SOS: An Object-Oriented Operating System – Assessment and Perspectives

Marc Shapiro, Yvon Gourhant,
Sabine Habert, Laurence Mosseri,
Michel Ruffin, and Céline Valot
Institut National de Recherche en
Informatique et en Automatique

ABSTRACT: SOS (SOMIW Operating System) is the result of a four-year effort at INRIA to define an object-oriented operating system. SOS provides support for arbitrary, user-defined, typed objects. The system implements object migration; this mechanism is generic, but can be tailored to specific object semantics thanks to the prerequisite and upcall concepts. SOS also supports Fragmented Objects (FOs), i.e. objects the representation of which spreads across multiple address spaces. Fragments of a single FO are objects that enjoy mutual communication privileges. A fragment acts as a proxy, i.e. a local interface to the FO.

All the other mechanisms of SOS are built upon these basic abstractions. Thanks to prerequisites,

This is a revised and expanded version of "Prototyping a distributed object-oriented OS on UNIX," Marc Shapiro, in: *USENIX Workshop on Experience with Building Distributed and Multiprocessor Systems*, Ft. Lauderdale, FL (USA), Oct. 1989. This work was financed in part by the Commission of the European Communities under Esprit Contract 367 SOMIW.

migration of data may cause the migration and dynamic type-checking and linking of the corresponding code. A distributed object manager, an object storage service, a naming service, as well as a protocol toolbox and some applications, have been built as FOs.

This paper gives a detailed account of the architecture and design decisions of the SOS prototype on UNIX. We examine both good decisions and problems. The basic good decision is our simple object model, and its ability to map user-defined semantics (policy decisions) on system-implemented mechanisms. The most important problem is the dynamic nature of Fragmented Objects, and inadequate support for them.

1. Introduction

Object-oriented programming methodology is becoming increasingly popular for all sorts of applications. Many object-oriented programming languages exist, such as Smalltalk [Goldberg & Robson 1983], Eiffel [Meyer 1987], C++ [Stroustrup 1985], CLOS [Gabriel 1989], etc. Each compiler enforces its own object model, and deals with the inadequacies of existing operating systems in its own way.

The main goal of the SOR (French acronym for Distributed Object-Oriented Systems) group of INRIA is to implement an object-oriented distributed system which offers an object management support layer common to all applications and languages. This should:

- offer a more simple universe for the development of applications;
- facilitate the implementation of object-oriented language compilers;

- make applications more efficient;
- allow independent applications to communicate and share objects, without prior arrangement.

The services of the common object management support layer include support for creating, deleting, migrating, storing, localizing, and invoking objects. If these services are sufficiently complete, low-level, generic, language-independent, application-independent, and efficient, then they can legitimately be called an *object-oriented operating system*.

Within the office-workstation Esprit project SOMIW (Secure Open Multimedia Integrated Workstation), from 1985 to 1988, we have built a prototype called SOS. It has been used for the SOMIW applications, such as BFIR2, a multimedia document toolbox, and Images, an user-interface management system. SOS is written in C++ and prototyped on top of UNIX.

SOS supports an elementary object model which is both simple and powerful. A reasonable granularity is of the order of a hundred bytes and up per object. Composite objects are built on top of elementary object mechanisms.

SOS is designed to encourage the use of the *proxy principle* [Shapiro 1986] for structuring distributed applications. It extends the object concept to distributed, or "fragmented" objects. Externally, a fragmented object appears to be a single object. Its interface is provided by its local fragments or proxies, which are elementary objects. Internally, the many fragments are distributed.

SOS is built using its own mechanisms: all the SOS system services are implemented as fragmented objects with local proxy interfaces.

This paper presents SOS, the decisions we made in designing it, and an assessment of the prototype.

The next section provides an overview of the main concepts of SOS. Section 3 is about elementary objects. It is followed by section 4, an explanation of fragmented objects. Follows section 5, which discusses object migration. Sections 6, 7, 8, 9, describe the main services of SOS: communication, dependencies, persistence and naming. Finally, in section 10, we give an assessment of the design and implementation of the prototype.

2. Overview of SOS

SOS is an object-oriented operating system. It provides support for arbitrary user-defined objects, including object creation, destruction, migration, storage, localization, communication, naming, support for Fragmented Objects, etc.

2.1 SOS concepts

An *Elementary Object* is some user-defined set of data and code. Elementary Objects don't need to be known by the system, if they are not intended to be migrated, stored or remotely accessed. For such objects, which we call *plain objects*, the system doesn't keep any information. In this paper we are only interested in the objects managed by SOS which we call *SOS objects*; we will consider that all Elementary objects are SOS objects.

Elementary Objects have a system descriptor.[1] Considering the overhead due to the descriptor existence and its management, a reasonable granularity for the object data part is a size of 50 or 100 bytes and up.

We assume that the data part is accessed only via its type-checked procedural interface. An object accesses system services by calling the appropriate primitives; we call this a *downcall*. Conversely, the system can invoke, with an *upcall*, a few well-known procedures of an object.

An object is mapped into a context. A *context* is an address space. It may contain any number of Elementary Objects. Elementary Objects may migrate between contexts; at any point in time, an Elementary Object is active within a single context, or stored on disk.

Each object has an unique identifier called its *concrete OID*. An object is designated by its address (within the context), or globally by a *reference* containing an OID and a location hint.

SOS extends the object concept to distributed or *Fragmented Objects*. A Fragmented Object is implemented as a group of Elementary Objects which can be located in several contexts, on

1. Composite objects, with multiple data segments connected by pointers, are built on top of Elementary Objects, but will not be considered here.

different sites; its representation is the reunion of the local "fragments." Just as an Elementary Object can access its own representation, bypassing the procedural interface, similarly the individual fragments of a Fragmented Object are allowed to use untyped communication to each other: invocation of fragments belonging to different contexts, called *cross-context invocation*, shared memory, etc. Objects which are not fragments of a same Fragmented Object are not permitted to communicate in this manner.

A fragment may create and add a new fragment to the group, and export it to another context. Group membership is preserved across migration; thus the Fragmented Object grows by spreading.

Applications on SOS are designed according to the "proxy principle" [Shapiro 1986]: services are composed of three kinds of Elementary Objects: servers, proxies and providers. The *server* is an object which is able to serve requests. The *proxy* is a local Elementary Object, which represents the service. Each client which wants to access a service, must have a proxy of this service in its context. The *provider* is in charge of providing proxies on client request. For clients, the proxy is the only interface to the service. A proxy can process requests locally, or forward them to the remote server (see figure 1).

The proxy principle is a powerful and flexible tool to structure distributed applications. The use of proxies allows to enforce security in the system and ensure a large location transparency.

2.2 The SOS prototype

Our prototype is implemented in C++ on top of UNIX (SunOS). This article describes SOS Prototype Version 5.

SOS comprises a kernel and system services running on top of it. The kernel provides separate address spaces (contexts), light-weight threads in a context (tasks), and inter-context communication. Programming for SOS requires the use of predefined libraries and a modified C++ compiler.

SOS objects are instances of the predefined class `sosObject` (or of a compatible class). C++ has so-called virtual procedures [Gautron & Shapiro 1987]. A class may override the pre-defined

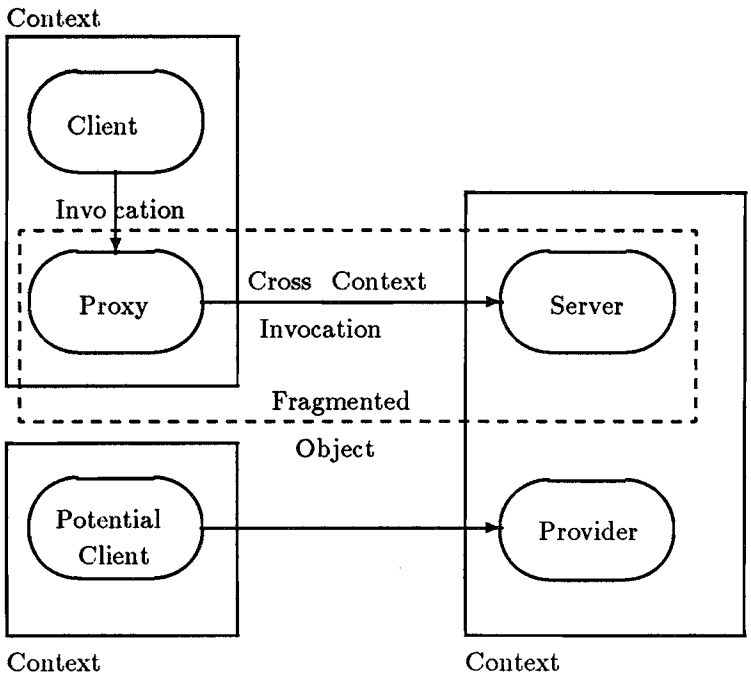

Figure 1: The proxy principle:
A service implemented as a fragmented object

actions of these procedures. Upcalls are performed by calling `sosObject`'s virtual procedures. Unfortunately, this is not language-independent.

Separate address spaces are provided by UNIX. Tasks are implemented as a library (the task library of C++ [Stroustrup & Shopiro 1987] with some additions). Context management is performed by a UNIX process called `sos`. Inter-context communication uses UNIX-domain stream sockets. Remote communication uses the SOS protocol toolbox [Makpangou 1989].

On each machine and for each system service, the program `sos` automatically starts system-service contexts set. Applications are run from the UNIX shell or the debugger.

The system services are structured as Fragmented Objects. Four basic system services are available:

The Acquaintance Service. This is the distributed object manager. It deals with localization, migration of objects, in cooperation with the communication and storage services. Each

context has an Acquaintance Service proxy at instantiation which yields to the context the basic operations on Elementary Objects (see section 3).

The Communication Service. The Communication Service provides communication between sites, and a set of invocation protocols allowing remote procedure calls and multicast (see section 6).

The Storage Service. The Storage Service handles the generic aspects of object persistence. It defines a minimal set of simple and generic tools to make storage of typed composite objects handled quasi-automatically by the system (see section 8).

The Name Service. The name service manages the binding of internal names to symbolic names. Any object can be named in the same manner. The name service allows clients to build their own view of the name space (see section 9).

After a comparison with similar work, we will take a deeper look at the design and implementation of the SOS prototype, and evaluate it in the light of our experience. For more detailed information about SOS, see SOR [1989] and SOR [1988].

2.3 Comparison with similar work

Emerald is an object-oriented language for distributed programming, featuring fine-grained mobility [Jul et al. 1988]. The compiler transforms the user-defined object representation in order to facilitate migration: its first few bytes are a standard descriptor, and all fields of a similar type are grouped together. Conceptually, all objects live in a single, network-wide address space. An object reference is global, but a local reference is optimized into a pointer.

In contrast, the SOS approach is operating-system based. We do not assume any standard representation. Instead, system information is well separated from programmer-defined data, and the system performs upcalls on objects. Instead of a single address space, we stress *structuring* the universe.

Choices [Campbell et al. 1989] is a family of operating systems built using object-oriented design. The services it exports to applications are fairly conventional. The emphasis in SOS was not its internal design, but providing new services to facilitate the implementation of distributed object-based applications.

Clouds [McKendry 1985] is another object-oriented OS. Its emphasis is on integrating support for reliable objects in the low level of the system (Our current design has no particular provisions for reliability.). Their objects are presumably much larger-grained than ours, since a Clouds object executes in its own address space.

Guide/Comandos [Decouchant et al. 1989] is a language-driven distributed programming environment. The universe is structured in separate, multi-machine address spaces called domains. When a domain needs access to an object located on a remote machine, it extends itself to that machine, and maps the object in. This structure is easier to use than SOS's Fragmented Objects; however the latter scales better, and deals better with replication.

Gothic [Banâtre et al. 1986], a language and system for reliable distributed programs, is based on a theory of Fragmented Objects invoked via "multi-functions" (side-effect-free invocations, with co-ordinated distributed threads), supported by the language. Our Fragmented Objects are ad-hoc but more flexible.

3. *Elementary Objects*

The basic entity managed by the SOS Acquaintance Service (AS) (i.e. the object manager) is the *Elementary Object*. We have made the Elementary Object as simple as possible, a "least common denominator" for all uses.

An Elementary Object is a single user-defined data segment, with a system descriptor. (Composite objects, with multiple, arbitrary data segments connected by "permanent pointers" are built on top of Elementary Objects; we deal with them in section 8.)

At any point in time, an Elementary Object exists in a single context on a single machine. Each Elementary Object is different from all others; it is characterized by its own unique identifier

called its *concrete OID*. An Elementary Object is known to SOS by its descriptor, called Acquaintance Descriptor (AD). There is a table of ADs per context, managed by the context's AS proxy.

An AD for some object contains the following information (the items in italics have to do with migration and groups, and will be defined later):

- Its concrete OID and (possibly) a list of *group OID*'s,
- The reference of its `code` object and (possibly) a list of *prerequisites*,
- The address and size of its direct segment,
- (Possibly) Its list of *channels*.

The class `code` is a predefined class of Elementary Objects. A code instance holds the compiled code for some class. For instance the code for some user-defined class `X` is managed by the code instance `code_for_X`. The reference from the AD to the object's code is necessary for migration. Modeling the code as a separate object is an example of uniformity and reuse: the system treats a `code` instance just like any other object.

The following table gives the downcall interface for elementary-object management. There are no upcalls.

In this and the following tables, we use a pseudocode notation for simplicity. The clause "`a.b(c)→d`" means: invoke procedure `b` of object `a`, with argument `c`, and returning value `d`. Object `AS` is the proxy of the Acquaintance Service.

`new sosObject () → obj`	Object creation
`obj . delete ()`	Object destruction
`obj . setCodeRef (ref)`	Set code reference of object
`AS . find (ref1, radius) → ref2`	Search for object location
`AS . getAddress (ref) → obj`	Translate global reference to local address
`AS . getReference (obj, OID) → ref`	Translate address to global reference

Downcalls for elementary-object management

3.1 Creation and destruction of Elementary Objects

In C++, creating an object triggers a chain of calls to *constructor* procedures, starting from the actual implementation class, up to the root of the inheritance tree (in this case, `sosObject`), and back down to the implementation class. A constructor is a mix of compiler-generated and user-defined code. Memory for the object is allocated (by `malloc`) in the compiler-generated part of the implementation class constructor; its address passed up, as the object's address, to the `sosObject` constructor.

Thus, there is no explicit primitive for object creation: it is implicit in the `sosObject` constructor, which allocates an unused AD, and fills it with a newly-allocated OID, and with the address and size of the data.

The size of the data is not explicitly available to the `sosObject` constructor: it is taken from the `malloc` header. The reference to the code object is not available either; the constructor sets it to nil. The other parts of the AD are also initialized to nil.

The implicit AS interface for object deletion is the *destructor* procedure of `sosObject`, called automatically when an instance is deleted, similarly to the constructor. The destruction of a context or a processor crash deletes all the contained instances, but the destructor may not be called. We have designed a new mechanism to reliably propagate object-destruction events to dependents of an object (see section 7).

3.2 Miscellaneous Elementary Object management procedures

The `find` procedure of the AS, given a reference to an object, finds the actual location of the object (possibly by asking all the AS proxies within the specified `radius`), and returns a reference containing that exact location. If the argument is a reference to a group (see section 4), the returned value is the reference of its closest fragment.

`AS.getReference` and `AS.getAddress` translate between local object addresses and global references. If `getAddress` is passed a reference of a group (see section 4), it returns the address of its

local proxy, if any. The `OID` argument of `getReference` allows to pick between a reference to the Elementary Object itself, or to its group.

3.3 Elementary Objects: assessment

3.3.1 Code objects

The code of an object is modeled as a separate object, an instance of class `code`, attached to its Elementary Object by a call to `setCodeRef`. The system treats it just like any other object. This is an example of the power of our elementary-object model. As we will see (in section 5.3), dynamic type-checking and linking is performed automatically. They are not wired into the system, but are performed by the class `code` reinitializer. This is an example of uniformity of treatment and reuse of generic functionality at the service of specialized semantics.

3.3.2 Creation and destruction

An object is created either as a plain object or an SOS object, and remains such for all its existence. A useful degree of flexibility would be to allow an existing object to become known to the system dynamically.

The object-creation scheme outlined in section 3.1 is convenient for the C++ application programmer, because the compiler automatically takes care of calling the object creation and destruction primitives.

One drawback is that the system interface is not clearly identified and not language-independent. Another problem is that the `sosObject` constructor does not have all the necessary information; for instance the size of the data is obtained by the `malloc`-header hack. Similarly, the `code` reference can not be set by the constructor; a separate call to `setCodeRef` is necessary prior to migration.

The convenience of automatic creation should be kept, but more information is needed from the compiler to make it effective. Furthermore, convenience for C++ programmers is no excuse for not defining a language-independent interface.

4. *Fragmented Objects, or groups*

A Fragmented Object is the structure for a distributed application. The purpose of a Fragmented Object is distributed communication and cooperation, without interference from other objects.

It is implemented as a *group* of Elementary Objects, called its fragments. Alternatively, a group can be viewed as is a single object with a fragmented representation. Clients of the group may access it locally by its strongly-typed procedural interface, provided by the proxy fragments. The group's public interface is a sort of a union of its fragments' interfaces.

Just as an Elementary Object may access its own representation directly, bypassing its public interface, similarly a fragment may access the group's internal representation. Therefore fragments may communicate, via untyped shared memory or by messages for instance.

A group is conceptually a protection domain, entered by invoking one of its local proxies.

The following table shows the interfaces for group management.

`AS . addGroupOID (obj, OID) → index`	Create a new group
`obj1 . giveMyOID (obj2, index1)`	Put `obj2` in same group as `obj1`
new `channel (obj1, obj0, opaque) → ch1`	Establish channel from `obj1` to `obj0`
new `channel (obj2, ch1, opaque) → ch2`	Duplicate channel
`ch . invoke (callMsg) → replyMsg`	Remote invocation
`ch . send (callMsg) → inv`	Asynchronous remote invocation

Group downcall interface

`obj . stub (callMsg) → replyMsg`	Invoke object, return reply
`ch . receiveInvoke (callMsg) → replyMsg`	Channel receives invocation

Group upcall interface

A group is characterized by the fact that each fragment carries the OID of the group, in addition to its concrete OID. An Elementary Object can be a fragment of zero, one, or more groups.

Members of a group enjoy mutual communication privileges, which are denied to non-fragments. An invocation *channel* is a unidirectional RPC connection between two Elementary Objects on the same machine, materialized by a field in the source object's AD pointing to the target. (Channels to remote machines, and channels implementing other protocols, are explained in section 6.)

Other types of communication within the group, such as shared files, are also available, but will not be detailed here. Shared memory should be possible, but we never implemented the appropriate interfaces.

4.1 Group management

The primitive `addGroupOID` assigns a fresh group OID to an object, in order to start a new group. It returns the index of the new OID in the object's AD's identifier list.

A group is created implicitly by `giveMyOID`, which gives away an existing concrete or group OID (designated by its index in the list of OID's of `obj1`), to some object.

A group disappears when its last fragment goes away.

4.2 Channels

A channel may be created only between fragments of a same group. The first channel creation procedure creates a channel between its argument objects, which must be located in the same context. Objects connected by a channel can be migrated. The second channel creation procedure duplicates an existing channel: before the call, `obj1` has a channel `ch1` to some fragment, say `obj0`; after the call, `obj2` also has a channel `ch2` to the same receiver. This is the only way to create a channel to an object in another context.

As its name implies, the `opaque` argument is not interpreted by the system. It is simply stored at the sender end of the channel, and will be automatically prepended to every invocation sent on

it. The receiver may test the opaque field of remote invocations to distinguish between its callers, for instance to establish their access rights.[2]

4.3 Using channels for cross-context invocation

The channel operation invoke sends an invocation on a fragment's channel to some other fragment, possibly located in some other context, and returns a reply. The send operation is similar, but executes asynchronously; it returns an invocation object which can be queried for results. Discussion of invocation objects is deferred to section 6.1.2.

Remote invocation arguments must be "marshalled" [Birrell & Nelson 1984] by the calling proxy, and passed to invoke or send as a message. A message is composed of a header and of a list of segment access rights. The message header is of limited size (1024 bytes); any larger data is to be stored in a segment and passed as a segment right.[3] Available rights are read, write, and create. A reply is also made of a limited-size message and, possibly, segment rights.

A cross-invocation causes an upcall to the stub procedure of the receiver within a fresh task. The receiver gets a copy of the invocation message, and may access the segments according to the rights passed. Stub returns a return message which is copied back to the caller.

It is up to stub to unmarshall the call message and segments, call the appropriate procedure, and marshal its results into the reply message and segments. It plays the rôle of Nelson's "server stub" [Birrell & Nelson 1984].

A channels' receiveInvoke procedure may perform some local processing of the message at reception. It is the counterpart of invoke, for end-to-end protocols. The receiveInvoke of a receiving channel is called before the stub of the receiver object.

2. The opaque attribute of a channel is similar to the rights field of a capability in Amoeba [Mullender & Tanenbaum 1986].
3. This is modeled after the V-System RPC [Cheriton 1984].

4.4 Fragmented Objects: assessment

4.4.1 Cross-context communication

This is our second design for the cross-context communication interface. Our previous one (see for instance [Shapiro 1989]) was much more primitive and exposed the kernel data structures. The new design uses `channel` and `invocation` objects to smooth the kernel interfaces. The use of objects allows us to overload the basic `invoke` and `send`, for instance for multicast protocols (see section 6).

4.4.2 Constructing a group

Currently each fragment type is programmed "by hand," and there is no guarantee of consistency even within a particular type of group. We are working on a new tool, a "fragment generator" (similar to an RPC stub generator). It will take care of the common aspects of programming fragments and providers (viz. allocating group OID's, setting up channels, and message marshalling and unmarshalling). It will also allow to define group types with a well-defined structure, and enforce their internal consistency at compile time. Finally, it will provide help in coordinating state changes between fragments.

4.4.3 Protection

Only the currently-executing Elementary Object should have access to its own channels; similarly for other primitive operations. The kernel attempts to enforce this, taking advantage of the fact that the C++ compiler adds the address of the invoked object as a hidden first argument to all invocations. When `invoke` or `send` is executed, the kernel checks that the sender identification, in the channel data, is equal to the current object argument of the penultimate stack frame.

The situation is similar with many system primitives.

Getting the current Elementary Object from the stack is a weak way of enforcing the group protection domain at run-time. Given our environment (vanilla UNIX and standard hardware), and the granularity of objects, it was unfeasible to implement a stronger form of run-time protection at a reasonable cost. A

stronger enforcement would be desirable, but weak enforcement is acceptable, because groups are intended as a program structuring concept, not a confidentiality mechanism.

To provide some protection of the group against spurious membership, `giveMyOID` and channel creation can only connect objects within the same context. The normal way of creating a group is to create proxies locally and migrate them (see below) to another context; group membership and channels are preserved across migration.

4.4.4 Non-uniform object identity

An Elementary Object has two different identities: its address; and a location-independent *reference* (containing its OID). An address is not meaningful outside of its instantiation context. It needs to be explicitly translated into a reference (by `getReference`) in order (for instance) to be embedded in a message.

A client of some service need not be aware of these distinct identities, as the service is accessed using the address of the local proxy.

In contrast, the programmer of a Fragmented Object must cope with them. Within the Fragmented Object, some Elementary Objects are local to each other, and others are in remote contexts. Communication between the former use addresses and local invocation; between the latter, references and cross-context invocation.

SOS offers tools to bridge this gap. Permanent pointers (section 8) automatically convert between references and addresses. Channel objects help hide the distinction between different modes of invocation. Finally, dependencies (section 7) replace explicit invocation by a more uniform mechanism for broadcasting state changes.

Some other systems, such as Emerald [Jul et al. 1988] and Amber [Chase et al. 1989] provide a more uniform view. In these systems, the local-address / global-reference distinction exists but it is hidden to programs, by providing a single, network-wide address space, and compiler support for trapping remote access. Similarly, Guide [Balter et al. 1987] supports multiple network-wide address spaces. An object is identified only by its global

reference. An object used by many applications is simultaneously mapped into each corresponding address space.

These systems are more tightly integrated than SOS, but require a close coupling between compilers and the run-time system. They are often restricted to a single, specially-designed language.

4.4.5 Future work

The problems of run-time object protection and non-uniform object identity are two symptoms of inadequate memory organization.

Proposals for the evolution of SOS include merging references and permanent pointers.

A structured memory organization, like that offered by capability machines, might improve run-time protection. A more attractive idea is to enforce the integrity of the group at compile time. Our proposed "fragment generator" (section 4.4.2) is a step in this direction.

An intriguing alternative would be to implement a Fragmented Object as a network-wide virtual address space. This should simplify the programming of Fragmented Objects considerably.

5. Migration of Elementary Objects

A Fragmented Object implements some distributed service. Its public interface is provided locally by its fragments, which act as *proxies* for the service. In order to get access to a service, a client must acquire an appropriate proxy. This is done dynamically by *migrating* a fragment into the client's context.

Migration is completely generic, thanks to appropriate upcalls (an upcall to `giveProxy` initializes an importation; an upcall to a *reinitializer* finalizes migration), and to the `code` and prerequisite objects.

5.1 Migration interface

The migration interface is given in the following table.

`AS . import (key, importReq, "class", service) → obj`	Request import of `obj` of type `class`
`obj2 . ch . export (desc1)`	Export object 1 along channel of `obj2`
`obj . giveSelf () → desc`	Use "move" semantics for migration of `obj`
`obj . giveCopy () → desc`	Use "copy" semantics for migration of `obj`

Migration downcall interface

`obj . giveProxy (importReq) → desc`	Import request
`obj . re-init (...)`	Finalize migration

Migration upcall interface

There are two possibilities for migration: import and export. Both use the algorithm exposed below, in section 5.2: migrate descriptor and data, recursively import code and prerequisites, call reinitializer.

We will first detail export, the simpler of the two. In section 5.1.2 we detail import. Finally in section 5.1.3 we give a typical SOS example: importing a proxy for a screen window.

5.1.1 Exporting

The call `ch2.export (desc1)` migrates an object `obj1`, described by `desc1`, along the channel of `obj2` indicated by `ch2`. Either `obj1.giveSelf()` or `obj1.giveCopy()` is used to prepare `desc1`. Export uses the migration algorithm of section 5.2; using `giveSelf` implies "move" semantics whereas `giveCopy` implies "copy" semantics. The object on the other end of the channel will receive a special invocation message, signalling the arrival of an exported object.

5.1.2 *Importing*

To acquire a proxy for a new service, a client will request an import from a proxy *provider* for that service, via the Acquaintance Service call `import`, which upcalls the provider's `giveProxy`. This procedure prepares a local object, which is then migrated back to the caller.

An easy-to-use import interface is implemented by a C++ compiler extension. The following extended C++ construct:

```
new dynamic (service) class (importReq, ...) → obj
```

generates a call to `AS.import` followed by a call to the proxy's re-initializer. Its arguments are: `service`, the reference of an object which will be requested to provide a proxy for the service, and `importReq`, an import request message carrying untyped request parameters. The AS automatically adds to the import request the reference of the requestor. Possible extra arguments (indicated by the ellipsis) will be passed to the *re-initializer*.

The other arguments to `AS.import` are automatically generated by the compiler: `key` describes the expected type of the imported object; and "`class`" is the name of the class in the `new dynamic` declaration, which is used to select a default provider, but is not otherwise used.

The mechanics of importation are the following. The AS proxy of the requestor performs a `find` based on the `service` reference. This yields the location of the provider object (or of one of its fragments if the provider is a Fragmented Object). The AS proxy at that location then performs the `giveProxy` upcall on the provider, with a copy of the import request, carrying sufficient information to identify the requestor.

After verifying the request and the requestor's credentials in some service-specific way, the provider's `giveProxy` selects some object `M` to be migrated, and calls either `giveSelf` or `giveCopy`, as above, to prepare a description which it returns; alternatively, it may return an error indication. The object `M` could be the provider itself, or some other object of its context, or a stored object. In the latter case it must be of the same group. When `giveProxy` returns, `M` is migrated to the requestor, according to the algorithm of section 5.2.

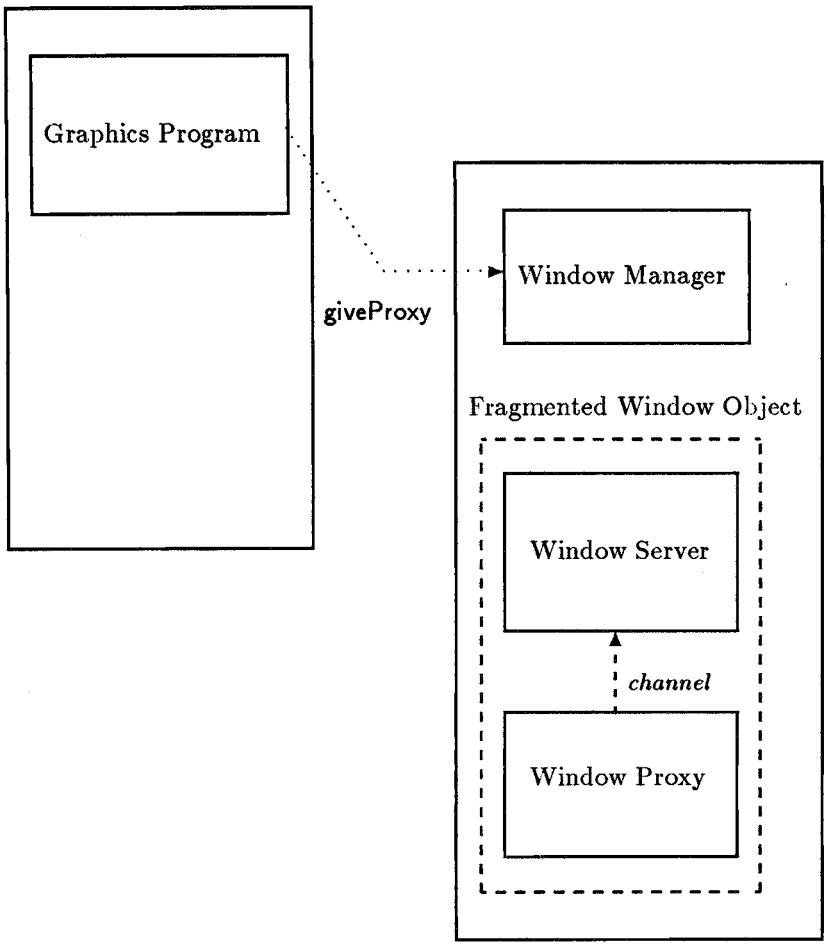

Figure 2: Before migration:
the Window Manager has created a Window Proxy and a Window Server and has connected them

At the end of a migration (step 5) a re-initialization procedure is up-called, to allow finalization. A typical use of the re-initializer is to set pointers to meaningful values, or to request more importations.

5.1.3 Importation example

Consider the example of a request to open a window on the screen. In SOS this will be an import request for a window proxy, addressed to the window manager. The window manager is the

proxy provider. It will create a window proxy P, which will be exported to the requestor as its window interface, and a window server S, which will do the graphics. This example is illustrated by figures 2 and 3.

The giveProxy code of the window manager will: put P and S into a newly-created group; connect them with a channel; and return P as a result.

The provider code will look some thing like the following.[4]

```
windowMgr :: giveProxy (importRequest) → desc {
    – check requestor's rights and arguments
    if (notOK (importRequest))
        then raise (refused)
        fi

    – create proxy and server, put them in a group
    P: windowProxy :=
        new windowProxy (importRequest.args)
    S: windowServer :=
        new windowServer (importRequest.args)
    i: integer := addGroupOID (P, new OID)
    P . giveMyOID (S, i)

    – prepare P for migration: set code, create channel
    P . setCodeRef
        ( NS . lookUp ("/export/windowProxy.code") )

    – suppose P has a field chan of type channel
    P . chan := new channel (P, S, "my proxy")

    – migrate P
    return (P . giveSelf())
}
```

5.2 Migration algorithm

Suppose Elementary Object X is to be migrated from source context S to destination context D. The algorithm starts when the decision to migrate X has been notified, and all access rights have been checked; we will ignore error cases. The algorithm is the following.

1. Make X unavailable to users in S.

4. The lookUp operation of the Name Service (see section 9) maps a symbolic name to a reference.

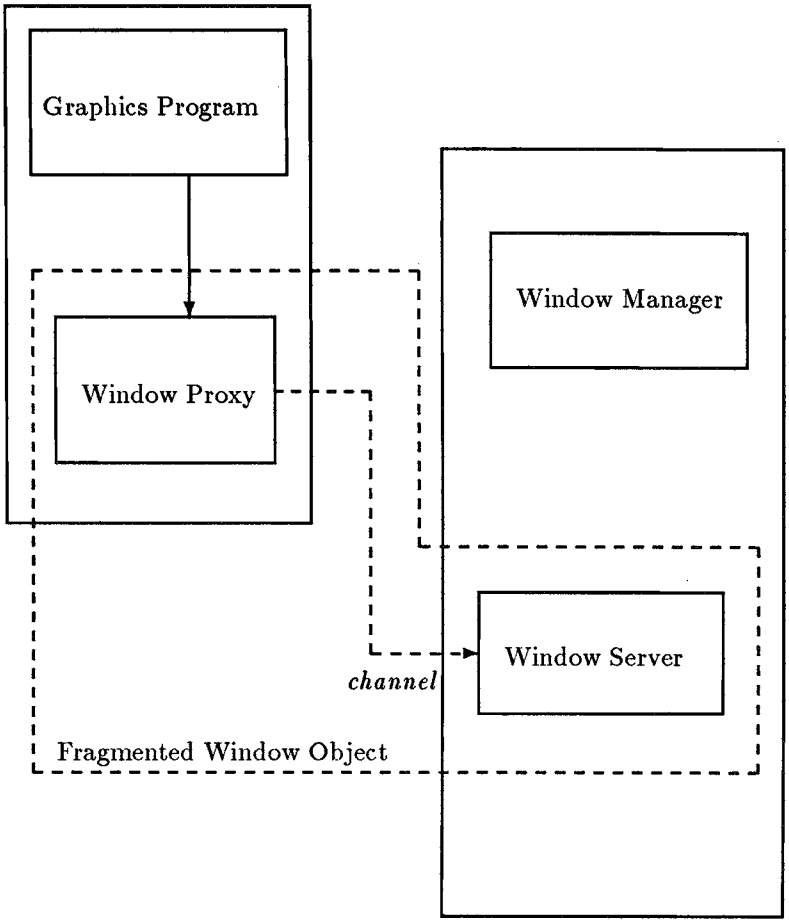

Figure 3: After migration:
the Window Proxy is in the client's context and remains connected to the Window Server

2. Copy the AD of `X` from source to destination context. All the contents of the AD are preserved: its concrete OID, group OID's, channels, code reference, prerequisite references, and size of data segment. However the address-of-data-segment field is invalid.
3. Using information in the AD, copy X's data from `S` to some arbitrary free location in `D`. Update the data address field in the destination AD.

4. If (a proxy of) X's code object is not yet present in D, import one. If present, skip this step. Similarly, import all prerequisites of X, if not already present.
5. Upcall the *re-initialization* procedure of X in D.
6. Make X available to users in D. The data and AD of X are destroyed in S.

The above describes the "move" variant of migration. The "copy" variant differs only slightly: a new concrete OID is allocated in D (step 2), and the source copy is not destroyed, but instead is made available again (step 6).

Group information and channels are preserved across migration. However the kernel only implements channels within the same machine. When an invocation is sent on a channel, if the kernel detects that the destination is remote, it cooperates with the Communication Service to recreate an appropriate indirection via a local protocol object. This is described in section 6.2.3.

5.3 Migration of code and prerequisites

We mentioned (step 4 of the migration algorithm of section 5.2) that the `code` and prerequisites are recursively imported, if not already present, before calling the re-initializer. The prerequisites are the environment the migrated object needs in order to function; the object's `code` is just one kind of prerequisite. These are imported if not already present, in order to avoid waste: this allows two imported objects to share code if they are implemented similarly. The same mechanism supports static linking of the code for proxies, without any loss of functionality.

The `giveProxy` procedure for class `code` migrates a copy of the code.

Since a prerequisite is imported according to the same algorithm as other objects, its re-initialization procedure is called in step 5. The re-initializer for a code object is a *dynamic linker and type-checker*. The type is checked against the `key` argument to `sosImport`. The linking and type-checking algorithm are language-specific; other languages could be supported simply by implementing a new `code` class.

5.4 Assessment

We stress that the upcalls to `giveProxy` and to the initializer, together with prerequisites, implement a very important concept: extending a system-defined mechanism with *programmer-defined semantics*. Arbitrary objects can be migrated, and the semantics of their migration is type-specific, above a single, generic, system-implemented mechanism.

The strength of this design is that prerequisites are Elementary Objects like any other. Dynamic linking and type-checking are automatic, without being wired in. The drawback is that type-checking is automatic only for the first import of an object of a certain class; type-checking for subsequent imports must be special-cased.

"Vertical" migration of data (from disk to memory) is essentially the same "horizontal" migration (between memory contexts). However vertical migration has the capability of dealing with composite objects (see section 8), which has not yet been integrated with the horizontal migration.

5.4.1 Calling the re-initializer

The C++ syntax for importation is an extension of the instantiation syntax, and in C++ the re-initializer is in fact a "virtual constructor."[5]

This raises the issue of whether the reinitializer call should be generated by the compiler, or performed by the system. The former solution permits extra arguments (in addition to the import request); the latter allows the system to know that re-initialization has succeeded. We opted for the compiler solution whenever possible, favoring the comfort of C++ programmers. However for exports and for prerequisite imports, the reinitializer can only be called by the system, hence the double interface.

The compiler decision was bad for three reasons. First, it treats exports and prerequisites differently from imports, which is confusing for the users. Second, the provider does not know if the import actually succeeded. Finally, and most importantly, an

5. We will not discuss this point in any detail since the language interface is out of the scope of this paper; see Shapiro et al. [1989] for details.

operating system should be independent of a particular language implementation; the system solution should be preferred.

5.4.2 Export vs. import

Exporting is a more primitive operation than importing. In fact, an import could be modeled as an import request, followed by an export from the provider to the requestor. Initially we refused to have an export primitive, because we were concerned with the protection issues involved, and we didn't know how let the target context make use of the newly-available object. Recently we realized that, for some applications, export is the only natural mechanism: for instance, the Images UIMS is modeled more naturally as a window manager exporting event objects to applications, rather than applications polling the window manager for events.

The export mechanism has been implemented, but the proposed interface is not yet available.

5.4.3 Static groups

A group is created when a proxy provider migrates a proxy to another context. But there is also a need for static groups. For instance, a system service such as the AS, the Name Service, the Storage Service, or the Communication Service, is implemented by one server on each machine, which comes up at boot time. In order to communicate with its remote peers, it must already be a fragment of their group as soon it starts up. Currently, a protection loophole is needed to circumvent this problem: when the server comes up, it forges an OID with a given value (taken from a configuration file) and inserts itself in the group using `addGroupOID`.

This loophole should be protected by some privilege but in fact it is not. Better still, the Fragmented Object should be *persistent*. SOS supports persistent objects, as a service above the basic mechanisms described here (see section 8); a much tighter integration is needed to support persistent groups.

6. Communication

The Communication Service [Makpangou 1988; Makpangou 1989] consists of a toolbox of protocols. Its encapsulation by *channel objects* (see section 4) should facilitate the work of adding new protocols and interfacing them with application programs. For example, the Acquaintance Service and the Name Service use multicast communications for distributed updates.

At present, this toolbox implements unicast and multicast, synchronous and asynchronous communication. The results of asynchronous and multicast communication are managed by an *invocation object*.

Multicast communication is based on the *family* concept. A family is a subset of a Fragmented Object. Its *members* communicate by multicast. A family comprises a set of members, operations to add and remove members, and multicast invocation interface to its members.

The Communication Service interface consists of two main layers:

The invocation protocols. For every invocation protocol type (such as RPC, multicast, synchronous or asynchronous), there is a corresponding class, which can be instantiated in the Communication Service context. The Communication Service is a toolbox of such classes. These base classes can be extended (by inheritance or redefinition) or mixed, to build new protocol types.

The application interface. The application interface consists of *channels*, for communicating between fragments. A channel encapsulates the access to protocol objects.

Two channel types are currently defined: `channel` for unicast communication, and `multiChannel` for multicast protocols. The second type is compatible with the first one.

6.1 Application interface

To use a protocol, an application instantiates a channel, which is a proxy of the needed protocol object. The protocol object is instantiated in the Communication Service context; The channel is connected to it by a channel.

The `channel` interface was given in the section 4.

6.1.1 Multicast communication interface

The `multiChannel` interface inherits from the `channel` interface. In addition, it offers the following procedures.

new `multiChannel (obj1, familyRef, opaque) → mch`	Create a channel from `obj1` to a multicast family
new `multiChannel (obj2, mch1, opaque) → mch2`	Duplicate multicast channel
`mch . multiInvoke (callMsg, replyDesc) (-> inv`	Synchronous multicast invocation
`mch . multiSend (callMsg) → inv`	Asynchronous multicast invocation

`multiChannel` downcall interface
In addition to `channel` *interface (see section 4)*

A `multiChannel` implements multicast communication. The reference of family (`familyRef`) must be provided at creation.

Invocations on `multiChannels` can be either unicast or multicast, and either synchronous or asynchronous. Callees see no difference between a unicast and a multicast invocation. With unicast invocation on a `multiChannel`, the protocol picks some arbitrary member of the family to be the callee; we call this *functional addressing*. A particular callee can be designated, by specifying its concrete OID in the invocation message; this is *selective addressing*.

With multi-invocation, all members of the family are invoked; this is *broadcast addressing*. Alternately, all the members on the site of a particular member can be invoked. This particular member is named by its concrete OID in the invocation message.

With synchronous multicast invocation, the caller waits until the number of replies received is equal to the number expected, as described by the descriptor `replyDesc`.

A multi-invocation creates an *invocation object*, which can be queried for incremental results.

6.1.2 Invocation objects

The following table shows the interface of an invocation object.

`inv . isReplyReady ()` `→ boolean`	Ask for possible reply
`inv . getNextReply ()` `→ replyMsg`	Wait for the next available reply
`inv . delete ()`	Terminate the current invocation, discarding any further results

Invocation object downcall interface

The `isReplyReady` method queries the invocation object for reply availability. The `getNextReply` method waits until the next reply is available and returns it. This allows several behaviors. For example, an application can decide to keep only certain replies, or only the first one.

An invocation object offers also the `delete` method to terminate the current call.

6.2 Internal structure

The only communication protocol implemented by the kernel is a cross-context invocation along a channel, within the same machine. Remote (across machines) access and other protocols (such as streams or multicast) are performed by *protocol objects*, implemented by the Communication Service.

A protocol object is layered underneath the application group that it serves; this is illustrated by figure 4. A protocol object is itself a group of cooperating elementary protocol objects, instantiated in the Communication Service context of the individual machines.

A protocol object has the privilege that it can be the source or the target of a channel, even though it is not a member of the application group. In all other respects, a protocol object is a standard Fragmented Object.

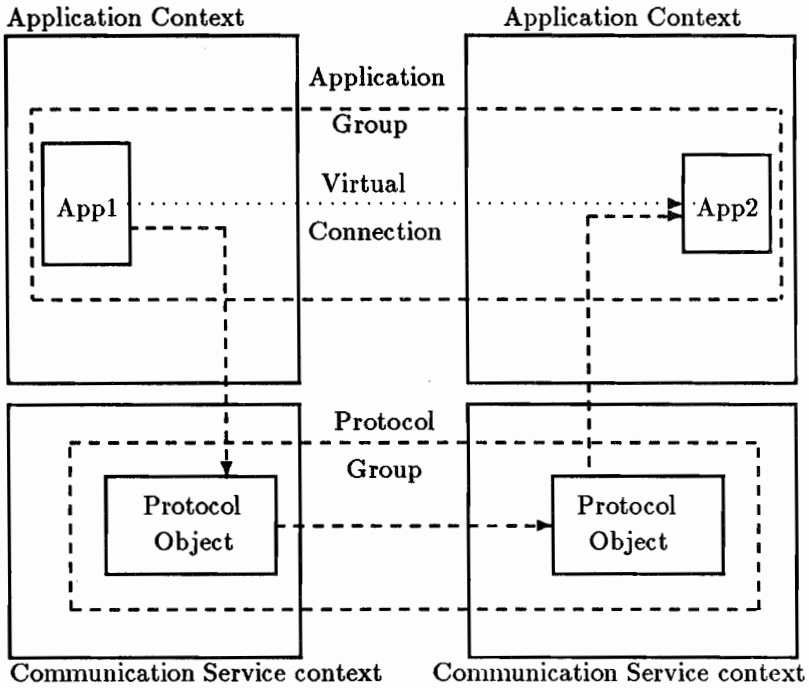

Figure 4: Distributed protocol object, layered underneath the fragmented application objects it serves

6.2.1 Families

A family is a subset of a Fragmented Object, offering multicast communications.

The following table shows the interface of families:

`obj . createFamily (groupOID) → familyRef`	Create a new family within a group
`obj . joinFamily (familyRef)`	The object `obj` joins the family
`obj . leaveFamily (familyRef)`	The object `obj` leaves the family

Family interface

Every family is identified by a reference (`familyRef`). A *family manager* is a Fragmented Object within the Communication

Service context, that manages operations to add and retrieve members in a family.

An application fragment can join a family (by `joinFamily`) to receive invocations within a family. Then a *family object* is instantiated in the Communication Service context. At first invocation by a client on a `multiChannel` connected to this family, toward this fragment, a protocol object is bound to the family object.

6.2.2 *Communication Service structure*

The Communication Service is composed of three layers (see figure 5): the network interface, the transport protocols and the invocation protocols.

A *network interface object* encapsulates access procedures to a specific network. We currently implement two types of network objects, for raw and UDP UNIX sockets.

The transport protocol layer is composed of a single *communicator object*, which implements reliable and unreliable, and point-to-point and multicast message transport. Unreliable communication doesn't pay the cost of the existence of reliability. (For more details, see Makpangou & Shapiro [1988].)

Two invocation protocol types are currently defined: RPC and MRPC. RPC is an extension of the `crossInvoke` primitive. MRPC is a multicast RPC from a fragment to the whole family.

The semantics are Only-Once-Type-1 in Spector's classification [Spector 1982]: in the absence of communication or processor failures, the callee is invoked only once per call.

6.2.3 *Establishing connection at first communication*

By default, a connection is set up between fragments at the first invocation.

As discussed in the section about the migration algorithm (see section 5.2), channels are preserved across migration.

Before the first invocation, a channel is a virtual unidirectional connection between fragments. An implicit binding is permitted by putting the reference of the callee object into the caller object's acquaintance descriptor.

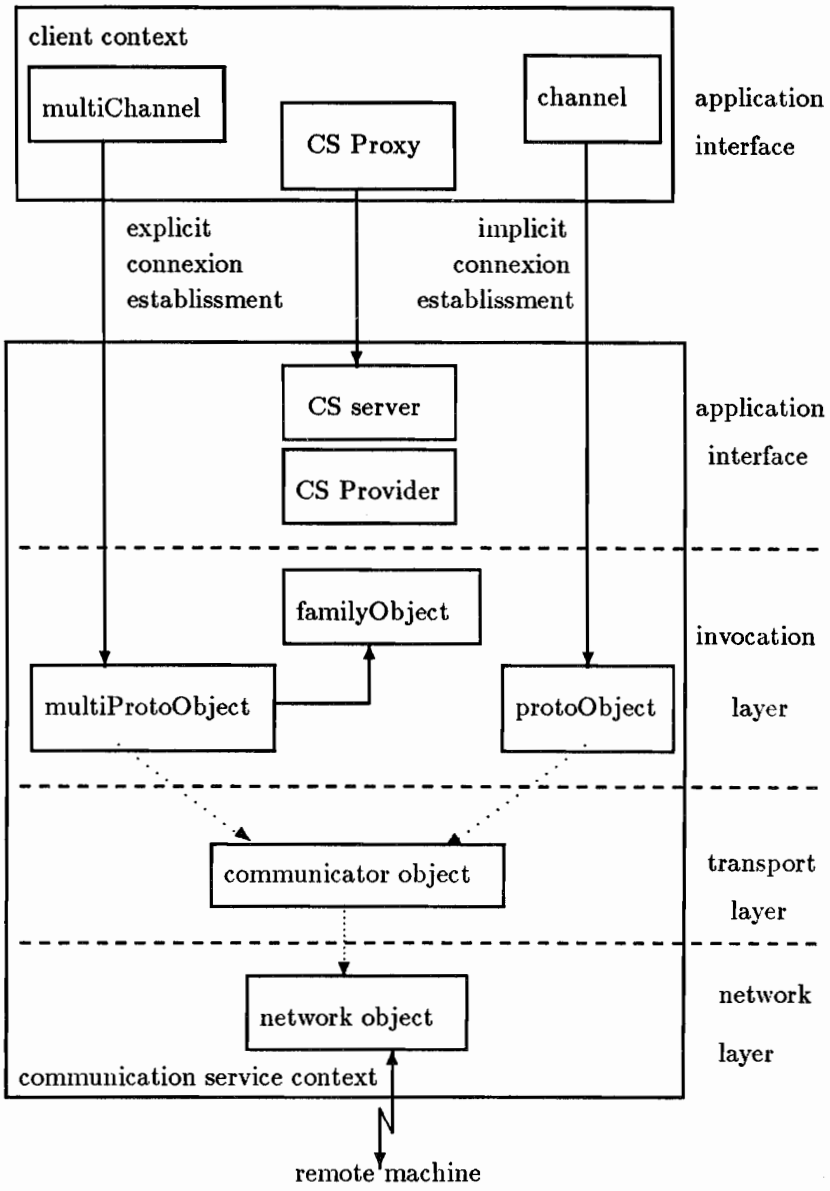

Figure 5: Communication levels

The kernel detects remote invocations, and requests the proxy of the Communication Service to establish the connection: First, it allocates a protocol object for this connection, using the protocol chosen by the channel object. Then it registers the reference of this protocol object in the acquaintance descriptor of the caller

for optimizing future invocations. At this point, the kernel retries the call and the protocol object invoked relays the call to its counterpart within the callee site. The two ends of the connection are installed and released separately.

Later, a connection might be reinstalled transparently, if one of the end objects decides to migrate.

6.3 Assessment

6.3.1 Protocol toolbox

There are not yet enough protocols in the present toolbox. However, our design is extensible, because communications are structured in several levels: Fragmented Objects manage protection domains; channels perform communication and protocol invocations; invocation objects are used to get multiple replies and manage asynchronous invocations. Above these levels, our proposed "fragment generator" (see section 4.4.2) shall manage the method level, by automatically generating the stub procedures of a Fragmented Object.

Finally, the hierarchical design of the Communication Service facilitates reuse. Multicast invocations reuse code for simple invocations and are distinct from the family concept. We hope to reuse them in turn in the future to implement new protocols, such as ISIS multicast [Joseph & Birman 1988].

6.3.2 Asynchronous invocations

In the prototype, asynchronous invocations are simulated upon synchronous, by creating a dedicated task to send the invocation. Both invocation types are necessary. Synchronous is more frequently used and more simple, so it was implemented first. Our implementation of asynchronous communication is limited by the task system limits and its performances.

6.3.3 Families

Invocations within a family are ordered. However, our implementation doesn't guarantee the order of delivery of requests to members belonging to several families. For instance, two

multicast invocations towards two non-disjoint families are not ordered for objects which are members of both families.

We need a light-weight primitive to give the members list order by protocol object. But this is not yet implemented. A broadcast to the family allows it to know members that are alive at the instant of the invocation.

7. Dependencies

So far, the mechanisms for structuring and handling communication between related objects have been described. From the system point of view, no semantics needs to be associated with inter-object invocations; it is part of the fragmented object internal protocol. This is a simple and flexible mechanism upon which a wide set of applications can be built. However, there are abnormal situations (or events) which, since they can imperil consistency within a group of objects, just can't be handled this way.

For instance, suppose we have a file server that controls sharing, using locks. Upon request to read or write a file, a proxy is exported in the client context, holding an appropriate lock. If a crash occurs on the client's side, the file server must be informed in order to remove residual dependencies and make the file available again for subsequent requests.

For this purpose, we define the *dependency* mechanism. It provides simple support for:

1. declaring objects as being part of a common *dependency family*,
2. detecting/taking into account situations which are relevant to a given dependency family, and
3. propagating the corresponding events within a dependency family.

7.1 Events

Two kinds of events are supported:

- system detectable events corresponding to *abnormal* and *normal* termination, and

- user-defined events.

Although both events types are uniformly treated, there are different motivations for supporting each kind. The abnormal termination event is generated whenever a site/context crashes; it is at the basis of the mechanism. Although normal termination could be handled directly by programmers, we support it for convenience. Finally, user-defined events provides support for a simple software signal facility. These events can correspond to any change in an object's state, not detectable by the system; a special call is used to inform the system that such an event has occurred.

7.2 Dependency family

A dependency family consists of a set of objects which agree to a common signalling protocol. We have identified two useful dependency family structures:

Master-slaves: this is the most suitable structure to use when one specific object is the propagation source of many other objects. For example, suppose we are dealing with replicated files: upon update, the master object can propagate the events to all its replicas in order to invalidate them.

Flat: in this structure all objects are at the same level. An event generated by any object in the dependency family is propagated to all the others. This can be used to define dependencies between various co-operating servers.

Broadcast is the only possible communication mode within a dependency family.

The following table gives the interface for dealing with dependencies.

`new dependencyFamily () → dep`	Create a dependency family
`dep . addDependent (dependent, dependsOn)`	Add dependent to dependency family
`dep . addOwner (owner)`	Define owner as an event propagation source

`dep . removeDependent (dependent)`	Remove dependent from dependency family
`dep . release (dependsOn)`	Remove all dependents
`dep . giveDependents(dependsOn) → dependentList`	Find all dependents of `dependsOn`
`dep . isDependent (dependent, dependsOn) → yes or no`	Check if dependent depends on `dependsOn`
`dep . hasDependents (dependsOn) → yes or no`	Check if `dependsOn` has any dependents
`dep . changed (dependsOn, event)`	A user-defined event

Dependency family downcall interface

`obj . stub(event)`	Notify event to dependent object

Dependency family upcall interface

After a dependency family has been created, calling the procedure `addDependent` has the following effects:

1. The arguments `dependent` and `dependsOn` are added in the dependency family, if not already present.
2. Any change on `dependsOn` will be propagated to `dependent`.

A master-slave dependency family is obtained by calling `addDependent` for each slave.

The `addOwner` procedure permits to specify that an already dependent object becomes an event propagation source. This means that all objects in the dependency family will be signalled if a change occurs on object `owner`. This is used to create a dependency family with a flat structure.

It should be noted that each dependent object is in charge of decoding events upon notification. System events have predefined values whereas user-defined event values are part of the protocol of each dependency family.

The dependency manager is built as an extension of the basic object manager described in section 3. Its implementation is mainly based on the use of multi-channels and family objects (see sections 6.1.1 and 6.2.1).

We believe that it should be very useful, especially for applications which are sensitive to failures. However, it is still in way of

implementation, so we can't yet, give a real evaluation of the mechanism.

8. *Persistence*

The Storage Service manages the physical storage of *typed* and *composite* objects on disks in a generic way. Once stored, objects become *permanent*; their stored representation can never be deleted. To be permanent, an object must be of type `permObject`, from which can be derived user-defined types. The state of a permanent object must be explicitly saved on storage. The permanent state of a `permObject` is never lost.

Our first goal was to implement a one-level storage, transparently integrating the so-called "vertical migration" (to and from disk) with "horizontal migration" (from context to context).

Indeed, vertical importation from storage into a context is identical to horizontal importation. Unfortunately, at the time the implementation of the Storage Service was started, horizontal migration included only object importation, not export. Therefore, a special `checkpoint` primitive is used to explicitly store an object, from memory to disk. `Checkpoint` is optimized to store only the modified portions of the object.

A permanent object can be composed of several segments. In order to be taken in account by the Storage Service, those segments must be referenced by special *permanent pointers*.

Permanent pointers allow to use indirect segments transparently. When first accessed by the object, the referenced segment is automatically retrieved from storage.

Conversely, modified segments are saved on storage at checkpoint time. Changes in the data of a segment cannot be detected automatically, because we have no control of UNIX virtual memory management. Therefore, it is up to the programmer to tell the system that a segment has been modified, by marking the permanent pointer which references it.

A permanent object has a storage object associated with it, which is a proxy of the Storage service. The `permObject` methods have a privileged access to the storage object, in order to communicate with the Storage Service. The `permObject` encapsulates

the communication with the Storage Service, hiding the storage proxy to the application. It transparently imports that proxy at creation or retrieval time.

The following table gives the interface for permanent object management.

new `permObject () → obj`	Create a permanent object in memory
`obj . delete ()`	Destroy memory image of permanent object
`obj . checkpoint ()`	Save current state on permanent storage

Permanent object down-call interface

8.1 Direct and indirect segments

The user data part of the object is composed of at least one segment, the *direct* segment, such named because it is directly referenced by the object's descriptor, the AD. The direct segment is the main body of the object allocated at instantiation time. The object can also allocate some more data, in what are called indirect segments. Both direct and indirect segments are allocated from the heap. In order to migrate together, segments reference each other by special, permanent and relocatable pointers, called `permPtr`. A permanent pointer keeps context-independent identification, along with the real pointer in the current context. Links between all `permPtrs` of a segment are maintained in order to preserve the overall structure of the object segments at migration time.

In order to make its indirect segments relocatable, an object must derive from a special class `permObject`. Mapping of indirect segments is managed by the Storage Service. The `permObject` class ensures the mapping of indirect segments of the instance, by communicating with the Storage Service (in a transparent way).

Using `permPtrs`, the user can create objects of arbitrary structure and complexity. When the object is checkpointed (see section 8), all of its modified segments are stored. The migration of a permanent object into a user context establishes the mapping of the direct segment only. Indirect segments are mapped when first accessed, by *segment faulting*.

8.2 *Permanent pointers*

The following table gives the interface for permanent pointer management.

`new permPtr () → permP`	Create a permanent pointer
`permP . delete()`	Destroy a permanent pointer
`PermP . setPtr (prevPtr, addr, size, id)`	Set the permanent pointer with `addr`
`PermP . cvt () → addr`	Convert permanent pointer to effective pointer, and map segment
`PermP . mark ()`	Mark segment as modified

Permanent pointers down-call interface

A `permPtr` variable must be assigned an effective pointer using the `setPtr` method. The first argument is a reference to the previous `permPtr` of the containing segment. All `permPtrs` of a segment are chained in a linked list, in order to encode the segments hierarchy in the state of the object. Each `permPtr` is potentially the head of the linked list for the segment it points to. The second argument is the effective address of the segment to be referenced and the third is the size of that segment. The last argument is an `OID` for uniquely naming that segment in a location-independent way. It is mainly used to control sharing of segments, since a segment may be referenced by several `permPtrs` inside the object.

As discussed in the previous section, it is up to the programmer to tell the system that a segment has been modified by calling the `mark` method of the `permPtr` which references it. It could have seemed more logical to couple the `mark` method to the segment. But in our implementation, memory segments are not objects.

A `permPtr` instance contains an effective pointer which is untyped. In order to generate the correct cast operations where necessary, a generic `permPtr` is provided as a macro. Users can define their own `permPtr` types with the following sequence of instructions:

```
typedef myclass * myClassP;
declare (gpermPtr, myClassP);
typedef gpermPtr (myClassP) myClassPtr;
```

This will produce a type-safe permanent pointer type named `myClassPtr` to objects of class `myclass`. We provide a conversion operator, which allows the use of the effective pointer. However, this operator has to be called separately before use of the effective pointer. If the dereferenced segment is not yet mapped in the object's context, it is then done transparently.

Future versions of C++ will allow to redefine the -> dereferencing operator, which will render `permPtr`'s easier to use.

8.3 Assessment

The `permPtr` classes allow to construct objects with arbitrarily complex graph of segments pointing to each other. During the vertical migration of a permanent object, the structure of its graph is preserved.

Unfortunately, this capability has not yet been integrated with horizontal migration. A complex permanent object can be migrated horizontally as a whole by check-pointing it first. But in the general case, the AS is not aware of the existence of `permPtrs` and indirect segments, and managing them is up to the programmer.

We believe that this current design is not the best one, being excessively modular: the Storage Service has been implemented as an independent service on the top of basic object management.

SOS services do not cooperate in a strong enough way. This is mainly due to the fact there is no common concept of memory segments between the kernel, the AS and the SS. An efficient and elegant solution would require control of virtual memory management.

The knowledge of the object structure, necessary to several services, is in the user part of the object, and is accessible to the Storage Service only by the expedient of dedicated storage objects.

In the future, horizontal and vertical migration of complex objects shall be unified. The `permObject` data shall be moved into the system data of the object. In order to ensure a more uniform object model, we should unify the `permObject` with the `sosObject`

class. Persistence is a mechanism which should be orthogonal to typing.

9. Naming

Naming in SOS is based on two levels: the lowest level is supported by references. However, this level isn't user-friendly enough.

Therefore, there is a second level of *symbolic names*, managed by the Name Service. The main task of the Name Service is to maintain the mappings between symbolic names and internal references. It supports naming of objects of any type. In designing the SOS Name Service, we wanted to provide much more flexibility than traditional systems. Clients have the ability to build their own views of the name space, which are the union of interesting name spaces.

9.1 The design of the Name Service

We define a *name space* as a general entity maintaining mappings between symbolic names and internal references.

A client's proxy of the Name Service will encapsulate a description of the name space. This description is materialized by *mount tables*. A mount table maintains bindings between partial names and a name space. The mount table encapsulation in the dedicated proxy's data allows the client to have its own view of the name space, independently of its execution site/context on the system, and independently of other clients.

On request from the client, name interpretation will be done by the proxy encapsulating the mount table. If a name can be partially matched, the request will be forwarded to the name space managing the name.

An interesting feature of naming in SOS is the capability to perform a *union* of name spaces under the same symbolic name. Union mounts are inspired by the similar mechanism of Plan 9 [Presotto 1988] and QuickSilver [Cabrera & Wyllie 1987]. Thus, objects from different sources can be presented and accessed under the same name. A possible conflict or ambiguity is solved

by priorities. The main advantage of this mechanism is to allow location and administrative transparency.

9.2 The Name Service interface

The initial NS proxy encapsulates a default mount table, which the client can extend. If a client is migrated, its NS proxy can be migrated too, to allow the client to keep the same naming environment.

The interface to the Name Service is implemented by a class hierarchy. A base class `baseNS` defines the common methods to all classes implementing name spaces. They are listed in the following table:

`new dynamic () → NS`	Initial NS proxy importation
`NS . lookup (name) → ref`	Lookup a symbolic name; returns associated reference
`NS . addName (name, ref)`	Associate name to object
`NS . delName (name)`	Remove name
`NS . list (name) → refList`	List objects associated with name

Name Service `baseNS` interface

From the base class `baseNS`, two classes are derived: class `nodeNS` and class `mountNS`. The class `nodeNS` represents a name space of type *directory*. It has two types of entries, leaf and directory.

The class `mountNS` represents a mount table, which accepts to mount any type of name space. Their interface is presented hereafter (in addition to the main interface above):

`new dynamic () → nodeNS`	`nodeNS` proxy importation
`nodeNS . addDir (name)`	create a directory

`nodeNS` interface
All methods inherited from `baseNS`, *plus these*

`new dynamic () → mountNS`	mountNS proxy importation
`mountNS . mount (name, ref)`	mount name space
`mountNS . umount (name)`	unmount name space
`mountNS . mask (name)`	mask name space
`mountNS . umask (name)`	unmask name space

`mountNS` interface
All methods inherited from `baseNS`, *plus these*

The `mount` operation attaches a name space to a name prefix. Successive mounts perform a union of the designated name space. For example, both *sun3:/usr/bin* and *sun3:/usr/local/bin* can be mounted on */bin*. `umount` is the inverse operation.

The `mask` and `umask` operations permit to temporarily hide, and later recover, the names managed by a name space.

9.3 Assessment

The Name Service allows users to name objects, independently of their types, and to tailor individual naming environments.

We chose to build a dedicated Name Service rather than integrating naming and filing as done in the V-System [Cheriton & Mann 1989]. The NS is built using an object-oriented methodology. The separation between implementation and well defined interfaces is flexible and extensible. We can implement integrated servers as done in the V-System.

Providing individual views of the distributed name space is justified by noting that all kind of users don't need to have the same view. A union of name space yields names unification independently of administrative constraints. Separate views are easy to implement, using proxies.

Name space union can bring name conflicts. We offer simple solutions (like priorities) to solve them.

Another important problem lies in the fact that inconsistency between an object named and its reference can arise in case of object's destruction or migration, without the Name Service being informed. This is caused by the fact that object management and their name management aren't integrated, but supported by two different system services (the Acquaintance Service and the Name

Service). Also, this yields an unstable naming environment, caused by each client having its own view.

10. Assessment of the prototype

SOS is a positive experience. It is a useful environment for prototyping distributed applications. Although we had implementation problems, it confirms that operating system-level support for arbitrary, user-defined migratory objects can be done and is useful. Some realistic applications have been built by our Esprit partners. This has allowed us to exercise our mechanisms.

All along this presentation, we already pointed out some positive and negative aspects of SOS. Many features can, however, be considered as having both pros and cons. For instance, UNIX provides a good development environment, but considerably limits its control over fundamental system resources (processor, memory), compared to a bare-machine implementation.

As a summary, we will now review the most important features of our design. For each we will mention the good and bad sides. Finally, we discuss language interfacing in the light of our experience using C++.

10.1 Kernel support

Our system is based on a minimal kernel. Most of the system functionalities are provided by a collection of system services.

However, our kernel is poorly designed. It lacks preemptive task scheduling, and concurrency control mechanisms. We use the C++ task library, a set of C++ classes for co-routine style programming, which we extended to support an exception mechanism [Stroustrup & Shopiro 1987].

A large number of tasks is pre-allocated by the kernel, at context setup time, for handling incoming invocations. Cross-context invocations are implemented using sockets in a non-optimized way. All this, together with the lack of shared libraries, contributes to large-sized, slow programs.

Finally, the kernel cannot control virtual memory. This is a great handicap, especially for the implementation of composite persistent objects.

10.2 SOS object model

Our elementary object model is both simple and powerful. The Proxy Principle [Shapiro 1986], which leads to the Fragmented Object concept, has proved to be a good structuring tool for building distributed applications. SOS is an extremely general system, that can support different object semantics.

However, our implementation is well suited only for medium and large objects (> 100 bytes). The consequence is that the programmer must distinguish between system objects and plain objects. This is confusing, especially as far as object referencing is concerned.

10.2.1 The `giveProxy` *procedure*

Programmers of a Fragmented Object are allowed to redefine their own way of giving proxies. They can give different interfaces, depending on the client's rights. More importantly, they can provide proxies with local power, according to the semantics of the resource they represent.

The main problem exhibited by our implementation is that a proxy can only be acquired dynamically. The consequence is that it is inefficient for simple, non-distributed applications.

10.2.2 The re-initializer

Calling a re-initialization procedure is a requirement for building high-level functionalities on top of the basic migration mechanism. For example, the re-initializer for a code object is a dynamic linker and type checker. This is also useful to re-validate context-dependent information. This is used, for instance, for permanent pointers management.

The call should however be generated by the system instead of the compiler.

10.2.3 Prerequisites

The prerequisite mechanism permits to express the required execution environment of an object. Indeed, a crucial issue when moving objects is to decide how much to move. This mechanism forms a good basis for code object management.

It can be used for other purposes, for instance, for a better integration of storage mechanisms.

10.3 Internal architecture

The modular structure of SOS allows extensions to be added easily. In this way, each application only pay the price for its specific requirements.

The system has been built using its own mechanisms. This demonstrates that these abstractions are adequate for building a distributed system.

However, this approach has its problems, especially at system boot time. It was also hard to debug because of the dynamic importation of system service proxies.[6] This approach also has a bad performance impact.

10.4 Interfacing with C++

The underlying mechanisms were rendered transparent by the inheritance mechanism of C++. Although object migration is implemented by the system, upcalls to the user-defined procedures `giveProxy` and the re-initializer permits to adapt various user-defined semantics.

However, the way we interface our system with the C++ language has bad impacts. Deriving from the root class `sosObject` implies some language dependency. More importantly, flexibility is lost, objects being managed according to their statically-defined type.

Another important aspect is the assumptions that such a language makes about shared address spaces. For example, allowing objects to refer to each other via pointers is a drawback for

6. We now have adapted the GNU debugger *gdb* to our dynamic link.

the support of mobility and persistency. C++ also has language features which violate encapsulation, such as public fields or "friends." This is an obstacle for Fragmented Objects to behave as real protection domains.

Finally, C++ doesn't preserve information about the class structure or the instance variables of an object at run-time. This could have avoid the need for user intervention and hacks to get knowledge about the name of the class and the size of an instance. Information about the class structure also could make the permanent pointer mechanism more transparent.

11. Conclusion

We have designed and implemented a full-sized prototype of a distributed object-oriented operating system, SOS. SOS is designed to encourage the structuring of distributed applications in Fragmented Objects (FOs), and is itself implemented as a set of predefined FOs. This article presented the design and interface of the system components, along with assessments of various design decisions. In this conclusion, we will briefly recapitulate the important lessons learned.

An object-oriented operating system is different from a traditional one in its object-oriented internal design. For instance, SOS exemplifies a communication system where protocols are objects, instantiated from a hierarchy of protocol types.

More importantly, in an object-oriented operating system, users are able to supplement system-defined mechanisms with object-specific semantics or policy. In SOS, this is done by upcalls from the system upon application objects, requesting object-specific actions before and after migration, and when an invocation is received.

OS support for arbitrary user-defined objects is viable and useful. SOS implements generic mechanisms for object management, such as identification, location, invocation, migration, storage, naming, communication. OS-supported objects incur some overhead; in SOS, objects will not reasonably be smaller than a hundred bytes. An application or a language system will typically

generate objects which are much smaller, and will map several small objects into a single OS object.

Current hardware typically supports 32-bit address spaces. This is too small to support a scheme in which objects would be uniquely identified by their address, in a single system-wide address space. This, plus the existence of two levels of object granularity (OS objects and language objects), leads to non-uniform identification. Any object is locally identified by its address; in addition OS objects have a system-wide unique identification. There is no obvious mapping between the two.

In order to mask this non-uniformity, a common technique is to generate stubs, which make remote invocation appear local. In SOS, we have the more general concept of a Fragmented Object, locally represented by a "proxy" fragment. Access to any service, be it local or distributed, always occurs by invoking a local object. Local cacheing, replication, or application-specific protocols, all fit in naturally in the proxy framework; network transparency is available but not wired in.

Fragmented Objects are important for structuring distributed applications. In SOS, unfortunately, a FO is more a concept than anything real, as there is no palpable mechanism attached to a FO. In particular, access to the internals of a FO is poorly protected. Furthermore, it is currently quite hard to program a FO. This was considered acceptable for a proof-of-concept prototype, but must be fixed for SOS to evolve into a real system. Directions for correction are: designing a specialized programming language (e.g. the "fragment generator" is a step in this direction); implementing a FO as a distributed protection domain; or using a capability-based hardware.

In an object-oriented system, components communicate by sharing objects. In SOS, a shared distributed object is naturally implemented as a FO, using the available basic building blocks. We have found that one very useful building block is atomic multicast, which guarantees that all fragments of an FO have a consistent view of its state. Other useful tools are dependencies, and local and distributed shared memory. The development of more such tools and building blocks is an important point on our agenda of future work.

Acknowledgments

In addition to the authors, SOS was designed and implemented thanks to the good ideas and the hard work of Vadim Abrossimov, Philippe Gautron, Jean-Pierre Le Narzul, and Mesaac Makpangou. Andreas Spiracopoulos implemented dependencies.

We are indebted to our partners in Esprit project SOMIW, and to the Esprit experts, George Colouris, Gianfranco Prini, and Prof. Schindler, for their strong moral (and also financial) support of our work.

Availability

The source code for SOS is available to all interested parties. However, some small parts of it are protected by AT&T licenses. Please enquire by electronic mail to *sos@sor.inria.fr*.

References

R. Balter, S. Krakowiak, M. Meysembourg, C. Roisin, X. Rousset de Pina, R. Scioville, and G. Vandôme, Principes de conception du système d'exploitation réparti Guide, Rapport Guide R1, Laboratoire de Génie Informatique, Saint-Martin-d'Hères (France), April 1987.

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

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

Luis Felipe Cabrera and Jim Wyllie, QuickSilver distributed file services: An architecture for horizontal growth, Research Report RJ 5578 (56697), IBM Almaden Research Center, San Jose, CA (USA), April 1987.

Roy H. Campbell, Gary M. Johnston, Peter W. Madany, and Vincent F. Russo, Principles of object-oriented operating system design, Technical Report R-89-1510, Department of Computer Science, University of Illinois, Urbana, Illinois (USA), April 1989.

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, Technical Report 89-04-01, Department of Computer Science, University of Washington, Seattle, WA (USA), April 1989.

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.

David R. Cheriton, The V-Kernel, a software base for distributed systems, *IEEE Software*, 1(2):19-42, April 1984.

D. Decouchant, A. Duda, A. Freyssinet, H. Nguyen Van, M. Riveill, and X. Rousset de Pina, Guide: Un système réparti à objets, In *Actes Convention UNIX 89*, pages 297-316, Paris, March 1989, AFUU.

Richard P. Gabriel, The Common Lisp Object System, *AI Expert*, pages 54-65, March 1989.

Philippe Gautron and Marc Shapiro, Two extensions to C++: A dynamic link editor and inner data, In *Proceedings and additional papers, C++ Workshop*, Berkeley, CA (USA), November 1987, USENIX.

Adele Goldberg and David Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, 1983.

Thomas A. Joseph and Kenneth P. Birman, Reliable broadcast protocols, Technical Report TR 88-918, Dept. of Comp. Sci., Cornell University, Ithaca, New York (USA), June 1988.

Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black, Fine-grained mobility in the Emerald system, *ACM Transactions on Computer Systems*, 6(1):109-133, February 1988.

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.

Mesaac Mounchili Makpangou, Invocations d'objets distants dans SOS, In Guy Pujolle, editor, *De Nouvelles Architectures pour les Communications*, pages 195-201, Paris (France), October 1988, Eyrolles.

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

Martin S. McKendry, Clouds: a fault-tolerant distributed operating system, *IEEE Tech. Com. Distributed Processing Newsletter*, SI-2(6), June 1985.

Bertrand Meyer, Reusability: The case for object-oriented design, *IEEE Software*, pages 50-63, March 1987.

S. J. Mullender and A. S. Tanenbaum, The design of a capability-based distributed operating system, *The Computer Journal*, 29(4):289-299, 1986.

David Leo Presotto, Plan 9 from Bell Labs – the network, In *EUUG Spring '88*, pages 15-21, London, April 1988, EUUG.

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

Marc Shapiro, Structure and encapsulation in distributed systems: the Proxy Principle, In *Proc. 6th Intl. Conf. on Distributed Computing Systems*, pages 198-204, Cambridge, MA (USA), May 1986, IEEE.

Marc Shapiro, Prototyping a distributed object-oriented OS on UNIX, In Eugene Spafford, editor, *Workshop on Experiences with Building Distributed and Multiprocessor Systems*, Ft. Lauderdale, FL (USA), October 1989, USENIX. Also available as Rapport de Recherche INRIA 1082, Rocquencourt (France), August 1989.

SOR, Programmer's manual for SOS prototype – version 4, Rapport Technique 103, INRIA, Rocquencourt (France), December 1988.

SOR, SOS reference manual for prototype V4, Rapport Technique 108, INRIA, Rocquencourt, June 1989.

Alfred Z. Spector, Performing remote operations efficiently on a local computer network, *ACM*, 25(4):246-260, April 1982.

Bjarne Stroustrup and Jonathan E. Shopiro, A set of C++ classes for co-routine style programming, In *Proceedings and Additional Papers, C++ Workshop*, Berkeley, CA (USA), November 1987, USENIX.

Bjarne Stroustrup, *The C++ Programming Language*, Addison-Wesley, 1985.

[*submitted Sept. 12, 1989; revised Dec. 5, 1989; accepted Dec. 15, 1989*]