Fragmented Objects for Distributed Abstractions

Mesaac Makpangou, Yvon Gourhant, Jean-Pierre Le Narzul, Marc Shapiro

Fragmented Objects for Distributed Abstractions*

Mesaac Makpangou Yvon Gourhant
Jean-Pierre Le Narzul
Marc Shapiro
INRIA, B.P. 105, 78153 Rocquencourt Cédex, France
tel.: +33 (1) 39-63-52-93, fax: +33 (1) 39 63 53 30
e-mail: mak@sor.inria.fr, telex: 697 033 F

October 1, 1991

Keywords: Distributed objects, object oriented programming, distributed abstractions, fragmented objects, connective objects, FOG

Abstract

Fragmented Objects (FOs) extend the object concept to a distributed environment. The abstract view of a FO is a single, shared object, of which the distribution is hidden to clients. In the concrete view the FO designer controls (if wished) the distribution of data and function and of the communication between fragments.

FO programming is supported by the FOG language, an extension of C++, and by a toolbox of predefined FOs. The FOG compiler ensures distributed type-safety of both the external and internal interfaces, verifies the encapsulation of FO instances, and automatically generates whatever coercions are necessary for marshalling/unmarshalling between layers. Currently, the toolbox contains mainly classes of primitive FOs such as RPC and multicast communication channels.

We present the basic fragmented object concepts, the toolbox of predefined FOs, the FOG language and its compiler. We also present an example of a distributed application, the SOS Naming Service, structured as FOs. Finally, we point out the benefits of the FO approach.

*This paper is submitted for publication in the IEEE Software journal. It will also appear in a IEEE Computer Society text.

1 Introduction

The object-oriented programming methodology is increasingly recognized of primary interest for structuring large, extensible, flexible, long-life software. It is natural to try to extend it to distributed applications. Some proposed extensions include support for either client/server remote invocation (as in ANSA [1]) or for the shared object model (as in Orca [2] or Comandos [12]). Both models permit an application object to transparently access another, shared, possibly remote, object. The network is transparent both for the clients and for the designer of a shared object.

Such distribution transparency simplifies the use and the implementation of a shared object. However, it prevents knowledgeable designers from taking advantage of distribution. There may be a need to *fragment* the data and/or the code of a shared object over several locations, for performance, availability, protection or load balancing. These fragments together constitute a single logical entity, shared by the several client objects. Fragments cooperate to maintain a consistent view of this single logical entity. We propose the uniform concept of a *fragmented object* (FO) for designing and building distributed abstractions.

As is common in object-oriented approach, a FO has two aspects, external (or "abstract") and internal (or "concrete"). Abstractly (to its clients), a FO appears as a single entity. It is accessed via a programmer-defined interface. Its components, and in particular their distribution, are not visible[1]. Internally (concretely), 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 designer of a FO has full control over its distribution, and can make good use of knowledge of the object's characteristics; for instance, by placing some particular data element or processing, choosing the most appropriate communication protocol, or handling failures according to the FO's semantics.

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

The FO concept encompasses several abstractions: the grouping of fragments, the specification of cooperation between fragments, the specification of internal and external interfaces, and a mechanism for binding its clients to a FO.

Like the client/server and the shared object models, the FO model sup-

[1] Unless of course the designer chooses to let them show through the public interface.

ports distribution transparency for the clients of a FO. Unlike the traditional model, however, FOs do not impose distribution transparency on FO designers. We specify distribution mechanisms, not policies; specific policies are programmer-defined. However, to facilitate their job, designers may choose existing policies from a toolkit of predefined FOs, implementing various distributed abstractions: e.g., communication protocols, synchronization facilities, and sharing policies.

The fragmented object concept and tools were first implemented as part of the SOS, a distributed object-support operating system [15], which we do not present here. We are currently in the process of porting them to Unix.

This paper has two goals. First, it presents the FO concept, and tools to support it. Second, it identifies the benefits of structuring distributed applications as fragmented objects. The paper is organized as follows. Section 2 defines FOs. Section 3 presents our tools for FO programming, namely the toolkit of low-level FOs, the FOG language, and its compiler. Section 4 details the example of a distributed naming service. In Section 5 we discuss the benefits of FO programming, as exemplified by the naming service example, and compare this approach to related work. Finally, Section 6 concludes the paper.

2 Fragmented Objects

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

For clients (see Figure 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 designer (see Figure 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.

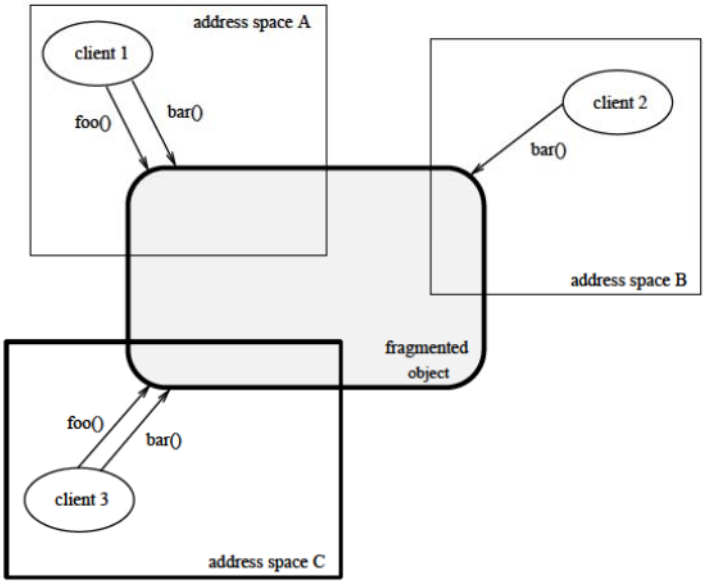

Figure 1: A fragmented object as seen from clients

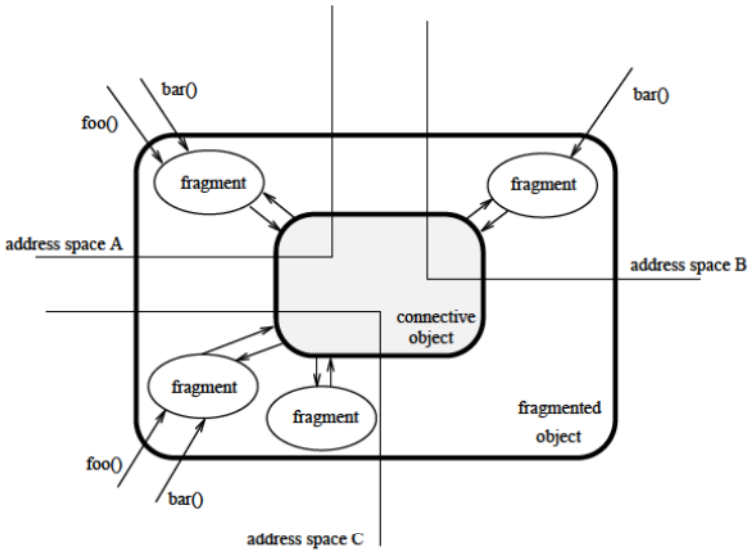

Figure 2: A fragmented object as seen by its designer

In the rest of this section, we define more precisely the construction of a FO. First, we show how a client accesses a FO in Section 2.1. Then, Section 2.2 takes a look at the various interfaces of the FO. Section 2.3 shows how connective objects are used. Finally, we briefly discuss the binding protocol in 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 (or *proxy*[2]) of that FO. A fragment is an ordinary local object. Its public interface may be invoked locally. The client cannot distinguish between the interface of the fragment, and that of the FO itself.

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[3] 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[4]. It is up to the compiler and the run-time to verify (as described in section 3.3) that the actual interface conforms to the one expected by the client.

For instance, a fragmented mailbox will export, to the user who owns it exclusively, a proxy allowing him 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.

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.

[2] An interface fragment is called a *proxy* in [14]. 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 [5].

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

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

2.2 Fragments

The concrete representation of a FO is fragmented onto several address spaces. The designer considers criteria such as protection, efficiency and availability, to decide the distribution of data among fragments.

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 3). The client interface to file is record-based: read(**out** record r), write(**in** record r), seek(**in int** position)[5]. The group interface is different, being block-based: put(**in** int blkno, **in** block blk)[6].

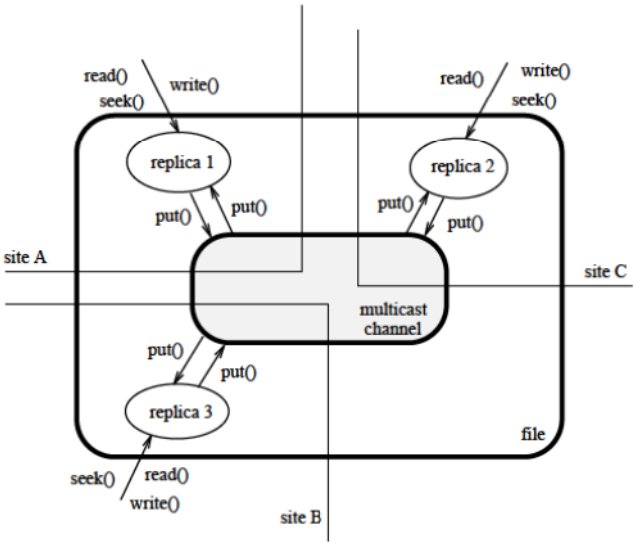

Figure 3: The fragmented representation of a replicated file

[5]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.

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

In the file example, the set of all the put methods of the replicas constitute the group interface. 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 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, and functional remote procedure call).

The types of parameters in the group interface of a FO may be totally unrelated to the interface of its connective object. This necessitates type coercions for "marshalling/unmarshalling". Returning to the example of the replicated file, Figure 4 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(in int blkno, in block blk)**.

Our type-safe treatment of inter-protocol-layer coercion is further detailed in section 3.3.

2.4 Binding to a FO

Just as in the traditional client/server model, a client must bind to an FO before invoking it. There are three ways to perform a binding: local binding, via a system-defined *binder*, or via an FO-specific *provider*.

The simplest case is local binding. When access is needed to a FO, its proxy is instantiated locally, which then uses internal knowledge to connect to its other fragments. This is appropriate for statically-configured services (e.g., the SOS Naming Service presented in Section 4).

For added flexibility, one would use a system-defined binder, such as ANSA's *Trader* [1], which maintains mappings of interface descriptions to

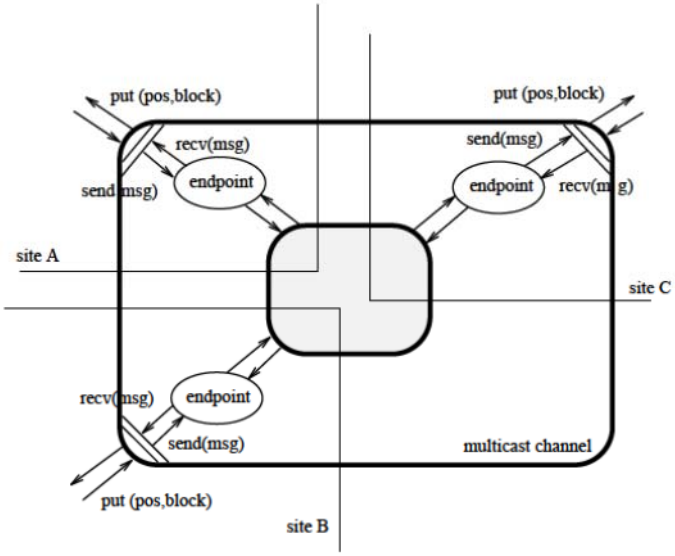

Figure 4: Invocation coercions

servers. An ANSA client interfaces to a distributed service using a stub. At binding time, the stub contacts the Trader with an interface description; this returns a server connection. Thereafter, the stub encapsulates that connection. This approach fits well for many services, that 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 designer 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 [9, 15], 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 three binding approaches are, each, appropriate for certain classes of applications. A FO designer enforces the one which best fits its needs.

The local binding consists of a simple instantiation of a proxy in the client's address space, followed by the establishment of connections between this proxy and other fragments of its FO. For the two other cases, a FO designer, first, declares the type of the binder to use; then, for each client interface, an associated binding procedure is generated. A binding procedure relies on the declared binder to perform the binding. When invoked,

any binding procedure returns a proxy fragment implementing its associated interface.

3 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, in Section 3.1. Then Section 3.2 presents the language and Section 3.3 the compiler.

3.1 Toolkit of Predefined Fragmented Object Types

To assist the programmers in structuring their applications as FOs, we provide a toolkit of predefined FO types, implementing some basic distribution mechanisms. These predefined FOs can be used as connective objects. We first describe briefly the current contents of this toolkit. Then, we present some ongoing work.

3.1.1 Communication Channels

Currently, the toolkit contains mainly communication channels [11]. A channel object offers to the designer of its associated FO (i.e., the FO using it as connective object), a well-defined interface to communicate with one another, through their group interface.

We distinguish point-to-point (between two fragments) and multi-point (between more than two fragments) communication channels.

3.1.2 Point-to-point Channels

The root point-to-point channel type is rpcChannel. A rpcChannel has two kinds of fragment types: channel and channelStub, associated respectively with a client and a server. The public interface of this FO is presented in Figure 5.

A FO of type rpcChannel implements both blocking and non-blocking remote procedure call. All calls and replies are eventually delivered to their destination.

Another point-to-point communication channel is fifoChannel: it has the same public interface as rpcChannel. In addition, invocations are delivered in FIFO order.

3.1.3 Multi-point Channels

The root multi-point channel is multicastChannel. A FO of type multicastChannel has two kinds of fragments: multiChannel and a multiChannelStub. The class multiChannel (resp. multiChannelStub) inherits from channel (resp. channelStub). The public interface of this FO, provided by multiChannel, is presented in Figure 5.

A multicastChannel supports both 1-to-N ("parallel") and 1-to-1-out-of-N ("functional") invocation. The invoked objects may be a subset of the server fragments. Messages eventually reach their destinations, but no ordering is guaranteed.

Other available multi-point channels are fifoMulticastChannel and atomicMulticastChannel channels. They have the same client interface as the multicastChannel type. The fifoMulticastChannel guarantees FIFO delivery. The atomicMulticastChannel guarantees that invocations are delivered to all destinations in the same order.

3.1.4 Future Work

Ultimately, our goal is for the toolkit to support most commonly used distributed abstractions. Our initial focus has been on communication protocols. Several new protocols (e.g., causal broadcast) will eventually be integrated to the toolkit.

We are actively investigating two other domains: concurrency control and synchronization, and replication. FO types implementing semaphores, locks, and rendez-vous, are already specified and will be implemented.

The initial investigation concerning the specification of general FO types for replication is encouraging. Such a FO type should offer a generic interface for concurrency control and for accessing replicated data. Different implementations will offer different consistency policies (e.g., causal consistency, strong consistency).

Other important distributed abstractions such as caching and distributed shared memory, will also be added.

```
// point−to−point communication
class channel {
public:
        // remote procedural call
        rpc (in message, out result);
        // deferred remote procedure call
        send (in message, out result);
};

// group communication
class multiChannel : channel
{       // in addition to the interface inherited from channel
public:
        // parallel remote procedure call
        prpc (in message, out, multiResult);
        // deferred parallel remote procedure call
        psend (in message, out multiResult);
};
```

Figure 5: FOG declarations of communication channels

3.2 FOG Language

We have defined a language extension to C++ [16], called FOG (Fragmented Object Generator) [8]. It provides features for the designer to specify class groups, group interfaces, client interfaces, accesses to connective objects, and how a client binds to an FO. It is essentially a declarative interface language. The designer of an FO specifies interfaces and connections [7].

3.2.1 Class Group

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.

To explain the FOG language, we return to the replicated file example. Figure 6 shows the declarations of this example in the FOG language. More

[7] For simplicity, the declarations presented in this paper take some liberties with the exact FOG syntax.

details can be found in [8]. A more complete example will be described in the next section.

```
// 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);
private:            // the private interface
       update (in int blkno, in block blk)
          ! chan.psend ! replica::put (blkno, blk);
                    // atomic multicast channel
       multicastChannel chan;
group:              // the group interface
       put (in int blkno, in block blk);
};
```

Figure 6: Declaration of file in the FOG language

The file class group is composed of one fragment class replica, which can be instantiated several times in several sites. All the replicas of some file constitute a single fragmented instance.

3.2.2 Interfaces

A client invokes file via the public interface of its local instance of replica, the client interface of file for this client.

The put method of the group interface is invoked by the forwarding method[8] update, signaled by the "!" syntax. This syntax means that an invocation of update invokes the psend method of the connective object chan, that upcalls the put method of the group interface[9]. To ensure consistency between replicas, each one has to obtain a token before accessing the data.

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

[9] To simplify the example, we present only the code for updates.

Consider for instance that a client wants to write a record. First, it invokes the write method of the client interface of a replica. The write method, after executing some code locally (computing the block number and possibly splitting the record across blocks), invokes the private update forwarding method for every block. The update method forwards the invocation, through the multicast channel chan, to the remote put method of all replicas (including the calling replica).

3.2.3 Binding

The FOG compiler helps support the local binding to simple[10] FOs. The FOG compiler generates a default **import** procedure, which performs the local binding: it instanciates a proxy implementing the client interface and connects it to other fragments of its FO; the reference of this FO is passed as argument to this procedure.

For the time being, the FOG compiler provides no support for the binding using a system-defined binder. The main reason for this is that, SOS, our supporting system, has no global binder.

To help support the FO-specific binding approach, the FOG language provides a feature for exporting client interfaces. This is done as follows.

The FO designer specifies a provider class. For each client interface, this class has a particular method, qualified by the keyword **export** and returning a fragment implementing this interface. For each export method, the compiler will generate its associated **import** procedure, with the same arguments, plus a reference which will designate a provider object[11].

When a client wants to bind to a FO, it invokes the import procedure corresponding to the expected client interface; the reference of a provider is passed as argument. The **import** procedure initiates a migration request to the designated provider. The effect is an upcall to the **export** method of the provider, that returns a fragment.

Upon completion, the reference of the fragment is returned to the client by the **import** procedure.

3.3 FOG Compiler

From the interface declarations, the FOG compiler generates C++ interfaces and code, so that clients see FOs as ordinary C++ objects.

[10] There is only one single client interface within a simple FO.

[11] There can be several providers for one FO.

It verifies also the correctness of the group interface declarations to ensure the encapsulation of the fragmentation.

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 the type-safety of the interconnection between fragments of the same FO. 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. At binding time, communication privileges are checked (fragment instances belong to a common FO instance, fragment classes belong to a common class group). The code generated by the compiler checks also the correctness of the connection establishment to a connective object by fragments: these fragments have to verify the first condition on the group interface.

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 deferred invocations, handling of exceptions, and so forth.

Unlike a traditional stub generator (based on a predefined communication primitive), the FOG compiler can generate marshalling/unmarshalling coercion methods based on different interfaces of communication objects: it automatically provides a glue between forwarding methods of an FO and the client interface offered by its connective object.

For instance, the **in** parameters (int, block) of the update method of replica (see Figure 6) are coerced into a message. The compiler looks up the declaration of the field chan of the class replica: multiChannel. Then, it looks up the declaration of the method psend of this class (see Figure 5). Finally, it generates a class putMessage inheriting from message, containing two fields of types **int** and block.

An instance of this putMessage is allocated by the generated update method and is passed to the psend method of the connective object chan, which in turn, invokes all the replica. At the reception side, the putMessage is coerced into the types expected by the put method of replica, also (int,block). There are no **out** parameters. If there were, they would be coerced into a result, when returning.

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.

4 Structuring a Name Service as FOs

The SOS system includes a naming facility, implemented by a distributed Name Service[12], allowing applications to associate symbolic names to objects that they manage.

The SOS Name Service is implemented by server objects instantiated at boot time in separate "naming contexts", and by Name Service proxies, located in the client's contexts. These objects cooperate to process the clients' requests and to ensure the consistency of the distributed Name Space. They constitute a FO, called NameService.

The NameService FO uses two connective objects. The first is a multicastChannel between clients and servers. The second, of type Tree, actually manages the distributed Name Space.

The Tree FO is discussed in Section 4.1. Section 4.2 presents the NameService FO.

4.1 The Tree FO

The tree is decomposed into multiple partitions according a per-server partitioning, similar to the one of V-system[7], in which each part is a vertical slice, starting at the root of the tree. Each partition, managed by a treeFragment, is made of nodes and leaves associated with info data. The set of treeFragments constitute the Tree FO (see Figure 7). The main role of Tree is to manage the distribution of the tree and to ensure its consistency.

Interfaces

The public interface includes three operations, shown in Figure 8. The lookUp function has local scope, as it looks up the path p in the local partition of the distributed tree (returning its associated info). The addLeaf (resp. addNode) function has global scope; it creates a leaf (resp. a node) somewhere in the distributed tree; this leaf is associated with info i.

[12]In this paper, for the sake of brevity, we describe only one aspect of the SOS Name Service. The actual Name Service is more advanced as it is layered, user-centered and configurable on a per-user basis. The SOS Name Space, described here, corresponds to its bottom layer.

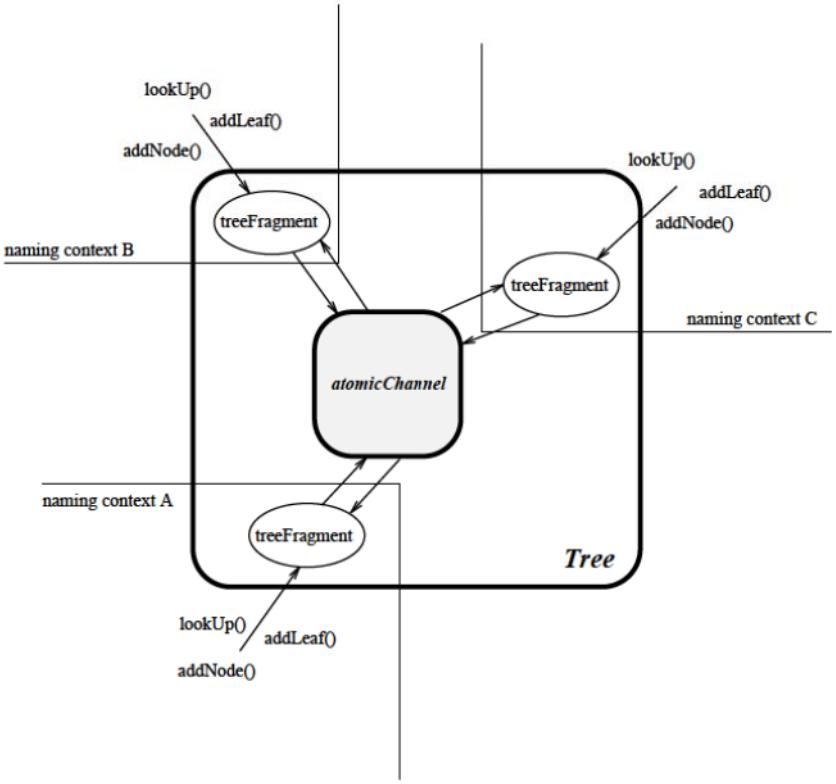

Figure 7: The Tree FO.

```
class treeFragment
{
public:
        lookUp (in path p, out info i);
        addLeaf(in path p, in info i);
        addNode(in path p, in info i);
};
```

Figure 8: Client interface of the Tree FO.

A treeFragment object also implements a group interface, consisting of two kinds of functions. First, there are functions corresponding to those of the public interface that have a global scope, namely groupAddNode and groupAddLeaf. Second, there are groupVerify, groupLock and groupUnLock to ensure consistency of the distributed tree despite concurrent requests addressed to different treeFragments.

For instance, a addNode request is processed as follows. First, groupLock locks the father node of p in all treeFragment partitions managing the father. Then, groupVerify checks that the path doesn't already exist in any partition; groupAddNode adds the node in the selected part. Finally, groupUnLock unlocks the father node.

Connective Object

The Tree fragments are connected by a communication channel of type atomicChannel. The groupLock and groupUnLock functions use a protocol on the atomicChannel, ensuring that all requests are delivered to all fragments in the same order. The groupVerify, groupAddNode and groupAddLeaf functions use a cheaper protocol (on the same atomicChannel) ensuring that all requests are delivered to all fragments (but not necessarily in the same order).

4.2 The NameService FO

The NameService FO is composed of two types of fragments: nsServer and nsProxy (see Figure 9). The corresponding declaration in the FOG language is described in Figure 10.

The role of a nsServer fragment is to resolve symbolic names and to map the Name Service requests onto operations on Tree.

A nsProxy fragment represents the Name Service for a particular client. Each nsProxy fragment manages a local cache of name prefixes; each entry of this cache associates a name prefix with the group of nsServer fragments handling the names with this same prefix.

Interfaces

A client of the SOS Name Service uses the public interface of the associated nsProxy fragment to create a directory (createDir), to associate a symbolic name to an object (addName), and to look up a symbolic name (lookUp).

Each of these functions performs the following operations. First, it searches for the longest prefix matching the argument of the request in the local cache;

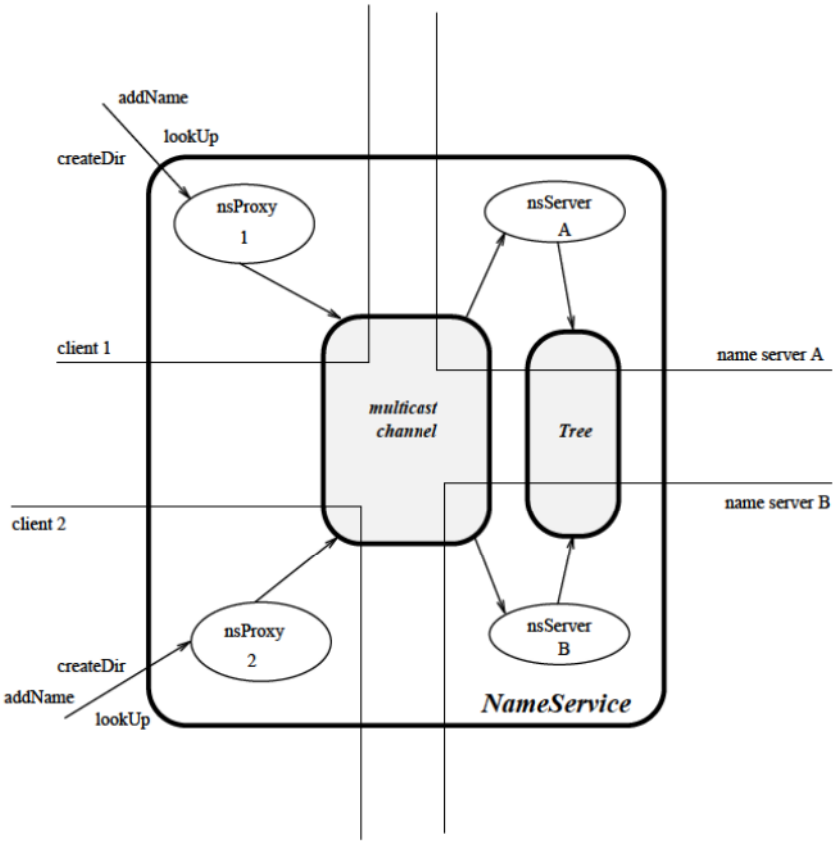

Figure 9: The NameService FO.

```
// The class group NameService FO type
group NameService { nsProxy, nsServer;}

// The fragment classes
class nsServer
{
 group:                                          // group interface
      groupLookUp (in String name, out reference obj);
      groupAddName (in String name, in reference obj);
      groupCreateDir (in String name);
};


class nsProxy
{
public:                                          // public interface
      lookUp (in String name, out reference obj);
      addName (in String name, in reference obj);
      createDir (in String name);
private:
      cache tableOfPrefixes;                     // the prefix cache
      multiChannel mchan;                        // a multicast channel

      // multicast to a group of servers
      forwardLookUp (in String name, future out reference obj[],
                     in reference callees)
         ! mchan.prpc (callees) ! nsServer::groupLookUp (name, nr);

      // functional rpc towards a group of servers
      forwardAddName (in String name, in reference obj, in reference callees)
         ! mchan.rpc (callees) ! nsServer::groupAddName (name, obj);
      // functional rpc towards a group of servers
      forwardCreateDir (in String name, in reference callees)
         ! mchan.rpc (callees) ! nsServer::groupCreateDir (name);
};
```

Figure 10: Declaration of the NameService FO.

this step determines the reference of the group of nsServer fragments expected to handle the subtree containing this name. Second, these nsServer fragments are invoked to continue the processing of the client request via their corresponding group interface: groupAddName, groupCreateDir and groupLookUp functions.

Connective Objects

As mentioned earlier, the NameService uses two connective objects: multicastChannel and Tree.

The multicastChannel (taken from the toolkit) connects each nsProxy to all nsServer fragments. It provides the invocation protocols that the nsProxy fragments need to invoke their nsServer counterparts. As stated earlier, binding of nsServer to Tree uses "local binding", i.e., static configuration data.

The second connective object, Tree, encapsulates the SOS distributed tree of names as explained in Section 4.1. It is shared by all nsServer fragments. Unlike multicastChannel, this connective object type is not offered by the toolkit of low-level FOs; it is programmer-defined.

Binding

Each nsServer binds locally (i.e., using static configuration data) to the two connective objects: the Tree and the multicastChannel.

A client binds to the NameService via a NS-specific provider. The export method of nsServer (acting as a provider) instantiates a nsProxy, initializes it, establishes connections with all nsServer fragments, and returns it.

Currently, only one single implementation of the nsProxy interface is supported; we plan several in the future (e.g., different caching policies). The provider will decide which one should be exported to any specific client

5 Discussion

In this section we discuss the advantages of our approach, which we then compare to related work.

5.1 Benefits of the FO Approach

The FO approach extends object-oriented programming to the distributed case. It retains the well-known advantages of object orientation, while addi-

tionally taking into account features specific to distribution: e.g., interacting between separate, independently-designed applications; existence of separate address spaces; consideration of geographical distribution of data and function; and interworking of different layers of protocol.

We now list the benefits we claim for FO programming, based on specific examples. First, the benefits of object-oriented programming, especially in a distributed setting: separation of interface from implementation; safer programming; and re-use.

1. *Separation of interface from implementation*

   An interface specification is called a *type*. An object of some type may be replaced by another object of a compatible type. A type may have many implementations (*classes*). As a result:

   - The abstract view of an object is separate from the object designer's concrete view;
   - It is easy to switch between different *policies* for the same *mechanism*;
   - Components can evolve independently.

   To illustrate this point, consider the Name Service example. The Tree designer may decide to replicate certain partitions, or on the contrary to maintain a centralized image of the tree; these changes would not affect the NameService FO. Conversely, the Name Service designer could change the distribution of service between nsServer and nsProxy (e.g., to a different caching policy); such a change would affect only the cooperation between the clients and servers.

2. *Safer programming*

   An object is *encapsulated*, meaning that access is allowed only through its public interface. Strongly-typed language compilers such as C++ ensure that the use of an object is compatible with its declared type. Traditionally, however, such safety is checked only within a single address space.

   The FOG compiler extends these checks to separate address spaces. Type checks extend to checking (both statically and dynamically) that binding to an FO returns an interface conforming to what the client requested, and that its usage conforms to its type. Encapsulation checks extend to verify that the group interface of a fragment is accessible only to fragment classes of the same fragmented type, and that fragment instances only invoke other fragments of the same FO instance.

   In the Name Service example, the FOG compiler checks not only (like any standard C++ compiler) that clients access NameService and Tree

via their public interface only, and that the actual arguments conform to declaration, but it also checks that the fragments of two separate Name Service instances do not interfere, and that only Name Service fragments do access e.g., groupLookUp.

3. *Re-use*

   The above characteristics encourage the development of libraries of generic types and classes, to be re-used in different applications. This is illustrated in our case by the toolkit of pre-defined FOs. The Tree FO could now be added to the toolkit, for re-use by other distributed applications manipulating a tree structure.

We now examine the answers to some specific problems of distributed programming addressed by the FO approach, by FOG and by our toolkit: support for different levels of transparency; separation of interfaces; high-level communication abstractions; and support for concept of layered protocols.

4. *Appropriate transparency for both clients and designers.*

   Distribution is transparent for clients. Yet the designer has full control of distribution of data items, their location, and all aspects of communication between them. For instance, the Tree fragments are connected by a channel of type atomicChannel, supporting atomic parallel invocations, while the NameService fragments are connected by a multicastChannel only. Each FO may use what it actually needs and no more; switching to a different policy is easy, just by replacing a connective object.

   It is sometimes useful to allow clients to control some aspect of distribution, by exporting an appropriate interface from the FO. For instance, in the actual implementation of Tree, each method has an extra partID parameter, controlling the location of directory data.

   Conversely, some FO designers would rather want network transparency *inside* their FO also. This is done easily, either using FOG as a pure stub generator (see Item 7), or by using a connective FO class that handles distribution internally, such as Tree.

5. *High-level communication abstractions*

   Communication occurs via shared fragmented objects. In most existing systems, only primitive, untyped communication objects are available, such as a send(bytestream)/receive(bytestream) channel. Our approach instead encourages the use of high-level communication abstractions. For instance Name Service nsProxys and nsServers communicate through a shared Tree. This is consistent with the object-oriented approach.

6. *Multiple layers of protocol.*

   Multiple levels of abstraction (layers of protocol) are present together in the same framework. In the case of the Name Service, these are NameService, Tree and the channel level (atomicChannel, multicastChannel).

   Coercions (marshalling/unmarshalling) between these multiple layers are handled automatically and in a type-safe manner.

7. *Separation of interfaces.*

   FOG separates the public interface of a FO from the internal (group) one, and supports both downcalls and upcalls ("!" notation). The compiler automatically generates any necessary forwarding between interfaces. Thus, it includes and supersedes the functionality of a traditional stub generator.

   The use of public vs. group interfaces is shown in the Name Service example: for instance, the public addName looks up data and sets locks before using the internal groupAddName; it would be dangerous to expose the latter in the public interface. Here forwarding is explicit.

5.2 Related Work

Several projects have proposed distributed extensions of the object paradigm: client/server-type remote object invocation as in ANSA, shared objects as in Orca or Comandos, or models more closely related to our FOs (Gothic, Topologies). In this section, we present them and point out some of their limitations.

5.2.1 The Client/Server Model

The better-known model is the client/server remote invocation, where the server is seen as a remote object accessed via a stub at the client's location.

Thus ANSA [1] supports remote object invocation. An ANSA client interfaces to a distributed service using a stub, a placeholder object for a remote server, automatically generated from an interface description. It has no useful local data or processing capability; its only function is to marshal/unmarshal arguments and forward invocations to the server.

A stub remotely extends access to the server object, but is not as flexible as a fragmented object. Stub generation is one of the functions of the FOG compiler; FOs are a a superset of the client/server model.

5.2.2 The Shared Object Model

Orca and Comandos support distributed shared objects, giving the illusion of a single global object space.

In Orca [2], shared objects are dynamically replicated under the control of the Orca run-time. All the replicas of a shared object form a single object. Orca ensures transparent access to the replicas and executes a consistency-preservation protocol among them. The decision to create new replicas, or to migrate the object on to the sites where the object is frequently used, is made automatically by the Orca run-time, based on statistics of recent access.

The Comandos [12] 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.

Orca and Comandos hide distribution to both the clients and designers of a shared object. While simple, this automatic approach has drawbacks. First, replication, migration, or diffusion of activities, are just specific sharing policies. Second, no single consistency policy is suitable for all applications.

In our model, the designer has a choice of levels of transparency and of policies. It is possible to choose an Orca-like automatic replication, by appropriate choice of connective objects. But is is also possible for knowledgeable designers to exercise more fine control, based on protection, efficiency or fault-tolerance criteria, deciding how and where to place data and function.

Other systems supporting a similar model are Emerald [10] and Amber [6].

5.2.3 Other FO Approaches

Gothic's fragmented objects [4] are based on "multi-functions" [3], a parallelized generalization of procedures to N callers and P callees. Although we use the same name, our FO model differs from Gothic in three ways. First, we focus on distributed, rather than parallel, computations. Second, we give the designer full control over the distribution, rather than only the automatic mechanisms provided by the Gothic system. Third, we support multiple client interfaces, whereas Gothic enforces a single global interface to a fragmented object.

Topologies [13] 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 primitive communication objects of our FO toolkit. The reference 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, whereas we allow high-level, programmer-defined interfaces.

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 listed some tools available to the fragmented object programmer: a library of predefined fragmented objects, and the FOG language and its compiler. The design of FOG draws upon the experience of the SOS Distributed Object-Oriented System[13] and of other distributed applications written for SOS.

From our experience with SOS, we strongly believe that, the FO approach is well suited for structuring many different kinds of distributed abstractions such as synchronization abstractions, distributed shared memory, cache and replication managers, and protocols. The toolkit of predefined FO types implementing these distributed abstractions is of a primary interest in encouraging programmers to write distributed applications. It will progressively accumulate abstractions needed by a large number of applications.

Acknowledgment

We are grateful to Willy Zwaenepoel, from Rice University, currently visiting the SOR group at Inria, for his many comments on the draft of this paper.

References

[1] Architecture Projects Management Limited. An engineer's introduction to the architecture. Technical Report TR.03.02, ANSA, Cambridge (United Kingdom), March 1989.

[2] 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.

[13] The SOS Distributed Object Manager, the SOS Naming Service, and communication protocols are structured as FOs.

[3] 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.

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

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

[6] 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.

[7] 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.

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

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

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

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

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

[14] 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.

[15] 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.

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