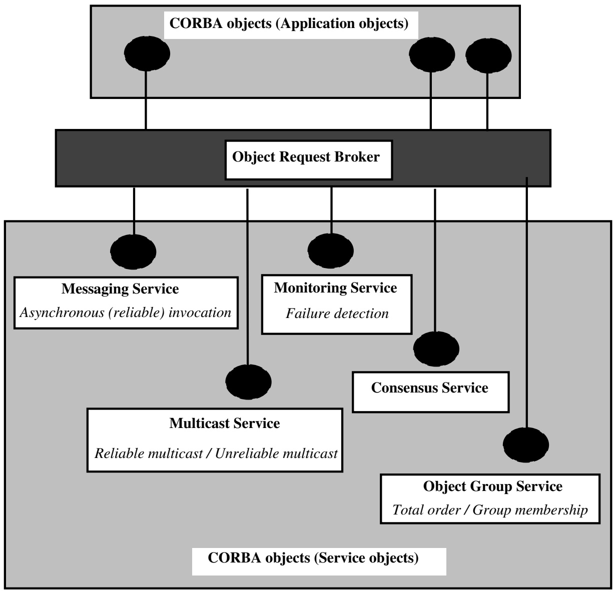

Figure 7: OGS and its components as CORBA services

cal stub, which marshals arguments, sends requests over the network, awaits a response, and returns it to the client. The client application code is blocked on the request until completion. This communication mechanism is sometimes called *Object Remote Procedure Call (ORPC)*. It is inspired by the well known RPC paradigm [3], and is also the basic communication paradigm of distributed middleware like DCOM [7] or Java RMI [25]. ORPC extends distributed computing in a straightforward manner from non-distributed object invocation and hence greatly eases distributed application development. However, there are many situations in which ORPC is not sufficient, including the programming of reliable distributed protocols such as failure detection and consensus, at the heart of our protocol framework.

CORBA specification allows to declare operations using the `oneway` IDL keyword. This keyword identifies an operation as flowing exclusively in one direction, i.e., the operation does not return any value. The CORBA specification also provides a *send now - receive latter* invocation style, called *deferred synchronous* invocations, through two basic operations: `send` and `get_response`.

The problem with these types of invocations is that their semantics are (intentionally) too vague in the specification. More precisely, the only guarantee provided is that the requested operation is performed at most once and the requester never synchronises with the completion, if any, of the request [37]. This, in fact, does not even prevent a *compliant* ORB to discard the invocation message without sending it. Furthermore, the specification does not enforce request-only operation to be non-blocking. Nothing indeed states that the client will not wait for the server to acknowledge the reception of the *message* that transports the remote invocation; thus, a remote call can block *forever at the transport level*, which can block in turn the application. This situation can typically occur in case of a link failure, if the network is congested, or if the server is extremely busy. For instance, an Internet Inter-ORB Protocol (IIOP) invocation can block the entire process on a `oneway` call just because a TCP/IP

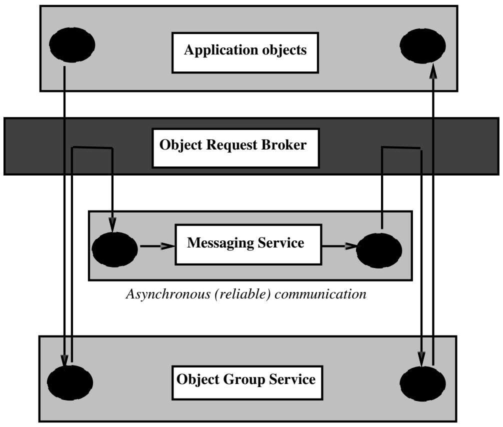

Figure 8: The Messaging Service

buffer fills up. Although the ORB at the client side could detect a possible deadlock when performing an IIOP call, this behaviour is not guaranteed by the CORBA standard.

As a consequence, in order to avoid having OGS rely on unreliable communication mechanisms, we decided to base our implementations on neither of the above mentioned mechanisms and we used multi-threading instead.

4.3 Multi-threading

A natural way to provide asynchronous communication on top of synchronous invocation is to use multi-threading, e.g., by starting threads for outgoing requests. However, in version 2.1 of the CORBA specification, thread support and management is not specified and thus is not portable. Although both Orbix and VisiBroker support multi-threading, they provide very different programming models. Both provide object oriented wrappers for threads, locks and condition variables that ensure platform independence (e.g., between Posix and Windows NT threads), but these wrapper classes are not compatible.

As a result, our OGS resulting code was not fully portable and we came out with two slightly different implementations: one for Orbix and one for Visibroker. Fortunately, the non-portable part of the code was confined within our messaging service: one of the services underlying OGS (Figure 7). This service relies on multi-threading to provide basic mechanisms for managing reliable asynchronous messages.

An *Object Messaging Service* will be soon standardised by the OMG, providing various qualities of service, including reliable asynchronous message passing. Although it might take some time before commercial ORBs support this service, OGS could be plugged on that service by simply replacing its messaging service, with minimal impact on other OGS services (Figure 8).

4.4 Alternatives

CORBA Event Service

Besides the basic CORBA communication mechanism, OMG specifies a *publish/subscribe* communication paradigm in the

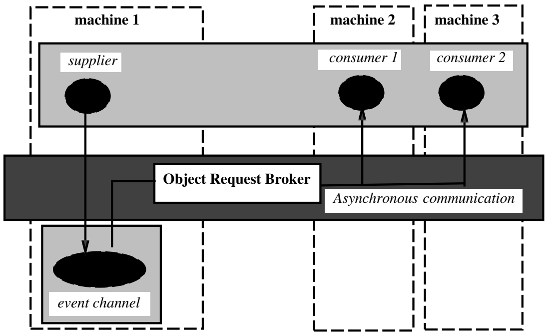

Figure 9: An event channel is a single point of failure

CORBA Event Service. This service decouples the communication between *suppliers* and *consumers* through *event channels.* Suppliers produce event data and consumers process event data. An event channel is an intervening CORBA object that allows suppliers to communicate with consumers *asynchronously* (Figure 9).

The major problem is that the architectural design of the Event Service is *centralised*: although consumers and suppliers use different interfaces for pushing/pulling event data to/from the channel, they have to invoke the same centralised event channel object in order to connect to an event channel. As shown by Figure 9, the event channel is a CORBA object located at a given machine and is thus a single point of failure. One might circumvent this problem by following either of the three following approaches:

- *By replicating the event channel.* This would lead to a recursive architecture as the aim of OGS is indeed to support replication.
- *By representing an event channel as an IP multicast address (and not as a CORBA object).* A supplier generates data by sending it to the multicast address. This approach was adopted by IONA in an Event Service implementation, called OrbixTalk [27]. However, 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 have a unique (centralised) access entry to be designated by an interoperable object reference.
- *By chaining event channels.* Several event channels could be located on the client and on the server site. This model provides two-way communication with no single point of failure. Distinct clients generate data using distinct request-suppliers and receive replies through distinct response-consumers. Although complex, this approach is fully CORBA compliant, does not modify the *Event Service* specification and does not introduce a single point of failure [12].

Our modular structuring of OGS would a priori make it easy to change the implementation of our messaging service and replace it with chained event channels or IP multicast (instead of multi-threading) without impact on the rest of OGS implementation.

Proprietary communication mechanisms

An alternative approach to the use of CORBA standard communication mechanisms and services would have been the use of *external* asynchronous communication primitives. This approach has been followed in Orbix+Isis [29], Electra [33] and Eternal [36]. Besides the fact that these systems rely on process group communication toolkits (Isis, Horus and Totem, respectively), with the inherent limitations that we pointed out in Section 2, they are proprietary and do not comply with the OMG architecture.

In Orbix+Isis [29] and Electra [33], new extensions to IDL and ORB architecture are required. In fact, a proposal to the OMG has been made for those extensions by Isis Distributed Systems, Inc [30], but the proposal was not adopted. Accepting that proposal would have meant that all current ORBs should have to be rewritten to include support for group communication. In Eternal [36], the ORB is not aware of groups. Internet Inter-ORB Protocol (IIOP) requests are intercepted transparently on client and server sides using low-level interception mechanisms and passed to a group communication toolkit (Totem) that forwards them using group multicasts. The interception approach does not require any modification to the ORB, but it relies on low-level mechanisms specific to Unix platforms and it is not clear how invocations to replicated and non-replicated objects are distinguished.

5 Separating the concerns

One of the major objectives of our experiences was to achieve clean separation of concerns, i.e., separation of group communication features from application functional code. The aim was to hide groups for a programmer that is not experimented in reliable distributed programming, but to provide enough flexibility for experimented programmers to customise the group protocols according to their application needs, without however touching the functional code of the application. We describe below how we achieved this objective using a simple two-level reflective model, and how we implemented it both in Smalltalk and C++/CORBA.

5.1 Background

Ensuring separation of concerns in our context means that application objects do not deal with grouping issues. They virtually communicate in a *point-to-point, synchronous, request/reply* manner. Aspects such as multicast invocations are plugged *transparently* underneath those invocations. To provide that capability, we have introduced two kinds of *"meta-objects"* [31]: *encapsulators* and *mailers.* The *encapsulator* plays the role of a group member *administrator* whereas the *mailer* plays the role of a group *accessor (or proxy).*

- *Encapsulators* are used to *wrap (encapsulate)* replicas by controlling the way they treat incoming and outgoing requests. An encapsulator contains the replication code that needs to be executed at the node of the replica, before and/or after the replica executes any of its operations. Encapsulators are located at the nodes of the replicas (in fact a replica and its associated encapsulator execute within the same process). Every replicated object class is mapped to a class of

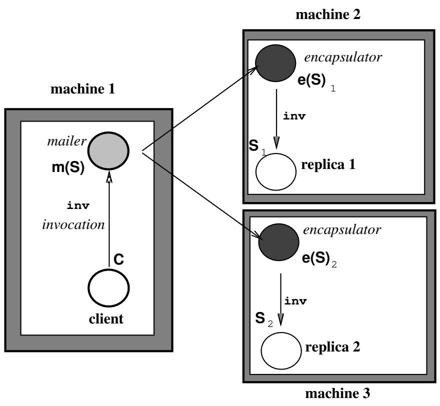

Figure 10: Mailer and encapsulators

encapsulators. The default encapsulator class implements the *active replication* strategy: it ensures that all replicas treat an invocation and return a reply. As long as one replica is alive, a reply is returned. No specific treatment is needed if a replica crashes (the default strategy assumes perfect failure detection, e.g., no network partitions). One can however map a replicated objects to an encapsulator class that implements a different strategy. For instance, one can uses the *passive replication* strategy (primary-backup), which consumes less resources as only one replica, the primary, performs the requests, but requires specific treatment in the case of the crash of the primary, i.e., a new primary must be elected.

- *Mailers* act as *smart* proxies of replicated objects. They are *smart* in the sense that they do not only forward communication, but they also transform point-to-point invocations into multicast invocations to sets of replicas. A mailer of a replicated object is created at every node where a reference of that object exists (in fact they execute on the same process). Every replicated object is mapped to a class of mailers. The default mailer transforms point-to-point invocations into totally ordered multicasts, and waits for the first reply. One can however map a replicated object to a mailer that provides a weaker semantic (e.g., reliable but not totally ordered multicast) or does not wait for any reply.

Figure 10 depicts a simple scenario where a client `C` interacts with a replicated object `S`. Object `S` is bound to an encapsulator `e(S)` of the default class `ActiveReplica` (active replication), and to a mailer `m(s)` of the default class `TOcast` (total order multicast). The replicas of `S` are `S1` and `S2`, respectively attached to the encapsulators `e(S)1` and `e(S)2`. The mailer `m(S)` is located on the node of `C` and acts as a proxy of `S`: `m(S)` transforms a simple invocation to `S` into a remote multicast to `S1` and `S2`. The mailer selects one reply among all (the first one to arrive), and forwards it to the client.

5.2 A pragmatic reflective model

Our mailer/encapsulator model can be viewed as a pragmatic reflective model as it has only one level of reflection, i.e., there is no *meta-meta-level*. The mailer/encapsulator model is flexible enough and promotes an incremental programming methodology. First, the programmer describes the functional aspects of the application without considering distribution and replication issues. At a later stage, the programmer turns to replication features by *binding* application objects to adequate encapsulator and mailer classes. The *binding* step can actually be performed at run-time, and one can change a replication policy in a dynamic way.

The mailer/encapsulator model is more flexible than the *Gaggle* model introduced in [6]: a *gaggle* can be viewed as a specific abstraction hiding replication from clients. In comparison, our model goes a step further in separating the concerns by decoupling the client aspect of the replication code (i.e., multicast) from the server aspect of the replication code (i.e., synchronisation). The first is confined within *mailers* whereas the second is confined within *encapsulators*.

5.3 On mailers and encapsulators

In Smalltalk

In our Smalltalk [20] developments, the mailer/encapsulator model was implemented in a straightforward way using the reflective capabilities of the language. The association between application objects, mailers, and encapsulators, is handled during object creation and communication. We simply had to replace the names of replicated object classes in the Smalltalk dictionary, in order to catch replicated object creation through the `doesNotUnderstand` exception mechanism. Instead of returning the actual object, we added some specific code in order to create the replicas, perform the binding to encapsulator and mailer classes, and return a *proxy* instead of the actual object. In a similar way, every invocation is intercepted through the `doesNotUnderstand` exception mechanism, and redirected through mailers and encapsulators (the Smalltalk implementation of mailers and encapsulators is fully detailed in [16]).

In CORBA

Although CORBA does not provide the same level of dynamicity as Smalltalk, it offers some mechanisms for message interception and redirection. In particular, CORBA 2.1 specifies a *Dynamic Skeleton Interface (DSI)* and a *Dynamic Invocation Interface (DII)*. The DSI locally accepts requests that are actually aimed at the remote server interface, and was used to redirect message through mailers. The DII constructs the invocations for the server interface, and is used to redirect messages through encapsulators (Figure 11).

An easy way to implement mailers and encapsulators in our CORBA experience would have been to use ORB specific mechanisms for message interception and redirection, such as *smart proxies* and *filters* in Orbix. Although most of commercial ORBs we know about provide similar mechanisms, these are not CORBA compliant features and would have heavily impacted the portability of our code.

The new CORBA specification introduces the notion of `Interceptor` object. An interceptor is an object interposed in the request and response paths between a client and server. After this notion is introduced into commercial ORBs,

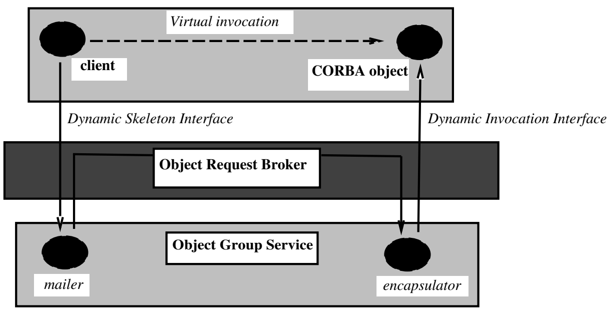

Figure 11: Dynamic Invocation/Skeleton Interfaces

it will be a reasonable alternative to dynamic invocation and skeleton interfaces for implementing mailers and encapsulators.

6 The cost of group transparency

We dissect in this section the global overhead of full group transparency, both in Bast and OGS. After describing the configuration of our network, we present the global overhead of group transparency, i.e., we compare the cost of transparently invoking a group of replicas as if it was a single (fault-tolerant) object, with the cost of a single remote object invocation. Then we detail the various costs underlying group transparency.

We do not present any measure on GARF/Isis because, at the time when this testing took place, Isis was no longer supported (by Stratus) and we did not own an Isis licence anymore. Some older measures of GARF/Isis are given in [16]. Even for Bast and OGS, we do not present exhaustive performance figures: these can be found in [13, 16, 19].

6.1 System configuration

The performance figures we describe below are obtained from Bast in Smalltalk using VisualWorks 2.x, and OGS in C++ using multi-threaded VisiBroker for C++ 3.0.

Testing took place on a 10Mbit Ethernet interconnecting 13 Sun SPARCstations 20 and UltraSPARC1 running Solaris 2.5.1 and 2.6. Each workstation was running Xwindows as well as several interactive applications (e.g., netscape and emacs): network and CPU loads were medium to high. We considered an object replicated on three Sun UltraSPARC1: each workstation was hosting one replica, and one client application was located on the same node as one of the replicas. When receiving the client invocation, every replica immediately sends back a reply, i.e., it does not perform any specific treatment. All tests consist in invoking an operation that takes a single parameter, which is a value of a complex type (a sequence of references). In the case of OGS, the parameter is an `inout` variable.

Since our objective is to compare the cost of a transparent group invocation with that of a standard remote invocations (i.e., with the invocation of a non-replicated object), we measured only failure-free executions.

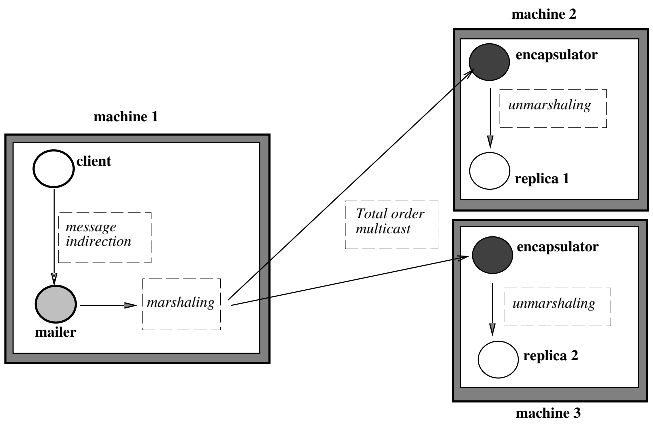

Figure 12: Dissecting the cost of group transparency

6.2 The overhead of group transparency

When measuring the cost of group transparency, we considered the case where a client issues a standard invocation that is intercepted and then transformed into a multicast invocation to all replicas. The client is not aware of the server being replicated. The invocation is performed within a total order multicast protocol which guarantees strong replica consistency. The invocation is terminated when the client receives back the first reply. We compare this cost with that of an invocation of the same, yet non-replicated, object. For Bast, we considered a reliable remote invocation, whereas for OGS, we considered a remote invocation over IIOP (TCP/IP), which is usually considered reliable.

System	BAST	OGS
Single remote invocation	8	160.923
Transparent group invocation	0.7	17.9

Table 1: The overhead of group transparency - 3 replicas - (invocations/sec)

Performance figures are presented in Table 1. Results are expressed in number of invocations per second (throughput). Interestingly, in both cases, group transparency has an overhead factor of around 10.

6.3 Dissecting the overhead

As shown by Figure 12, there are various costs underlying group transparency: (1) message indirection (i.e., invocation transparency), (2) marshaling/unmarshaling (i.e., argument transparency), and (3) total order multicast (i.e., behaviour transparency). For OGS, there is an extra cost, related to the use of an OGS daemon process to achieve language transparency (this cost is however small when compared to marshaling). In the following, we consider each of those costs individually.

Invocation transparency

In our context, invocation transparency means the ability to reuse an application code written without replication in mind and plug group protocols underneath object invocations. This is basically the cost of message indirection.

In Bast, invocation transparency is achieved through the Smalltalk exception `doesNotUnderstand`. When bypassing the mailer/encapsulator indirections, the response time of a local invocation was around 10μs. When intercepted and redirected through mailers and encapsulators, the same invocation took about 500μs. The reason for the overhead is that during normal operation execution, the Smalltalk virtual node is partially bypassed, whereas when the exception is raised, the virtual node *reinterprets* the method that caused the exception. This is what happens when the method `doesNotUnderstand` is called. Other slowdown factors are the manipulation of stack frames as Smalltalk objects and the cloning of mailers.

In OGS, transparency is achieved through the *Dynamic Skeleton Interface (DSI)* and the *Dynamic Invocation Interface (DII)*. Table 2 presents the invocation throughput for both transparent and non-transparent invocations in OGS. In the latter case (non-transparent), invocations are performed with stubs and skeletons that are generated at compile time.

Protocol	Transparent	Non-transparent
Broadcast	36.4	81.4
Reliable multicast	32.5	55.3
Total order multicast	17.9	22.6

Table 2: The overhead of invocation transparency in OGS (invocations/sec)

Not surprisingly, transparent invocations are less efficient that non-transparent ones. The difference gets however smaller with complex protocols such as total order. This is due to the fact that the DSI and the DII are used only once per invocation, and add a fixed cost to the invocation time. If the protocol is complex, more messages are generated, without increasing the fixed cost of dynamic request processing.

Argument transparency

The protocol frameworks underlying Bast and OGS enable us to perform group invocation, whatever the types of the invocation arguments are. In Bast, parameters can be of any subclass of `Object`, whereas in OGS parameters can be of any subtype of the generic CORBA type `Any`. Table 3 presents an approximation of the time percentage spent in marshaling and unmarshaling. We consider only the case where invocations are performed with a total order multicast: this is the case where the maximum number of remote messages are exchanged and where marshaling and unmarshaling consume the highest percentage (see [13, 19] for more details on alternative multicast protocols).

Implementation	**Marshaling**
BAST	65%
OGS	45%

Table 3: The overhead of marshaling (percentage)

These results clearly convey the fact that marshaling and unmarshaling are important causes of overhead. BAST implementation is based on VisualWorks Binary Object Storage Service (BOSS), which is not optimised for remote communication. In the case of OGS, the cost associated to marshaling and unmarshaling is due to the management of `Any` values. Unlike other IDL types, `Any` values are augmented with type information. Constructing this information and checking its validity increase message size and slow down the remote invocation process. For instance, extracting a complex structure from an `Any` value requires a time similar to that of performing a remote invocation (in a LAN).

Behaviour transparency

We compare below the costs of various multicast protocols. The first protocol is a simple multicast that does not guarantee any consistency as the replicas may receive the invocations in different orders and, even worse, some replicas might receive an invocation while other might not. This protocol can however be considered sufficient for a *read-only* invocation. The second protocol is a reliable multicast which ensures that either all replicas receive the invocations or none of them does. This protocol does not ensure total order, but is enough to guarantee consistency if the operations are commutative. Finally, the third protocol is a total order protocol which always guarantees strong replica consistency (even with non-commutative operations). It ensures *behavioural transparency*: all replicas behave the same, and the replicated object looks like a non-replicated one. The results conveyed by Table 4 clearly confirm the fact that strong consistency (i.e., behavioural transparency) introduces a considerable overhead.

Implementation	**BAST**	**OGS**
Multicast	7.5	100.1
Reliable multicast	2.2	62.4
Total order multicast	0.7	25.7

Table 4: The overhead of strong consistency - 3 replicas - (invocations/sec)

The algorithms we have used for reliable and total order multicast are similar to those described in [8]. In particular, the total order multicast protocol ensures that, whatever the number of crashed nodes, and even if nodes are falsely suspected to have crashed, total order is never violated. One could consider a weaker, yet more efficient, algorithm that assumes for instance perfect failure detection and guarantees total order agreement in a probabilistic manner. Thanks to Bast and OGS open structures, changing the algorithm will not require any modification to the application.

Language transparency

The OGS implementation considered so far uses a specific daemon process on every node to run the OGS code at the client side. This approach has the advantage of achieving *language transparency*, e.g., a Java client can use the C++ version of OGS. Table 5 compares the cost of the *daemon* approach with that of a *library* approach where OGS is simply linked with the client. In the second case, the client needs to be written in the same language as OGS (namely C++).

Protocol	Daemon	Library
Broadcast	36.47	81.4
Reliable multicast	32.5	55.3
Total order multicast	17.9	22.6

Table 5: The overhead of language transparency (invocations/sec)

Not surprisingly, the daemon solution introduces an overhead of an additional inter-process communication, i.e., the cost of the indirection through the daemon process. As for invocation transparency, the overhead gets smaller with complex protocols such as total order multicast.

7 Summary

For a couple of decades now, many projects have been devoted to the design and implementation of object based concurrent and distributed systems, mainly focusing on building support for object based concurrency control, remote object invocation and object migration [4]. Very few however have been devoted to building support for replication and object groups. This is actually not surprising as most object based distributed systems have been experimented on academic applications for which fault-tolerance is rarely a mandatory feature.

This situation has recently changed when distributed objects became the cornerstone of industrial standards such as CORBA and DCOM. Applications in various areas are being developed or ported on object based distributed systems, including those with strong requirements for continuous availability such as air traffic control, telecommunication, medical and financial systems. In fact, the OMG has issued a *Request For Proposal (RFP)* to enhance CORBA with fault-tolerance through object group support, and our Object Group Service complies with many of the requirements of that RFP.

From a more general point of view, we believe that the observations we summarised in this paper can be of valuable help to the designers and implementors of future reliable object based distributed systems.

We claim for instance that the protocols underlying object groups should be developed themselves within an object oriented protocol framework. This provides nice flexibility and modularity features and avoids the mismatch between object group and process group semantics. In the context of CORBA, our claim means that the best way to introduce group support is to follow a *service approach* where group communication protocols are themselves CORBA objects exporting IDL interfaces. We pointed out the fact that group protocols cannot be easily implemented using standard CORBA synchronous object remote invocations. One should better rely on a separate asynchronous invocation mechanism, which could be replaced by a standard one as soon as the CORBA *Object Messaging Service* announced by the OMG, is supported by commercial ORBs.

We also argued that a *simple* reflective mailer/encapsulator model is sufficient to separate group primitives from other application aspects and there is no need for complex multi-level reflective models. The mailer/encapsulator model can easily be implemented in a dynamic environment like Smalltalk and with a reasonable effort in middleware like CORBA or DCOM, using their dynamic invocation facilities.

We also argue that group transparency should be provided *à la carte*. It is indeed important to hide groups from non-experimented programmers and promote code reuse by separating application functional code from specific calls to group primitives. Nevertheless, transparency comes in different flavours and an experimented programmer should be able to *switch-off* any of those flavours and trade it with better performances. For example, we have experimented with a Bast scenario where distributed protocol messages with *ack* and *nack* values were not marshaled (i.e., we switched-off argument transparency): the throughput of group invocation using a total order multicast protocol was doubled. The *transparency à la carte* principle has indeed been recognised as important for other distribution aspects, but we believe that this principle is crucial for object group support because of the cost and complexity of the underlying protocols.

As already pointed out, the aim of this paper was to draw general observations and not describe specific aspects of our systems. More information on GARF, Bast and OGS, are available, respectively, at:

- *http://lsewww.epfl.ch/Research/Garf.*
- *http://lsewww.epfl.ch/Bast.* The Smalltalk implementation of Bast is public free and available there. A Java version of Bast will also be available at that address soon.
- *http://lsewww.epfl.ch/OGS.* C++ implementations of OGS, for Orbix and Visibroker, are public free and available there.

Acknowledgements

We are very grateful to Ron Resnick for his comments on an earlier draft of this paper.

References

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

[2] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison Wesley, 1987.

[3] A. Birell and B. Nelson. *Implementing Remote Procedure Calls.* ACM Transactions on Computer Systems, 2 (1), 1984, 39-59.

[4] J.P. Briot, R. Guerraoui and K.P Lohr. *Concurrency and Distribution in Object-Oriented Programming.* ACM Computing Surveys, September 1998.

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

[6] A. Black, and M. Immell. *Encapsulating Plurality.* European Conference on Object-Oriented Programming, Springer Verlag (LNCS 707), 1993, 57-79.

[7] N. Brown and C. Kindel. *Distributed Component Object Model Protocol.* DCOM, http://www.microsoft.com/oledev/olecom.

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

[9] D. Cheriton and D. Skeen, *Understanding the Limitations of Causally and Totally Ordered Communication.* ACM Symposium on Operating Systems Principles, 1993.

[10] E. Elnozahy, V. Ratan, and M. Segal. *Experiences Using DCE and CORBA to Build Tools for Creating Highly-Available Distributed Systems.* IEEE Symposium on Fault-Tolerant Computing Systems, 1996.

[11] P. Felber, B. Garbinato, and R. Guerraoui. *The Design of a CORBA Group Communication Service.* IEEE Symposium on Reliable Distributed Systems, 1996, 150-159.

[12] P. Felber, R. Guerraoui, and A. Schiper. *Replicating Objects with CORBA Event Channels.* IEEE Workshop on Future Trends of Distributed Computing Systems, 1997, 14-21.

[13] P. Felber, R. Guerraoui, and A. Schiper. *The Implementation of a CORBA Object Group Service.* Theory and Practice of Object Systems, John Wiley and Sons, Inc. Special issue on High Availability in CORBA, 1998, 4 (2), 93-106.

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[15] B. Garbinato, R. Guerraoui, and K. Mazouni. *Distributed Programming in GARF.* In Object-Based Distributed Programming, Springer Verlag (LNCS 791), 1993, 225-240.

[16] B. Garbinato, R. Guerraoui, and K. Mazouni. *Implementation of the GARF Replicated Objects Plateform.* Distributed Systems Engineering Journal, 1 (2), 1995, 14-27.

[17] B. Garbinato, P. Felber, and R. Guerraoui. *Protocol Classes for Designing Reliable Distributed Environments.* European Conference on Object-Oriented Programming, Springer Verlag (LNCS 1098), 1996, 316-343.

[18] B. Garbinato and R. Guerraoui. *Using the strategy design pattern to compose reliable distributed protocols.* Usenix Conference on Object-Oriented Technologies and Systems, 1997, 221-232.

[19] B. Garbinato and R. Guerraoui. *Flexible Protocol Composition in BAST.* IEEE International Conference on Distributed Computing Systems, 1998, 22-27.

[20] A.J Goldberg and A.D Robson. *SMALLTALK-80: The Language and its Implementation.* Addison Wesley, 1983.

[21] R. Guerraoui and A. Schiper. *Transactional model vs Virtual Synchrony model: bridging the gap.* In Theory and Practice in Distributed Systems, Springer Verlag (LNCS 938), 1995, 121-132.

[22] R. Guerraoui, R. Oliveira, and A. Schiper. *Atomic updates of replicated data.* European Dependable Computing Conference, Springer Verlag (LNCS 1150), 1996, 365-382.

[23] R. Guerraoui and A. Schiper. *Software-Based Replication for Fault-Tolerance.* IEEE Computer, 30 (4), 1997, 68-74.

[24] R. Guerraoui, B. Garbinato and K. Mazouni. *GARF: A Tool for Programming Reliable Distributed Applications.* IEEE Concurrency, 5 (4), 1997, 32-39.

[25] E. Harold. *Java Network Programming.* O'Reilly, 1997.

[26] H. Huni, R. Johnson, and R. Engel. *A Framework for network protocol software.* ACM Conference on Object-Oriented Programming Systems, Languages and Applications, 1995.

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

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

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

[30] Isis. *Object Groups: A response to the ORB 2.0 RFI.* Isis Distributed Systems, Inc, 1993.

[31] G. Kiczales, J. des Rivières, and D. Bobrow. *The Art of the Metaobject protocol.* The MIT Press, 1991.

[32] M. Little and S. Shrivastava. *Object Replication in Arjuna.* Broadcast Project deliverable report, Vol. 2, 1994 (available from Dept of Computing Science, University of Newcastle upon Tyne, UK).

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

[34] K. Mazouni, B. Garbinato, and R. Guerraoui. *Filtering Duplicated Invocations Using Symmetric Proxies.* IEEE International Workshop on Object Orientation in Operating Systems, 1995, 118-126.

[35] S. Mishra, L. Peterson, and R. Schlichting. *Implementing Fault-Tolerant Replicated Objects Using Psync.* IEEE Symposium on Reliable Distributed Systems, 1989.

[36] P. Narasimhan, L. Moser, and M. Melliar-Smith. *Exploiting the internet inter-ORB protocol to provide CORBA with fault-tolerance.* Usenix Conference on Object-Oriented Technologies and Systems, 1997, 81-90.

[37] OMG. *The Common Object Request Broker Architecture: Architecture and Specification.* OMG.

[38] OMG. *CORBAservices: Common Object Services Specifications.* OMG.

[39] L. Peterson, N. Hutchinson, S. O'Malley, and M. Abott. *Rpc in the x-Kernel: evaluating new design techniques.* ACM Symposium on Operating Systems Principles, 1989, 91-101.

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

[41] L. Rodrigues, K. Guo, A. Sargento, R. van Renesse, B. Glade, P. Verissimo, and K. Birman. *A Transparent Light-Weight Group Service.* IEEE Symposium on Reliable Distributed Systems, 1996, 130-139.

[42] S. Shrivastava, G. Dixon and G. Parrington. *An Overview of Arjuna: A Programming System for Reliable Distributed Computing.* IEEE Software, 8 (1), 1991, 63-73.

[43] Visigenic. *Visibroker C++ 3.0 Programmer's Guide.* Visigenic Software, Inc., 1997.

[44] M. Wood. *Replicated RPC Using Amoeba Closed Group Communication.* IEEE International Conference on Distributed Computing Systems, 1993.