A Binding Protocol for Distributed Shared Objects

Marc Shapiro

A Binding Protocol for Distributed Shared Objects

Marc Shapiro
INRIA Rocquencourt, projet SOR*
and Cornell University, Department of Computer Science
mjs@cs.cornell.edu

Abstract

A number of actions, collectively known as binding, prepare a reference for invocation of its target: locating the target, setting up a connection, checking access rights and concurrency control state, type-checking, instantiating a proxy, etc. Existing languages or operating systems support only a single binding policy, that cannot be tailored to object-specific semantics for the management of distribution, replication, or persistence. We propose a general binding protocol covering the above needs; the protocol is simple (a single RPC and one upcall at each end) but recursive; however the recursion can be terminated at any point, trading off simplicity and performance against completeness. This comprehensive, unified protocol is capable of supporting different languages and object models, and may be tailored to support various policies in a simple manner.

1 Introduction

Any computer system supports some *reference* mechanism (such as ports, sockets, file descriptors, UIDs, capabilities, or the like) for identifying and accessing objects so that they can be shared by programs. When applied to large-scale distributed object-oriented applications, existing distributed reference mechanisms have serious shortcomings: they do not support garbage collection, type safety, object groups, replication, persistence, migration, nor application policies for managing distributed data in general. The current work examines some of these issues, and proposes a binding protocol that includes up-calls to application- or language-specific policy modules. This protocol is used to ensure type safety and to implement group policies, which in turn support migration and persistent, replicated, or otherwise fragmented objects.

This study takes place in the context of SSP Chains [9], a light-weight, fault-tolerant reference mechanism supporting garbage collection; but it is likely that our ideas could be applied to other reference mechanisms as well.

*This research was supported in part by Esprit Basic Research Action BROADCAST 6370, and by a grant from Unix Systems Laboratories.

2 Background and motivations

Objects are of fine grain and arbitrary type; types change over time; and objects may be composed of distributed fragments.

Our object and reference model (explained later, in Section 3) distinguishes logical objects (the targets of references) from the lowest-level resources that implement them, called atoms (identified by address).

2.1 Limitations of existing binding systems

In operating systems, binding means locating a target, checking rights, and setting up an access path and method from the client to the target. (For instance Unix primitive open binds a file reference.) The final target is typically outside the client's address space (*e.g.*, a file implemented by the kernel or a service provided by another process). An OS binding is done at run time, is location independent, and relies on an untyped channel.

In language systems, atoms are memory locations; a binding is a mapping between a variable and such an atom along with the code operating on the data. A language binding is type safe, but references do not cross address space boundaries.

Some systems combine features of both operating system and language bindings. For instance in a remote procedure call (RPC) system [1], binding sets up a typed stub object that hides an untyped connection interface to a server. Similarly, a persistent object system [6] faults on the first access to an on-disk object, and copies it into a memory cache object, before the application can observe that it wasn't there; the result of binding is the cache, itself bound to a disk location.

In order to transparently share (potentially persistent) objects in the distributed system, there is a need for a more flexible and smooth combination of system and language bindings.

2.2 A general binding protocol

This paper specifies a general binding protocol that supports the needs stated above. It is designed to support late binding and language- or application-specific policies. It is conceptually simple (it consists of two local method calls and a single RPC) but recursive. An actual implementation may terminate the recursion at any point, trading off performance and simplicity against completeness. The examples will show that in most common cases, no more than a single RPC is needed (no recursion).

An outline of the protocol is as follows (a more detailed presentation comes in Section 5). The unbound reference targets some remote atom of the referenced object; binding produces a local *proxy* for the object (for instance, a stub or a cache) that in turn connects to a further atom (respectively, a remote server or some on-disk data). Binding invokes method accept-bind of the initial target. On the client side, instantiation method new is upcalled with the information returned by the target.

An important goal of distributed system design is transparency, *e.g.*, the target of a reference being accessed independently of its location. From the perspective of an object's client, transparency is a good thing. But the implementor of an object may need control over location; for instance, a replicated file manager must control replica locations. In our proposal, the target of a reference controls transparency via accept-bind. The accept-bind default sets up transparent remote access, but we will look at other examples, notably persistent objects.

3 Objects and references

In this section we state the object and reference model assumed hereafter. This model is relatively strong, but restricted versions of the protocol will run in a system with a weaker model.

3.1 Objects and references vs. atoms and addresses

An *object* is any entity of interest in the computer system. A *reference* designates some particular *target* object. The abstract concept of reference is implemented by a system-provided object called a *handle*.[1] The holder of a handle (a "client") may pass it as an argument or result of an invocation, and may invoke the target. Handles have a well-defined interface, described in Section 4.

An object is a logically encapsulated entity, possibly composed of sub-objects. A bottom-level object (a physical resource, such as a memory location, or a system primitive, such as a transport connection) is an *atom*. A handle designating an atom contains its *address*. The base level of invocation is the local procedure call of an atom.

User objects are composed of two kinds of atoms: memory extents (attached to some class) and OS primitives. The former is identified by a memory address, the latter by an OS identifier, that we also call an address to emphasize the efficiency requirement. Such an OS address is normally hidden by a stub. A stub is a normal memory atom known by its memory address.

3.2 Object, class and type model

We assume very little about objects: only that every object accepts upcalls to method accept-bind, specified in Section 5.2.

We assume the existence of *classes*, defined as objects that can create other objects at run-time, called *instances* of that class. A class supports upcalls to the instantiation method new, specified in Section 5.3.

A class carries the code for the instances it supports. Instatiating (*i.e.*, creating) the class itself may occur at compile/link time (as in standard C++) or at run time (as in SmallTalk or CLOS, and in C++ extensions such as SOS/C++ [3]). Run-time instantiation of a class may require dynamic linking of the code.

[1] In what follows the word "object" is reserved for an application object, as opposed to a handle.

A *type reference* characterizes an interface, *i.e.*, the signature of the *methods* or operations that apply to objects of that type. An object supports at least all the operations of its *effective type*, known at run time. The client will call at most the methods of the statically-known *presumed type* of the reference. The effective type of an object must conform to the presumed type of references to it. Conformity checking occurs either at compile time (within a statically-linked set of compilation modules) or at run time (across static checking boundaries). In order to support dynamic type checking, a handle must store the the presumed type of the reference, provided by the compiler.

We make no assumption about types. We only assume the existence of type references,[2] and that a class has a way to check its conformity with a given presumed type reference. We do not define conformity, because it is language dependent.

3.3 Fragmented objects (FOs)

An object is a single logical encapsulated entity. However, many interesting objects are actually represented by a group of atoms instantiated at different times and/or in different locations. For instance, a persistent object (see Section 6.3) has a disk image (one atom), and zero or more images cached in memory (more atoms). We call any such distributed group of atoms a "fragmented object" (FO).

Binding a reference to an FO yields a local atom, the caller's *proxy* for the FO [8]. We return to FOs in Section 6.

4 Handle primitives

Recall that a handle is the system-provided object that implements the reference abstraction. The operations on references are listed below (ignoring the obvious ones such as create, delete, duplicate).

4.1 Binding

The operation bind binds a reference, in order to make later invocations simple and efficient. Binding performs some checks and sets up an *access chain* to the target; it yields a handle to be used in place of the original. The checks should verify the pre-requisites to invocation, *e.g.*, access rights, concurrency control state, type, etc. The binding protocol is detailed in Section 5.

For instance, the Unix binding operation open takes a pathname and yields a file descriptor. It sets up an access chain consisting of the sequence: file descriptor number, file descriptor, memory-inode, cache blocks, disk block addresses, disk blocks. Thus the access chain is composed of a sequence of handles and objects, possibly of different kinds.

To support specific policies, our binding protocol gives some control to the target object.

4.2 Invoking target

A bound handle supports invoking by following down the access path to the target, and executing one of its methods.

For instance, dereferencing a pointer yields the corresponding memory address, in order to load or store the corresponding memory cell. Dereferencing a Unix file descriptor number occurs when executing a read, write or similar system call.

Just as every reference resolves to an address, every invocation resolves to a local atom invocation.

4.3 Faulting

Some systems support implicit binding through *faulting*: an attempt to invoke through an unbound handle raises a fault.

To provide faulting, unbound and bound handles appear the same to client software but the system checks the state of the reference at invocation time.

4.4 Unbinding and redirecting

Operation unbind breaks an existing binding; the reference must be bound again before use. Unbind can be user- or system-initiated, *e.g.*, using an LRU algorithm or timeouts. In general whenever any correctness conditions established at bind time may have been violated, the binding should be broken. This can occur either at the client or at the target end of the reference.

To implement unbinding, the system adds some state at both ends of the access chain and tests this

[2] Conceptually a type reference is the reference of a type object, *e.g.*, one that contains a description of the type. However, for this work, there is no obligation to keep type objects around at run time. The implementation of a type reference could just be a hash of its interface (as in SOS/C++ [3]), a unique type identifier, or some other compact representation (more in Section 7).

state at each invocation. This state can be the same as the one used to implement faulting.

Combining unbinding with faulting is one way of supporting *redirection* of references. Redirection breaks existing bindings and selects a new target. Run-time mobility of users, objects, machines, or applications, necessitates redirection. This same mechanism supports fragmented objects, as will be seen.

Redirection is transparent and atomic, for all clients of the same reference. Examples include Unix I/O redirection or the SmallTalk become primitive.

5 Binding in detail

This section details the binding protocol. First we specify it without justification. Section 5.1 outlines the different steps; Sections 5.2 and 5.3 describe the up-calls to application- or language-specific modules. In Sections 6 and 7, we will apply it to a number of interesting examples.

5.1 Specification of the binding protocol

Binding a reference is the execution of the bind method on the *handle* object representing that reference:

```
unbound-handle.bind (appl-specific-args)
        → bound-handle
```

The bind method is system-provided. It extracts the presumed type of the reference from the handle, and locates the target atom. It then performs a remote invocation of the target's binding method, passing the application-specific arguments (of which we will see some examples in the course of this paper) and the presumed type:

```
target.accept-bind (prsumd-type-ref, appl-specific-args)
        → proxy-class-ref, cntntn-ref, init
```

The accept-bind method is provided by the target object; it returns a reference to a class (that can create the proxy object), a "continuation" reference, and some initial data. These results are returned to the client side, where the system then upcalls:

```
proxy-class.new (prsumd-type-ref, cntntn-ref, init)
        → proxy-object
```

This returns a proxy, which will be an initialized atom of the specified class. The continuation reference connects the proxy to the rest of the access chain.

The default accept-bind returns a stub class for invoking the target (the server) remotely, an open connection to the target, and null initial data.

Note that proxy-class-ref is a reference to a class object. Before calling new it must be (recursively) bound, yielding proxy-class (see Section 7). The protocol will also iterate if the continuation reference is unbound (see Section 6).

5.2 Specification of upcall method accept-bind

The unbound reference designates some initial atom. The binding protocol calls method accept-bind of that original target, allowing it to control the outcome of the binding.

The first argument is the presumed type of the reference. If it conforms, then the target is assured of understanding messages that will later be sent by the proxy. The target checks this type in some language-specific manner.

The other arguments are (unspecified) "application-specific" arguments. These could include an access mode, an authenticator, a site identifier, or a transaction identifier for the caller.

Method accept-bind returns the information necessary to choose or create the proxy on the client side: a proxy class reference, some initial data, and a continuation reference. Different proxy classes may implement different presentation and/or transport protocols or different consistency policies.

Normally the initial target will return a bound reference to itself, but it can also return a null reference if no continuation is needed, or an unbound reference to itself or another atom. An unbound reference forces the proxy to iterate the bind protocol again before invoking.

5.3 Specification of proxy-returning upcall method new

After the RPC to the initial target returns to the client, the system upcalls (at the client side) method new of the specified class, with the presumed type, initial arguments and the continuation reference.

The new method is supposed to first check the class for conformity with the presumed type, presumably by calling a language-specific dynamic type checker.

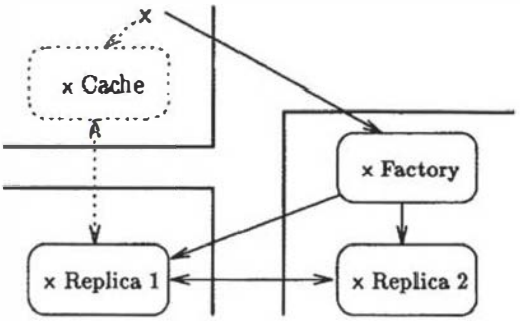

Figure 1: Fragmented object example. *Client has reference to Factory; at binding, factory initializes Cache, redirects client to Cache, redirects cache's reference to Replica 1.*

Method new returns the address of a local proxy. It creates one, or possibly re-uses an existing appropriate proxy. If creating the same proxy twice is to be avoided, then the arguments must contain enough information to ensure uniqueness.

The proxy uses the continuation reference to set up the access chain. For instance a stub will connect itself to its server; a cache to its file server; a replica to other replicas.

Note that the RPC to the initial target returned not a class object but a class reference. In order to call the method new of the class object, the reference must be bound, recursively invoking the binding protocol. If the class has already been bound, then the binding protocol can return immediately. Otherwise is should go to a trusted class repository. The proxy returned by the class repository contains the actual code for the class (it is a locally-cached, read-only copy of the contents of the repository).

6 The binding protocol applied to distributed object management

In distributed systems a logical object is often replicated for availability and fault-tolerance, or cached for performance. Thus a logical object can be *fragmented*, *i.e.*, constructed as a group of *fragments* in different locations. Few existing reference systems support groups or fragmented objects (FOs); those that do wire in a single binding policy. We show here how the general binding procotol supports many different forms of fragmented object and different object distribution policies.

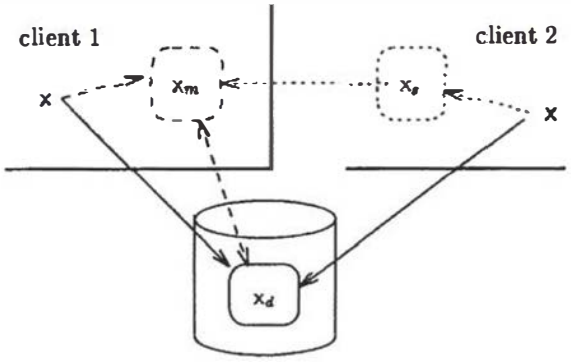

Figure 2: Persistent object and remote access examples. *When binding Client 1's reference, x_d is copied to x_m, and the reference redirected to x_m. When Client 2 binds, its reference is redirected to remote-invocation stub x_s, itself bound to x_m.*

6.1 Referencing an FO

Our solution for referencing an FO is to reference a specific *factory*[3] fragment; at bind time, the factory's accept-bind may redirect the reference to another fragment (possibly creating or migrating it on the fly). For an example, see Fig. 1. Any fragment will do as a factory, as long as it implements an appropriate accept-bind. For fault tolerance, the factory itself may be a replicated object.

6.2 Remote access through a stub

Let us first illustrate the default application of the binding protocol, providing remote access to a "server" through stubs [1]; see right part of Fig. 2.[4]

In this example, the reference of Client 2 to x_d is redirected, at bind time, to a local stub, itself connected to a remote in-memory server x_m. Thereafter, the client calls the stub's methods locally. A stub method marshalls arguments into a call message, sends the message, and awaits and unmarshalls the return message. A "stub generator" mechanically generates stubs and scions from interface specifications.

[3] We use the word factory differently from Meyer [4] for instance. He uses it for a class. We use it for the manager of a fragmented object instance, that decides in particular when to create new fragments within that FO.

[4] In this and the following figures, the arrows represent references, rounded boxes represent atoms, and thick lines delimit address spaces. The solid items exist initially; the dashed items appear during the execution of the binding protocol. A reference can be redirected, as in the case of the x reference in space B, initially directed to x_d and later redirected to x_m.

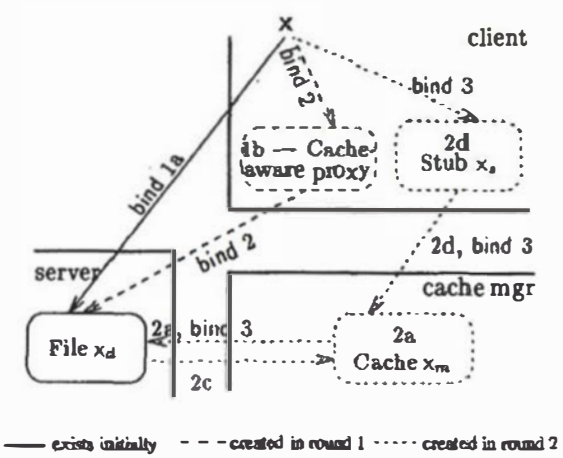

Figure 3: Cache manager example (pessimistic implementation). *Round 1: client binds file reference (1a), yielding cache-aware proxy with continuation to server (1b). Round 2: proxy instantiates empty cache x_m, passing file reference (2a); proxy binds to file (2b), passing reference to x_m (2c); this yields stub x_s connected to cache (2d). Round 3: stub binds to cache, cache binds to file and is initialized.*

The default accept-bind (also generated by the stub generator) returns a client stub class as proxy class, an open connection to the server as continuation reference, and empty initial data.

6.3 Cached persistent object

We show how the general binding protocol supports distributed access and caching of a persistent object. This is illustrated by the left part of Fig. 2. Initially, the persistent object exists as an atom on disk x_d, managed by an object storage server. The initial reference designates the disk atom through the server; the server implements accept-bind.

The purpose of bind is to copy the on-disk atom into a cache x_m, an in-memory atom. Thus accept-bind checks the client's access rights, and allocates a scion at the storage server, describing the open object. It returns a cache class with an appropriate interface, an open connection to the storage server to serve cache misses and to flush updates made by this client, and initial data to prime the cache. The cache class is instantiated, yielding x_m.

6.4 Cache call-back

In the next section (Section 6.5) we will look at some cache consistency policies. All require a *call-back* reference from the server to caches. The same binding protocol supports passing the call-back reference from the client to the server, but this requires a bit of ingenuity. We will describe two implementations, a pessimistic and an optimistic one.

The pessimistic implementation requires two rounds of binding. The first creates the cache, and the second passes the cache reference to the server. In the first round, the server's accept-bind returns a cache class, empty initial data, and an unbound reference to itself. This creates an empty cache proxy that will implement the second round, by faulting at the first access. In the second round, the application-specific arguments, supplied by the proxy, contain a reference to the cache, which will be recorded by accept-bind as the call-back. The second round returns the same cache object, primed this time with initial data to prime it, and a bound reference to the server. This implementation is fully transparent, as the client need not be aware that something special is being done to set up the call-back.

The optimistic implementation avoids the first round by instantiating the cache in advance, for instance at compile time. Based on the presumed type of the reference, it is bound at compile time to a cache (a partial binding) that implements the optimistic protocol. The proxy's bind method allocates an empty cache at the client end and passes its reference in the application-specific arguments. The class reference returned by accept-bind normally would be the same as the statically-bound cache class; if so new just returns the address of the existing cache, primed with the initial data. If however the class is different, then another round is necessary, as above.

Existing systems with call-backs, such as AFS [5], install proxies statically. Our optimistic implementation also binds a caching proxy statically. It has the advantage over AFS that, if a different policy is decided after compiling the client, our optimistic implementation automatically falls back to the pessimistic implementation.

6.5 Shared persistent object caching policies

Even when multiple caches are allowed for the same object, no more than one should be allocated per client

or per machine. The Spring binding protocol [7] is especially designed for cache management and ensuring uniqueness. There is a single (optional) file cache manager per machine; all clients on one machine share a single cache to any file, via the cache manager.

Again, the general binding protocol accomodates this. A pessimistic, transparent implementation needs three rounds of binding, as illustrated in Fig. 3. Initially the client has a reference to a file at the file server. In the first round the client contacts the file server; the file server returns a proxy class that knows about cache servers (and an unbound reference to itself). In the second round, the proxy first asks the local cache server to instantiate a cache x_m, passing the file reference as an argument; the cache is initially empty and refers to the server through the (unbound) reference passed by the client. The proxy then binds, passing the x_m reference as application-specific argument; the file server responds with the same proxy class and the (unbound) reference to x_m (*not* to itself). In the third round, the client binds to a stub x_s, which binds to the cache, with itself binds to the server, which returns initial data.

As an optimization, if the server already has a callback reference to the appropriate cache manager, it responds with a reference to the cache, skipping the second round. If the cache is already bound, the cache-server binding of the third round can also be skipped. Also, an optimisitic implementation (along the lines of Section 6.4) needs only a single round, binding directly to the cache manager. The optimistic implementation is the same as the binding protocol used in Spring [7], with the added advantage of automatic fall back onto the pessimistic implementation in case a different policy is decided at run time.

7 How the recursions in the protocol support type and class management

In the previous section we showed how the binding protocol supports object-specific distribution policies. In the current section, we will show how it supports language-specific code and type management policies.

7.1 Recursive definition of handles and types

If a handle contains a presumed type, and a type is represented by a handle, then a type-handle contains a type-type-handle, and so on. What then is the type of a type? One reasonable interpretation is to consider an object universe with a partitioned type universe. For instance, each site in the system manages a separate schema; then the type-type identifies the schema. Two types with different identifiers in different schemas may still conform; if a global schema oversees the local schemas, then recursively invoke the global conformity checker.

The recursion terminates by a (static) decision not to implement any higher levels. For instance, for those references that can be type-checked statically, the presumed type can be left out of handles, and the corresponding run-time check omitted. In a system that needs dynamic type checking but supports only a single type system, then type-types (and higher levels) are omitted.

There is clearly a trade-off between flexibility and potential complexity of a type check; it is doubtful a real system would need any more than a small number of levels. For instance, Spring uses a single level; the SOS bind is recursive but only a single level is ever used in practice. Similarly, the CORBA specification implicitly assumes an optimistic implementation [2].

7.2 Dynamic linking and recursive binding of classes

The binding protocol itself is recursive since binding a reference returns a class reference, which in turn must be bound before use.

The first level of recursion ensures that the class of an object is loaded before creating instances of that class. A class manager class class loads new classes, *i.e.*, reads the class code from a class repository, and dynamically links it into the client's address space. The recursion terminates there if the class manager is statically linked (this is the case in SOS for instance [10]). If however multiple class managers were needed (for instance to support different object code formats) then the binding for a class-object would return the reference for the corresponding class-class object, causing another level of recursion.

Again, class references can use a specialized representation, such as string name or registration number in the class repository, when there is no possible confusion with other handles. Multiple representations can be distinguished from one another by recursion.

Again there is a trade-off between flexibility and complexity. Dynamic linking can be avoided entirely if all classes are known statically. Or, dynamic linking

can be supported, but under control of a statically-linked class manager, class class.

8 Conclusion

We have presented a simple but comprehensive binding protocol for references to distributed shared objects. This protocol consists of one RPC from client to target, one up-call to the target object's accept-bind, and one up-call to the language-support run-time new at the client's end. The latter up-call is directed by the data returned by the former.

We have shown many examples of useful applications of the general protocol; by appropriate action in accept-bind and new, objects can select between many different distribution policies, *e.g.*, data shipping (caching) or function shipping (remote access). Similarly, the protocol enables the language-support runtime to perform run-time type checking and/or loading of code efficiently and safely.

The efficiency of the protocol can be questioned, since it is based on unlimited recursive RPCs. However, the examples make it clear that in most cases, an optimistic implementation uses a single RPC, emulating the binding protocols of systems such as RPC systems, SOS, Spring, or CORBA. However, our optimistic implementation is also more versatile, since it automatically falls back on the pessimistic approach when the static assumptions are false. It does this at the price of some local checking.

The work described is part of the Soul distributed object support system. The reference system of Soul (SSP Chains) has been implemented; however it currently supports only a simplistic version of the binding protocol. At the time of this writing, the protocol and the examples are specified on paper only, but are we plan to integrate these ideas into Soul and check them out in practice.

An extended version of this paper, as well as related papers are accessible by anonymous FTP on host ftp.inria.fr, *directory* INRIA/Projects/SOR.

Acknowledgments

I wish to thank Paulo Amaral, Andrew Black and Jeff Chase for first raising some of the issues discussed in this paper. Mesaac Makpangou and Peter Dickman provided the initial ideas for referencing and binding to FOs. David Plainfossé implementated SSP Chains; Marcin Skubiszewski did a graphical animation of the implementation. Our partners in the BROADCAST Esprit Basic Research Action encouraged me to attack the architectural issues of referencing objects in a large-scale distributed system. Comments from readers of early versions of this paper, notably Laurent Amsaleg, Alain Sandoz, Robert Cooper and Marcin Skubiszewski, were extremely helpful.

References

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

[2] Object Management Group. The Common Object Request Broker: Architecture and specification. Technical Report 91-12-1, Object Management Group, Framingham MA (USA), December 1991.

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

[4] Bertrand Meyer. *Object-Oriented Software Construction*. Computer Science. Prentice Hall, 1988. ISBN 0-13-629031-0.

[5] James H. Morris, Mahadev Satyanarayanan, Michael H. Conner, John H. Howard, David S. H. Rosenthal, and F. Donelson Smith. Andrew: a distributed personal computing environment. *Communications of the ACM*, 29(3):184–201, March 1986.

[6] R. Morrison, M. P. Atkinson, A. L. Brown, and A. Dearle. Bindings in persistent programming languages. *SIGPLAN Notices*, 23(4):27–34, April 1988.

[7] Michael N. Nelson, Yousef A. Khalidi, and Peter W. Madany. Experience building a file system on a highly modular operating system. In *Proc. Symp. on Experience in Distributed and Multiprocessor Systems (SEDMS IV)*, pages 123–141. Usenix Association, September 1993.

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

[9] Marc Shapiro, Peter Dickman, and David Plainfossé. SSP chains: Robust, distributed references supporting acyclic garbage collection. Rapport de Recherche 1799, Institut National de la Recherche en Informatique et Automatique, Rocquencourt (France), nov 1992.

[10] 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.