Structuring distributed applications as fragmented objects

Structuration d'applications réparties en objets fragmentés

Rapport de Recherche INRIA 1404

Mesaac Makpangou

Yvon Gourhant

Jean-Pierre Le Narzul

Marc Shapiro

INRIA, projet SOR

January 1991

Abstract

Most distributed systems offer only primitive communication objects (e.g. channels). Their undisciplined use obscures, and exposes the implementation of, higher-level concepts.

We propose instead a high-level, structured approach, called Fragmented Objects. A Fragmented Object is a distributed shared object. Clients see a fragmented object as an ordinary object. The implementor of a fragmented object decides of its interface and representation, including (if necessary) aspects such as placement of data items, communication between fragments, protocol layering, and binding. The fragmented object model provides a common framework for different distribution mechanisms, such as client/server stubs, replication, cacheing, and data partitioning.

We present the basic fragmented object concepts and a full example, the SOS Naming Service, layered into Name Space and Naming View objects. Each of these layers is implemented as fragmented objects; we point out specific benefits of the fragmented object approach.

We have defined a fragmented-object language called FOG, a compiler, and a toolkit of primitive fragmented objects. The compiler enforces encapsulation, checks interface consistency, and generates hooks to the toolkit. The compiler and the toolkit facilitate the most common distribution policies.

Résumé

La plupart des systèmes répartis offrent des objets de communication de bas niveau (p.ex. des canaux). Leur utilisation indisciplinée obscurcit la réalisation de concepts de plus haut niveau et expose leur implémentation.

Nous proposons une approche structurée de haut niveau : les objets fragmentés. Un objet fragmenté est un objet partagé réparti; il apparaît à ses clients comme un simple objet. Le programmeur d'un objet fragmenté décide de son interface et de sa représentation, y compris, s'il le désire, du placement des données, de la communication entre les fragments, de la décomposition des protocoles et du mécanisme de liaison.

Le modèle d'objet fragmenté fournit un cadre commun à différents mécanismes de répartition tels que les talons client/serveur, la réplication, l'utilisation de caches et le partitionnement des données.

Nous présentons les concepts de base des objets fragmentés, ainsi qu'un exemple d'application, le service de nommage du système SOS. Celui-ci est structuré en deux niveaux: les objets "espace de noms" et les objets "vue de nommage". Chacun de ces niveaux est réalisé par des objets fragmentés. Quelques avantages de l'approche par objet fragmenté sont exposés.

Nous avons défini un langage à objets fragmentés appelé FOG, un compilateur, et une boîte à outils d'objets fragmentés de bas niveau. Le compilateur assure l'encapsulation, vérifie la cohérence des interfaces et génère le code d' accès à la boîte à outils. Le compilateur et la boîte à outils facilitent la réalisation des politiques de répartition les plus courantes.

Contents

1 Introduction **1**

2 Fragmented Objects **3**

2.1 Client access to a FO 5

2.2 Interfaces 6

2.3 Connective objects 6

2.4 Binding 9

3 Structuring a naming service as FOs **10**

3.1 Object structuring of the SOS Name Space 10

3.1.1 Distribution of function within SOS_NS 11

3.1.2 Discussion 13

3.1.3 Distribution of function within SOS_NS_Tree 13

3.1.4 Discussion 15

3.2 Naming views 15

3.2.1 Naming view fragmented object 15

3.2.2 Discussion 18

4 Tools for fragmented objects **19**

4.1 Toolkit of low-level fragmented objects 19

4.2 FOG language 21

4.3 FOG compiler 22

5 Related work **25**

6 Conclusion **27**

Chapter 1

Introduction

The object-oriented programming methodology is increasingly recognized of primary interest for structuring large, extensible, flexible, long-life software. It could be extended to structuring distributed applications; yet, existing object-oriented programming languages or systems do not take distribution into account.

In the object-oriented approach, two objects may communicate via the interface of a third, shared object. In a distributed object-oriented system, this would naturally extend to a distributed shared object.

For instance, one may consider a communication channel as a distributed object, with two interfaces, one for each endpoint: send(bytestream) at one end, and receive(bytestream) at the other. However, it is not satisfactory to support only a fixed collection of low-level communication objects (such as channels): firstly, because the data they carry is untyped; secondly, because the object-oriented approach is chiefly concerned with designing high-level object types, with an interface and semantics appropriate for the application at hand.

We remark that the fragmentation of a single logical entity across several locations is useful in many distributed applications. For instance, all the replicas of a replicated file logically form a single file. Its interface hides the existence of multiple replicas, and of internal communication to maintain consistency. The replicated file is a shared distributed object.

We propose the uniform concept of a *fragmented object* (FO) for designing and building distributed applications. As is usual in object-oriented approach, a FO has two aspects, external (or "abstract") and internal (or "concrete").

Abstractly, a FO appears (to its external clients) as a single entity. It is accessed via a programmer-defined interface. Its components, and in partic-

ular their distribution, are not visible.

Internally, the FO encapsulates a set of cooperating *fragments*. Each fragment is an elementary object (i.e. with a centralized representation). The fragments cooperate using lower-level FOs, such as communication channels. The programmer of a FO (its *implementor*) controls the location and the communication between the fragments. This addresses distribution issues (such as where to place a particular data element, or how to handle failures) according to the semantics, and expected usage pattern, of the FO.

The link between the external and the internal view is the interface exported by the FO. One critical observation is that, for a particular client, the interface is obtained through a particular fragment.

The SOR (Systèmes à Objets Répartis – Distributed Object-Oriented Systems) group at INRIA-Rocquencourt (near Paris, France) has implemented a distributed object-support operating system, called SOS, based on FOs [19]. In SOS, a distributed object manager [8], communication protocols [13, 14], a distributed name service [11], and some user-level applications are structured as FOs.

This paper has two aims. Firstly, it presents the fragmented object concept, and tools to support it. Secondly, it identifies the benefits of structuring distributed applications as fragmented objects.

The rest of the paper is organized as follows. Chapter 2 defines FOs. Chapter 3 details the example of a highly distributed, layered, user-centered, naming service. Chapter 4 presents some tools for FOs. These include a toolkit of low-level FOs; a language called FOG its compiler. In chapter 5 we compare our approach to related work. Finally, chapter 6 concludes the paper.

Chapter 2

Fragmented Objects

A fragmented object can be viewed at two different levels of abstraction, corresponding respectively to the client's (external, abstract) view, and to the implementor's (concrete, internal) view.

For clients (see Figure 2.1), a FO is a single shared object. It is shared by several client objects, localized in different address spaces, possibly on several sites. A FO can offer distinct, strongly-typed, interfaces to different clients.

For the implementor (see Figure 2.2), it is an object with a fragmented representation. It is composed of:

- A set of elementary objects, its *fragments*. Each fragment is mapped within only one of the many address spaces overlapped by the FO.
- A client interface. The FO's interface is presented to each client via the public interface of a local fragment.
- An interface between fragments, called its *group interface*.
- Lower-level shared FOs for communication between fragments, called *connective objects*.

In addition, a *binding* interface allows clients to bind to the FO.

For the operating system, a FO is a group of elementary objects, with a common inter-address-space communication privilege. The privilege is checked by the primitive connective objects, implemented by the system: e.g. communication channels, or shared memory regions.

In the rest of this chapter, we define more precisely the construction of a FO. First we show how a client accesses a FO (section 2.1). Then section 2.2

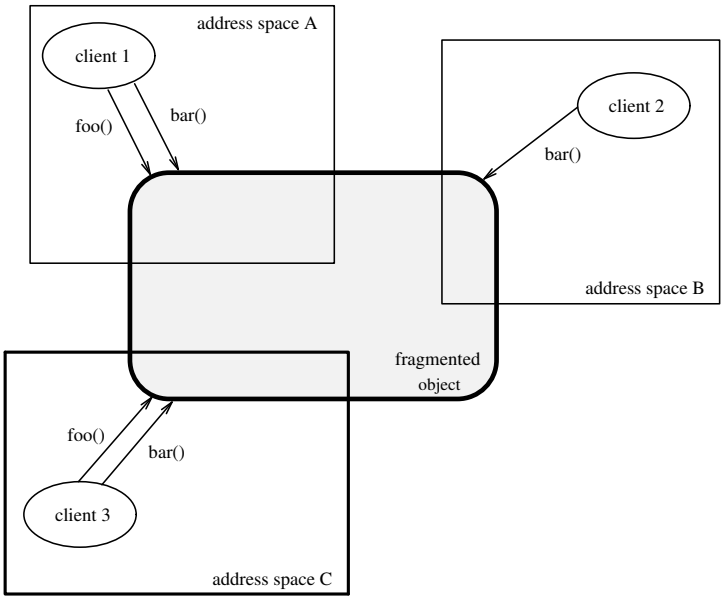

Figure 2.1: A fragmented object as seen from clients

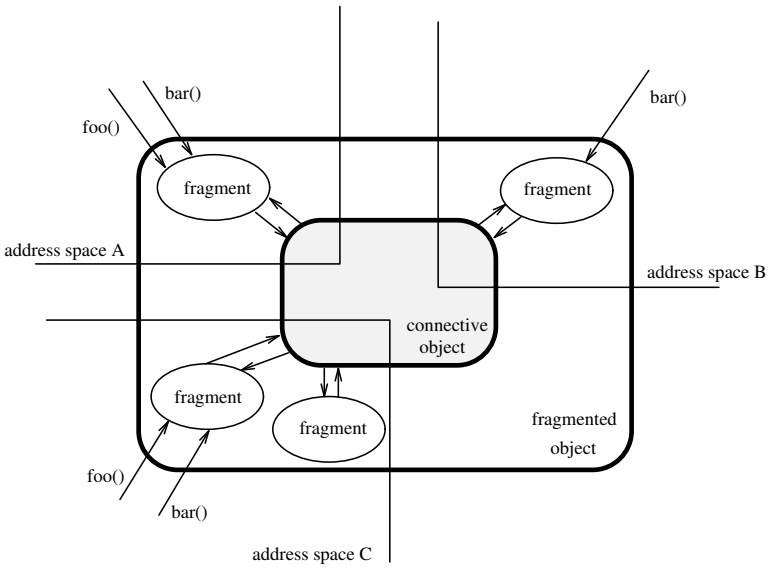

Figure 2.2: A fragmented object as seen by its implementor

takes a look at the various interfaces of the FO. Section 2.3 shows how connective objects are used. Finally, we speak a bit about the binding protocol (section 2.4).

2.1 Client access to a FO

The abstract interface of a FO is provided to some client by a local interface fragment of that FO. A fragment is an ordinary local object. Its public interface may be invoked locally. The client can not distinguish between the interface of the fragment, and that of the FO itself[1].

For instance, a mailbox object, implemented by a central mail server, can be made accessible by proxies exported to users. A mail user with no special knowledge will bind to a particular mailbox, then call the drop(letter) or pickup(letter) methods[2] of his local proxy.

The interface provided to a client of a FO is defined by a contract, ensuring that every method it will invoke is effectively implemented by the FO. The client expects a specific interface; the FO provides that client with a proxy possessing this interface[3].

For instance, a fragmented mailbox will export, to its owner exclusively, a proxy allowing to pickup messages. Other users will get a drop-only proxy. It is up to the fragmented mailbox to check the identity of the user. Accordingly, it provides the user with an appropriate interface and implementation of that interface.

It is up to the compiler and the run-time to verify (as described in section 4.3, chapter 4) that the actual interface conforms to the one expected by the client.

The interface of the fragment offers transparency of the distribution to a client. A method of the fragment interface can be entirely implemented by the fragment itself, or it can trigger invocations to other fragments.

---

[1] An interface fragment is called a *proxy* in [17]. It may hold local data, and process computations locally, or else forward them for processing to remote fragments. Remote communication entails marshalling/unmarshalling invocation parameters into/from a communication message. A *stub* is special case of a proxy, performing no local processing, and reduced to the communication function [4].

[2] A *method* is a procedure associated with an object. In C++, methods are also called *member functions*.

[3] We allow different clients to see different interfaces to the same FO.

2.2 Interfaces

The concrete representation of a FO is fragmented on several address spaces. The implementor considers criteria such as protection, efficiency and availability, to decide the distribution of the fragments. The implementor has full control over the fragmentation of data among fragments, the localization and movement of fragments (e.g. by migrating them), and the cooperation protocol (by choosing an appropriate connective object).

A fragment's interface is divided in three distinct parts:

- the *public* (or client) interface contains methods accessible by clients;
- the *private* interface is composed of internal methods, accessible only from within the fragment;
- the *group* interface comprises those methods which are internal to the FO as a whole (i.e. which can be invoked remotely from other fragments).

For instance, consider the implementation of a replicated file file as a FO. Each replica constitutes a fragment (see Figure 2.3). The client interface to file is record-based: read(**out** record r), write(**in** record r), seek(**in** int position)[4]. The group interface is different, being block-based: put(in int blocNo, **in** block b)[5]. Figure 2.4 shows the declarations of this example in the FOG language. We will explain the syntax of this language in section 4.2, chapter 4.

In the file example, all the put methods of the replicas constitute the group interface of the fragmented object. The type-checking of the group interface ensures strongly-typed communications between fragments. Communication between fragments is carried out by connective objects.

2.3 Connective objects

A connective object is just another FO at a lower level of abstraction.

The most primitive connective objects are the communication objects implemented by the system, with a fixed, predefined interface. At instantiation time, a primitive communication object checks that the connected ends are

[4]Meaning: read is a method taking no input parameter, and returning a result of type record; write takes a input parameter of type record, and returns no result; seek takes an integer parameter called position.

[5]There is no need for a get() operation if file is fully replicated.

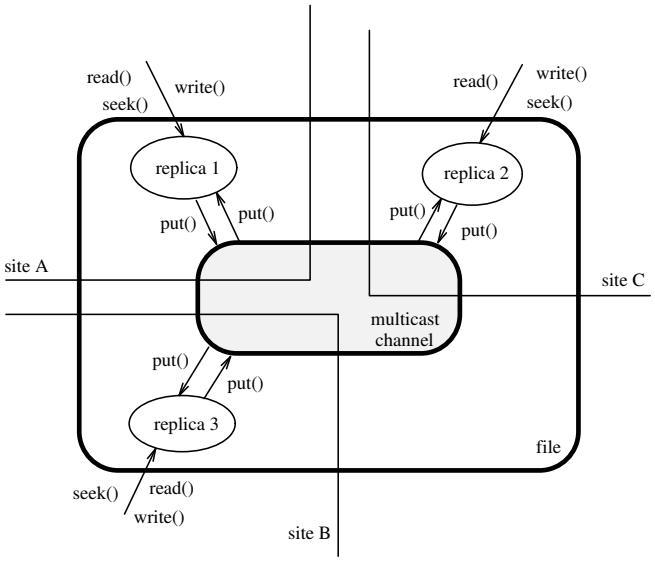

Figure 2.3: The fragmented representation of a replicated file

```
// A class group defining the FO type file
group file { replica };

// the class of file fragments
class replica
{
public:             // the public interface
        read (out record r);
        write (in record r);
        seek (in int position);
group:              // the group interface
        put (in int blockNo, in block b);
};
```

Figure 2.4: Declaration of file in the FOG language

indeed allowed to communicate directly, i.e. that they are fragments of a same FO.

Communication objects implement the basic communication facilities, such as communication protocols (e.g. remote procedure call, asynchronous remote procedure call, parallel remote procedure call, functional remote procedure call, etc.), cacheing, replication, or other distribution policies.

The types of parameters in the group interface of the superior FO may be totally unrelated to the interface of the connective object. This necessitates "marshalling/unmarshalling" type coercions. Returning to the example of the replicated file, Figure 2.5 illustrates this issue. The group interface of the file FO is the put method. Each invocation of put maps onto a send(message) at one end of the transport communication channel and a recv(message) at the other end. The (int, block&) parameter list must be coerced into the message datatype. Conversely, a recv(message) is mapped onto an upcall to put(int blockNo, block& b).

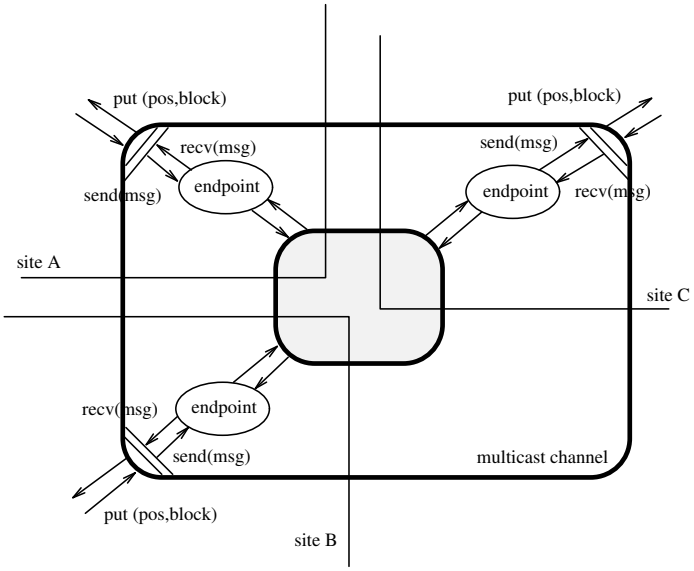

Figure 2.5: Invocation coercions

Our type-safe treatment of inter-protocol-layer coercion is further detailed in section 4.3, chapter 4.

2.4 Binding

Just as an ordinary object must be instantiated before use, a client must first *bind* to a FO. To request access to some particular FO, the client invokes a binding procedure, associated with the type of the expected interface. This procedure returns a proxy. The binding procedure is programmer-defined[6]. Typically, it will open a connection to other fragments.

The implementor of a FO may rely on a system-defined binder, such as ANSA's *Trader* [9], which maintains mappings of interface descriptions to servers. The standard binding procedure in ANSA is to contact the trader with an interface description, which returns a server connection. Thereafter the client uses a stub encapsulating that connection. This approach fits well for simple services, which do not require a dynamic selection of the specific implementation of the interface requested by a client.

In SOS, we generally use a more involved binding procedure, giving the FO implementor more control. A binding has three steps. In the first step, a name lookup (similar to ANSA's Trader lookup) yields a *provider* object for the named interface. In the second step, the binding request is forwarded, by the distributed object manager [8, 19], to a particular method of the provider. In the third step, this method may dynamically instantiate a proxy implementation, based (for instance) on the user's identity, on the binding request arguments (e.g. type of access required), on the type of the underlying system or architecture, or on the load of the client's host.

These two binding approaches are, each, appropriate for certain classes of applications. None is satisfactory for all. It is up to the implementor to choose the one which fits its needs best.

[6]Similar to an instantiation procedure (a "constructor" in C++).

Chapter 3

Structuring a naming service as FOs

In this chapter, we apply the FO approach to a layered naming service [11], implemented for the SOS object-support operating system [19]. In the following chapter (chapter 4) we will present various programming tools, used to build such FOs.

A naming service associates names to objects. The one presented here is layered according to its two complementary functions:

- the bottom layer is formed of Name Space objects;
- the top layer is made of Naming View objects, each of which presents a specific view of the underlying Name Spaces.

3.1 Object structuring of the SOS Name Space

A Name Space is a homogeneous set of persistent name/object associations. Each Name Space may have its own syntax or naming conventions. Some well-known Name Spaces are a Unix file system, a list of authorized users for some computer installation, or the Internet host names. Hereafter, we detail the design of the SOS Name Space.

The SOS Name Space is a tree of names, decomposed into parts maintained by separate servers. It is partitioned similarly to the V-System naming space [6]: each server holds a vertical slice, starting at the root of the tree. The SOS Name Space implements two complementary functions:

- to receive, and act upon, client requests;

- to maintain a consistent image of the tree by peer cooperation between servers.

The first function constitutes a FO, called SOS_NS, with three kinds of fragments: sos_clientProxy, sos_adminProxy and sos_serverStub (associated with normal clients, administrator clients, and servers, respectively).

The second function constitutes another FO, called SOS_NS_Tree; it is composed of two kinds of fragments: sos_treeStub and sos_treeProxy (both associated with servers).

Furthermore, each server instantiates a sos_partMgr object, which manages its own part of the naming tree. The sos_partMgr objects accomplish all the server's management activities of the naming tree; the other objects (sos_serverStub, sos_treeStub and sos_treeProxy) only deal with distribution. In this article, we will not detail the sos_partMgr object because its role is not central to the FO-oriented features of the SOS Name Space.

Sections 3.1.1 and 3.1.3 present the SOS_NS and SOS_NS_Tree FOs. The corresponding declarations in the FOG language are presented later, in Figure 4.2, and, are explained in section 4.2, chapter 4.

3.1.1 Distribution of function within SOS_NS

Let us now describe in more detail the fragments of a SOS_NS and their cooperation (see Figure 3.1).

The FO type SOS_NS is composed of fragments of type sos_clientProxy and sos_serverStub, connected by a multicast channel connective object.

When a server boots, it binds to SOS_NS, which instantiates a sos_serverStub. The requests in its group interface, groupAddName and groupLookUp are simply passed on to the local sos_partMgr object.

A client of the SOS_NS (i.e. a naming view object) binds to SOS_NS with the result of instantiating a sos_clientProxy. This object type, with a public interface allowing to add and look up names (addName/lookUp), manages a local cache of name prefixes. Each entry associates a prefix with the server, or set of servers, known to manage the corresponding subtree. For administrator clients, the sos_adminProxy object type inherits from sos_clientProxy. It extends its interface with the addNode operation for controlling the mapping of nodes to servers.

An execution of the lookUp method of a sos_clientProxy first looks up the argument in its cache. The longest matching prefix corresponds to a remote subset of sos_serverStubs. Their groupLookUp method is invoked (by

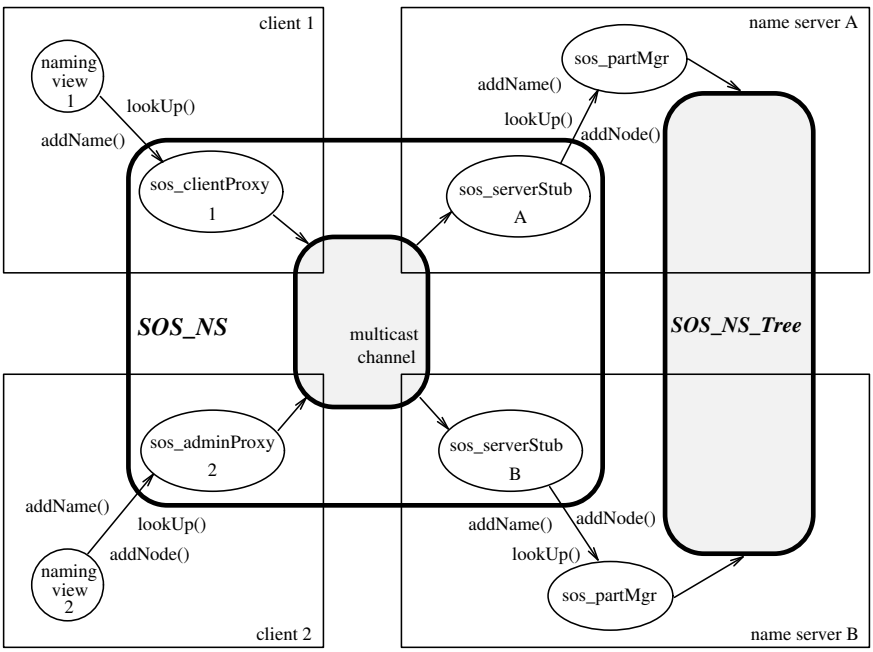

Figure 3.1: Distribution of function in SOS_NS

a multiple RPC on the multicast channel); their replies are collected and returned to the client.

Similarly, an execution of a sos_clientProxy's method addName looks up, from the cache, the longest matching prefix of the name. This time however, the groupAddName method of a single sos_serverStub is invoked remotely (by a selective RPC on the multicast channel). Its reply is then returned to the client.

3.1.2 Discussion

The flexibility of the FO approach is helpful for SOS_NS. It would not be available with a standard stub generator.

The SOS_NS FO offers two different interfaces to clients: sos_clientProxy for a normal client, and sos_adminProxy for administrators. Externally, the latter differs from the former only by having the additional addNode method to administer the mapping of nodes to servers. Delivery of these proxies is controlled by the SOS_NS binding protocol; a provider object decides, depending on the rights of the requester, to deliver either a sos_clientProxy or a sos_adminProxy.

The implementor of the SOS_NS fragmented object controls the distribution of data within the different fragments. A sos_clientProxy has client-specific data (the local cache), whereas a sos_serverStub doesn't have any.

There is a clear distinction between the client interface, implemented by the sos_clientProxy objects (through which a naming view object accesses to the SOS Name Space), and the group interface, implemented by the sos_serverStub objects. A sos_clientProxy offers forwarding methods to invoke the fragmented methods of the group interface.

3.1.3 Distribution of function within SOS_NS_Tree

Let us now describe in more detail the fragments of a SOS_NS_Tree and their cooperation (see Figure 3.2).

The FO type SOS_NS_Tree is composed of fragments of type sos_treeProxy and sos_treeStub, connected by a multicast channel connective object.

When a server boots, it binds to SOS_NS_Tree (in addition to SOS_NS as explained in section 3.1.1), which instantiates a sos_treeStub and a sos_treeProxy. The requests in the sos_treeStub group interface, groupVerify and groupAdd, are simply passed on to the local sos_partMgr object.

When a sos_partMgr receives an addName request, two cases may occur : either, it holds the vertical slice in which the name must be registered, or

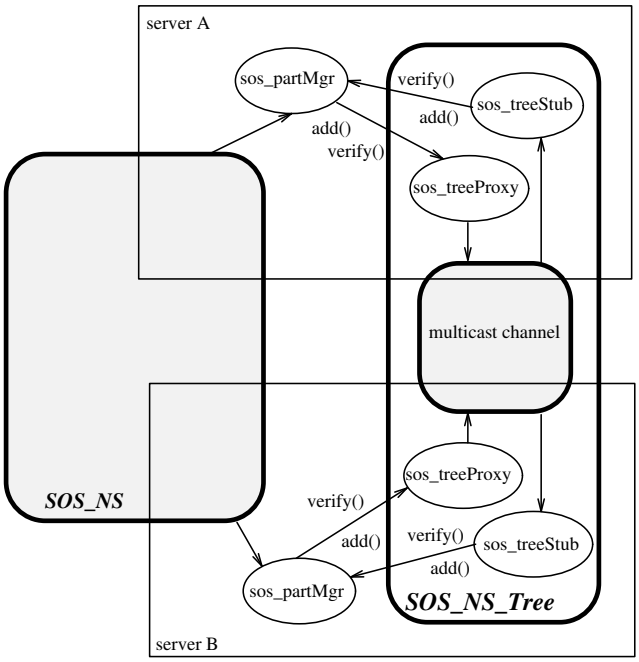

Figure 3.2: Distribution of function in SOS_NS_Tree

it doesn't manage it. In the first case, the sos_partMgr has to verify if the name is not already registered in another name server[1]. In the second case, it has to forward the request to a server managing the vertical slice. For both cases, the sos_partMgr calls the appropriate method of the sos_treeProxy client interface which, in turn, invokes the group interface (groupVerify/groupAdd) of the sos_treeStub objects.

3.1.4 Discussion

The impacts of the FO approach for designing the SOS_NS_Tree FO are the following.

It hides the distribution of the naming tree to a particular sos_partMgr. A sos_partMgr object only knows that it manages itself a part of the naming tree, and that the rest of the tree is managed somewhere else.

The implementor defines a static binding protocol instantiating a sos_treeProxy and a sos_treeStub at the server's boot time.

There is a clear distinction between the client interface, implemented by the sos_treeProxy objects and the group interface implemented by the sos_treeStub objects.

3.2 Naming views

A Naming View is a window into a selected set of (portions of) Name Spaces. Whereas Name Spaces reflect administrative management constraints, a Naming View represents the naming configuration choice of a user.

We call our name service *user-centered* because each user accesses names via one or more specifically tailored views, which are guaranteed to keep the same meaning even as the user moves around the system. Some examples of useful views are: the set of binaries for the current machine architecture; the view of the company-wide information files; the user's current working set.

3.2.1 Naming view fragmented object

In SOS, a Naming View is a tree, local to a client, in which leaves are Name Space proxies (see Figure 3.3). Clients of a Naming View submit requests to different Name Spaces through its client interface.

[1] Due to the partitioning feature of the naming tree, a name may have several authorities distributed among name servers.

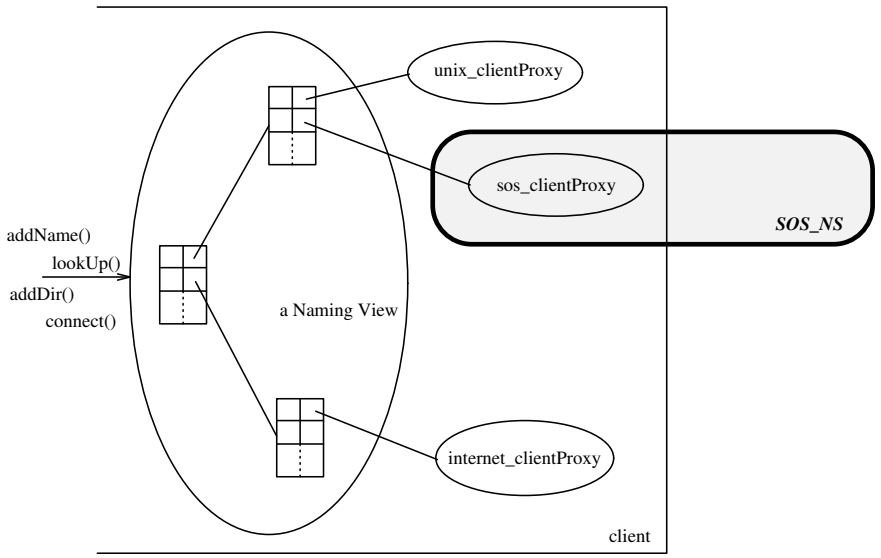

Figure 3.3: A naming view

The Naming View itself is not fragmented. Fragmentation arises when several users share a common Naming View (for instance, a company-wide view shared by its members).

The sharing of a Naming View constitutes a FO, called NAMING_VIEW, composed of viewReplica fragments (see Figure 3.4).

A shared Naming View is replicated in several address spaces. When a client modifies the viewReplica object, the modification must be propagated to all replicas through the group interface. To ensure consistency, replicas are connected by an atomic multicast channel primitive FO.

When a client user process logs in, it binds to NAMING_VIEW, which instantiates a viewReplica, based on the request arguments provided by the client (the symbolic name of the requested Naming View[2]) and on the user's rights. The public interface of this object allows to connect (connect) Names Spaces (i.e. their proxies), to modify the Naming View by adding directories (addDir) in the local tree, and to add and look up names (addName/ lookUp) in the connected Name Spaces.

An execution of the connect method of a viewReplica first multicasts the groupConnect method on all viewReplicas. Then, each viewReplica receiving the request binds to the right FO implementing the requested Name

[2]Naming Views are registered, i.e. named, in a predefined Name Space.

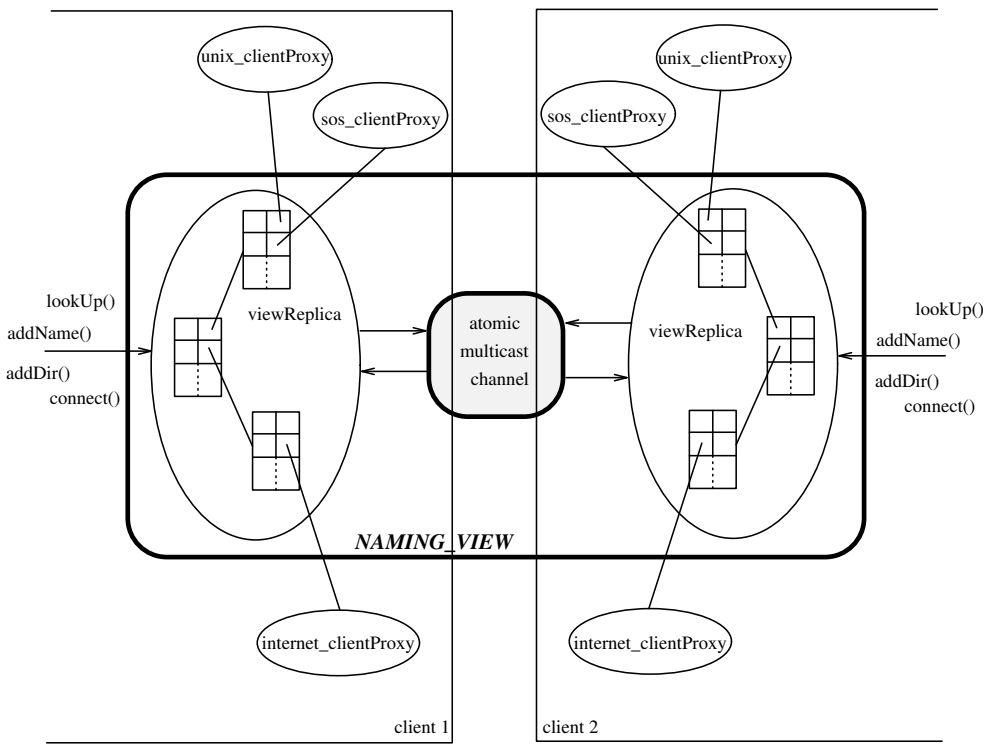

Figure 3.4: The NAMING_VIEW FO

Space (for instance, the SOS_NS as explained in 3.1.1) and adds the mapping in its local tree.

An execution of the addDir method of a viewReplica first multicasts the groupAddDir method on all viewReplicas. Each viewReplica receiving the request modify its local tree by creating the directory and adding the mapping.

The lookUp and addName method are simply forwarded to the underlying Name Spaces.

An atomic ordered multicast is made on the connective FO to avoid inconsistency between the viewReplicas in case of concurrent execution of connect or addDir methods.

3.2.2 Discussion

The main advantages of the FO approach for designing the NAMING_VIEW FO are the following.

The clear separation between client interface and group interface of the NAMING_VIEW FO allows to hide the replication of the viewReplica object to the clients.

The uniform support for communication should allow to easily exchange the basic FO used for a more sophisticated one, with better replication management policies.

Chapter 4

Tools for fragmented objects

The fragmented object concept encompasses the specification of a group of fragments, of internal cooperation, of the different interfaces and of a mechanism for binding fragments to clients.

A specialized language is helpful to write these specifications. Its compiler will check the correctness of the specification (e.g. type-compatibility of the group interface) and automatically generate code for common cases (such as marshalling/unmarshalling parameters). The language is complemented by a toolkit of low-level FO types, which we present first. Then section 4.2 presents the language and section 4.3 the compiler.

4.1 Toolkit of low-level fragmented objects

Basic FOs like communication channels, semaphores, and mutex locks are used by several distributed applications. To assist the implementor of FOs, we provide a toolkit of such general-purpose low-level FOs. Future extensions of the toolkit include support for distributed shared virtual memory, and for multiple replication and cacheing policies.

For a FO implementor, the toolkit will offer a set of predefined low-level FOs for communication, synchronization, concurrency control, fault-tolerance, etc.

Currently, our toolkit contains essentially communication channels [14]. We distinguish two families of communication channels: point-to-point and multi-points communication channels. The former is a FO connecting two fragments, and the latter a FO interconnecting a set of fragments. The two basic communication channels are defined as follows.

Simple communication channel type. A simple communication channel

FO has two kinds of fragments: a channel object and a channelStub object associated respectively with the caller and the callee fragment.

The rpc and send methods of channel constitute the client interface of this FO (see Figure 4.1). The rpc (resp. send) method implements remote procedure call (resp. asynchronous remote procedure call).

Upon invocation, each of these two methods invokes its remote counterpart (a member of the group interface of this FO) implemented by channelStub. The role of these counterparts is to invoke the appropriate method, belonging to the group interface of the superior (i.e. the callee) fragment.

Multicast communication channel type. A multicast communication channel FO has two kinds of components: a multicastChannel object and a multicastChannelStub object, associated with each fragment of the interconnected set.

The class multicastChannel (resp. multicastChannelStub) inherits from channel (resp. channelStub).

In addition to the methods inherited from channel, the client interface of multicastChannel offers two new methods, parpc and multisend. parpc implements a parallel remote procedure call to the components of a fragmented method[1]. multiSend is the non-blocking version of parpc.

Upon each invocation of the parpc or multiSend method, the multicastChannel object forwards this to the group of multicastChannelStub objects associated with the set of interconnected fragments.

For brevity we omit the description of channelStub and multicastChannelStub. Internally, each low-level FO is built by, plugging together, elementary protocol objects. Each protocol object implements a specific function (e.g. message fragmentation/reassembling, ordering, synchrony, or multiplexing and dispatching). A family of related objects (i.e. implementing the same type of function) are organized in a class hierarchy, with the same interface [14].

This building block approach, plus the object-orientation, facilitate the extensibility of the toolkit. For instance, one can define a new protocol object by inheriting, redefining or composing existing protocol objects. Once new protocol objects are available, one will define easily new low-level FOs.

Future extensions of the toolkit include support for distributed shared virtual memory, and for multiple replication and cacheing policies.

[1] A fragmented method is a set of methods implemented by several fragments of a FO. They have the same behavior, and execute in parallel part of client requests.

```
// point–to–point communication
class channel {
public:
        rpc (in message im, out message rm);
        send (in message m, out result r);
};

// group communication
class multicastChannel : channel            // inheritence
{
public: // in addition to the interface inherited from channel
        parpc (in message m, out multiRes r, in reference callee);
        multiSend (in message m, out multiRes r, in reference callee);
};
```

Figure 4.1: FOG declarations of communication channels

4.2 FOG language

We have defined a language extension to C++ [20], called FOG (Fragmented Object Generator) [7]. It provides features for the implementor to specify class groups, group interfaces, client interfaces, and accesses to connective objects.

Just as an elementary object is an instance of a class, a FO is an instance of a *class group*. The class group defines the behavior and representation of the FO by listing the classes of the fragments. These in turn specify the public, private and group interfaces, as well as the component objects (fragments and connections). Just as there can be several instances of a class, several fragmented instances can be instantiated from the same class group.

Figure 4.2 presents a part of the FOG declarations for SOS_NS (as described in section 3.1.1, chapter 3 and Figure 3.1). The class group SOS_NS is composed of three fragment classes: sos_clientProxy sos_adminProxy and sos_serverStub.

The group interface of SOS_NS is made of three methods: groupLookUp, groupAddName and groupAddNode. These are all declared in class sos_server-Stub.

The remote methods of the group interface are invoked by *forwarding*

methods[2] of sos_clientProxy and sos_adminProxy, signaled by the "!" syntax.

A client invokes SOS_NS via the public interface of its local instance of sos_clientProxy or sos_adminProxy.

These three interfaces are related as follows. Consider for instance that a client wants to look up a name. First, it invokes the lookUp method. This method is part of the client interface of a clientProxy. The lookUp method, after checking the cache, invokes the private forwardLookUp forwarding method. The second argument of forwardLookUp is preceded by the keyword **future**, a syntax for multiple incremental results. Its third argument is passed to the communication channel object; it designates the subset of servers to which this group invocation is addressed. Finally, this forwarding method forwards the invocation, through the multicast channel (mchan), to the remote groupLookUp method.

Chapter 3 faithfully described the Name Service currently in use in SOS. It was written (in a slightly extended version of C++) before the FOG compiler was available. The fragmented objects described there were therefore written by hand in C++. The design of FOG draws upon this experience and other distributed applications written for SOS. FOG remains to be tested on a large-scale application.

4.3 FOG compiler

The FOG compiler verifies the correctness of the group and client interface declarations.

First, for each method of the group interface, there must exist at least one forwarding method implemented by a member of the class group. Also, for each forwarding method, there must exist a corresponding method in the group interface. Their signature must match.

Second, it checks that the public interface of the FOs, as proxies delivered to some clients, conforms to what the client expects. Fragments are located in different address spaces, compiled separately, and instantiated at different times. Therefore, in addition to the compile-time checks, run-time checks are needed. Interfaces are checked (i.e. that fragment classes belong to a common class group) at fragment instantiation time, either compile time or run-time. The FOG compiler generates a 32-bit key [18] for the interface expected by the client and for the interfaces, checked for compatibility based on the inheritance mechanism. Communication privileges (i.e. that fragment

[2] A forwarding method is similar to the traditional RPC stub method: its role is to forward the invocation to its corresponding method(s) of the group interface.

```
// The SOS_NS FO type
group SOS_NS { sos_clientProxy, sos_adminProxy, sos_serverStub };

// The fragment classes
class sos_serverStub
{
 group:                              // group interface
      groupLookUp(in String name, out reference obj);
      groupAddName(in String name, in reference obj);
      groupAddNode(in String name, in reference node);
};
class sos_clientProxy
{
public:                              // client interface
      lookUp (in String name, out reference obj);
      addName (in String name, in reference obj);
protected:                           // private data and methods
      cache tableOfPrefixes;         // the prefix cache
      multicastChannel mchan;        // a multicast channel FO

      // multicast to many servers
      forwardLookUp(in String name, future reference obj [ ], in reference callees)
         ! mchan.parpc (callees) ! sos_serverStub::groupLookUp(name,obj);

      // rpc to single server
      forwardAddName(in String name, in reference obj, in reference callee)
         ! mchan.rpc (callee) ! sos_serverStub::groupAddName(name, obj);
};
class sos_adminProxy : sos_clientProxy // inheritence
{
public:                              // local interface
      addNode (in String name, in reference node);
protected:                           // private interface
      // in addition to the interface inherited from sos_clientProxy
      addNode (in String name, in reference node, in reference callee)
         ! mchan.rpc (callee) ! sos_serverStub::groupAddNode(name,node);
};
```

Figure 4.2: Declaration of a class group

instances belong to a common FO instance) are checked at binding time by the binding method of the primitive (system-defined) communication FOs based on a common capability.

The FOG compiler also plays the role of a stub generator in traditional RPC packages: it generates the code and structures necessary to marshal/unmarshal parameters of group interface invocations. It also generalizes traditional stub generation to handle parallel and/or asynchronous invocations, handling of exceptions, and so forth.

The FOG compiler can generate marshalling/unmarshalling coercion methods based on different interfaces of communication objects. The FOG stub generation allows flexibility and extensibility of interfaces: it automatically provides a glue between forwarding methods of an FO and the interface offered by its connective object.

For instance, the parameters (string, reference), of the forwardLookUp method of the sos_clientProxy object, are coerced into a message (see Figure 4.2). This message is then passed to the parpc method of the multicastChannel object, which in turn, invokes the set of sos_serverStub objects. At the reception side, this message is coerced into the types expected by the groupLookUp method of the sos_serverStub objects, also (string,reference) (see Figure 4.1). These two coercions are handled automatically by the FOG compiler.

Currently, coercions are done according to a hardwired external representation of data. As an extension, we plan to replace this with programmer-defined coercions. These are guaranteed to be type-safe, because the compiler has checked the group interface.

Chapter 5

Related work

Our approach could be compared with standard RPC stubs, e.g. Birrell and Nelson's work [4]. A stub is a placeholder object for a remote server, automatically generated from an interface desciption. It carries no local data or processing, having only the capability to marshal/unmarshal arguments and forward invocations to the server. A stub remotely extends access to the server object, but does not offer the flexibility of a fragmented object.

Several projects have proposed distributed extensions of the object paradigm: e.g. Orca's shared objects, Gothic's Fragmented Objects, Emerald, Amber, Comandos, and Topologies.

In Orca [1], shared objects are dynamically replicated under the control of the Orca run-time. For its clients, a shared object is a single object. Orca ensures transparent access to the replicas and guarantees their consistency. Orca's run-time decides to create new replicas, or to migrate the object on to the sites where the object is frequently used, based on statistics of recent access.

Orca hides distribution to both the clients and implementors of a shared object. This simplifies their task. However this fully automatic approach has drawbacks. For instance, it is not appropriate to applications such as system services for two reasons. First, replication and migration are just specific fragmentation policies. Second, a specific type of consistency is not suitable for all applications. In our model, the implementor considers criteria such as protection, efficiency and availability to decide what is partitioned or replicated and the number of replicas. As in Orca, distribution is transparent to a client of a fragmented object. We can also provide automatic mechanisms similar to Orca.

Gothic's fragmented objects [3, 12] are based on the "multi-functions" [2], a parallelized generalization of procedures to N callers and P callees. Multi-functions are powerful for expressing parallel computing because syn-

chronization is implicit. However, they do not address the issues of sharing objects between applications in a distributed system.

Although we use the same name, our FO model differs from Gothic in three ways. Firstly, we focus on distributed, rather than parallel, computations. Secondly, we give the implementor the full control over the distribution, rather than only the automatic mechanisms provided by the Gothic system (such as automatic creation and placement of fragments, and the choice of a communication protocol). Thirdly, we support multiple client interfaces, whereas Gothic enforces a single global interface to a fragmented object.

The Comandos [15] approach is opposite to ours. In Comandos, shared objects are not fragmented. Instead, address spaces are fragmented across sites. When an activity requests access to a remote shared object, its address space "diffuses" over to the site of this object. The simplicity of this approach has drawbacks similar to Orca.

In Amber [5], a single address space is fragmented on several sites. Each object is referenced by a unique virtual address and is localized on a single site. As in Emerald [10], the sharing of an object is implemented either by migrating the object or by diffusing the execution.

Topologies [16] bear some similarities to our FOs. They allow programmers to define distributed shared objects on a message-passing multicomputer. Topologies bear a close resemblance to the communication objects of our primitive FO toolkit. [16] describes only communication-oriented Topologies, and seems to lack our general concept of fragmented objects. Their implementation concentrates on high-performance communication on a Hypercube, based on kernel-level Topologies. Consequently, they restrict Topology interfaces to send and receive operations, instead of allowing programmer-defined interfaces.

Chapter 6

Conclusion

This article defined the Fragmented Object concept, an extension of objects to the distributed environment. Its main strength is that it provides the appropriate level of distribution visibility to the implementor of a distributed service, while hiding the distribution of fragments from its clients.

We presented our fragmented object model, which makes a clear distinction between distribution mechanisms, and policies. Policies are programmer-defined. For instance, the model requires a binding procedure for interfaces exported by a fragmented object; it does not impose a specific binding protocol.

Our model, and its implementation in SOS and FOG, offer the following desirable features:

1. Programs communicate via distributed shared objects, i.e. high-level, programmer-defined abstractions.
2. A clear distinction is made between public ("client") and internal ("group") interfaces; both are are strongly typed.
3. Distinct interfaces can be given to different clients.
4. Distribution is transparent for clients.
5. The implementor has full control over the abstract view and the concrete representation of a fragmented object. The representation of data items, their location, and all aspects of communication between them are (if desired) under implementor control.
6. Multiple protocol layers are supported.
7. The compiler supports type-safe "marshalling/unmarshalling" coercions between protocol layers.

8. Uniform support for multiple communication paradigms, such as client-server, peer-to-peer, group communication, multiway rendez-vous, etc.

9. Uniform support for multiple binding paradigms.

We related our experience designing a complex distributed application, the SOS naming service, based on fragmented objects.

We listed some tools available to the fragmented object programmer: a library of primitive fragmented objects, and the FOG language and its compiler. The design of FOG draws upon the experience of the SOS Name Service, and of other distributed applications written for SOS.

Future work includes the support of programmer-defined coercions, and an extended library of primitive communication-oriented FOs, with support for distributed shared virtual memory, and for multiple replication and cacheing policies.

Bibliography

[1] Henri E. Bal and Andrew S. Tanenbaum. Distributed programming with shared data. In *Proceedings of ICCL*, pages 82–91, Miami, FL, October 1988. IEEE, Computer Society Press.

[2] Jean-Pierre Banâtre, Michel Banâtre, and Florimond Ployette. The concept of multi-function: a general structuring tool for distributed operating system. In *The 6th International Conference on Distributed Computer Systems*, pages 478–485, Cambridge, Mass. (USA), May 1986. IEEE.

[3] Jean-Pierre Banâtre, Michel Banâtre, and Florimond Ployette. An overview of the Gothic distributed operating system. Rapport de recherche 504, INRIA, March 1986.

[4] A. D. Birrell and B. J. Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Programming Languages and Systems*, 2(1), February 1984.

[5] Jeffrey S. Chase, Franz G. Amador, Edward D. Lazowska, Henry M. Levy, and Richard J. Littlefield. The Amber system: Parallel programming on a network of multiprocessors. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 147–158, Litchfield Park, Arizona USA, December 1989. ACM.

[6] David R. Cheriton and Timothy P. Mann. Decentralizing a global naming service for improved performance and fault tolerance. *ACM Transactions on Computer Systems*, 7(2):147–183, May 1989.

[7] Yvon Gourhant and Marc Shapiro. FOG/C++: a fragmented-object generator. In *C++ Conference*, pages 63–74, San Francisco, CA (USA), April 1990. Usenix.

[8] Sabine Habert. *Gestion d'objets et migration dans les systèmes répartis.* PhD thesis, Université Paris-6 Pierre-et-Marie-Curie, Paris (France), December 1989.

[9] Dr. A. J. Herbert and Prof. J. Monk, editors. *ANSA Reference Manual.* Advanced Networked Systems Architecture, Cambridge (United Kingdom), June 1987.

[10] Norman C. Hutchinson. Emerald: An object–based language for distributed programming. Technical Report 87-01-01, Department of Computer Science, University of Washington, Seattle, WA (USA), January 1987.

[11] Jean-Pierre Le Narzul and Marc Shapiro. Un service de nommage pour un système répartis à objets. In *Séminaire Franco-Brésilien sur les Systèmes Informatiques Répartis*, pages 127–133, Florianopolis (Breésil, September 1989. LAAS (Toulouse), l'UFSC.

[12] Philippe Lecler. *Une approche de la programmation des systèmes distribués fondée sur la fragmentation des données et des calculs, et sa mise en œuvre dans le système Gothic.* Thèse de doctorat, Université de Rennes I, Rennes (France), September 1989.

[13] Mesaac Makpangou and Marc Shapiro. The SOS object-oriented communication service. In *Proc. 9th Int. Conf. on Computer Communication*, Tel Aviv (Israel), October–November 1988.

[14] Mesaac Mounchili Makpangou. *Protocoles de communication et programmation par objets : l'exemple de SOS.* PhD thesis, Université Paris VI, Paris (France), February 1989.

[15] Comandos Project. Comandos — construction and management of distributed office systems. Final report on the global architecture, Esprit project 834, September 1987.

[16] Karsten Schwan and Win Bo. Topologies—distributed objects on multicomputers. *ACM Transactions on Computer Systems*, 8(2):111–157, May 1990.

[17] Marc Shapiro. Structure and encapsulation in distributed systems: the Proxy Principle. In *The 6th International Conference on Distributed Computer Systems*, pages 198–204, Cambridge, Mass. (USA), May 1986. IEEE.

[18] Marc Shapiro, Philippe Gautron, and Laurence Mosseri. Persistence and migration for C++ objects. In *ECOOP'89*, Nottingham (GB), July 1989.

[19] Marc Shapiro, Yvon Gourhant, Sabine Habert, Laurence Mosseri, Michel Ruffin, and Céline Valot. SOS: An object-oriented operating system — assessment and perspectives. *Computing Systems*, 2(4):287–338, December 1989.

[20] Bjarne Stroustrup. *The C++ Programming Language.* Number ISBN 0-201-12078-X. Addison Wesley, 1985.