A Binding Protocol for Distributed Shared Objects*

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

1 April 1994

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,

*Extended version of a paper presented at the 14th Int. Conf. on Distributed Comp. Sys. (ICDCS), Poznan, Poland, 21–24 June 1994.

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

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 [21], 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.

2 Background and motivations

Current distributed systems support a number of predefined abstractions, such as files or ports, that can be referenced and shared. We wish to extend such support to user-defined objects. This has consequences on the reference and binding system, because 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; it locates a file server, checks the user's rights, and sets up internal data structures and network channels for future `read`s and `write`s.) 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). The binding returns a local OS-provided atom to communicate with the remote target (*e.g.,* an open file descriptor or a socket or port). 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 [6], binding sets up a typed stub object that hides an untyped connection interface to a server.

Similarly, a persistent object system [16] 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.

Despite the similarity of mechanism, typically RPC systems do not support persistent objects, nor vice-versa. 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. The mechanism should be independent of any particular language or application area; it should support type-safe bindings even for languages not designed for run-time checking (such as C++). A flexible policy would be furthermore would be extensible to new uses.

2.2 Requirements of large scale on referencing and binding

Large scale imposes its own requirements on the reference system. We do not address the algorithms for efficiently and reliably federating heterogeneous addressing domains, described elsewhere [20, 21]. We point out that when a reference crosses a domain boundary, it is necessary to do address translation at the boundary. For instance, a stub translates a local address to a connection address.

A large scale distributed system, applications execute continuously. New objects and new types must be accomodated without interruption of service or recompiling. This creates a need for dynamic instantiation of objects, dynamic linking of code, and dynamic type checking of references.

Large scale also creates a need to replicate an object or fragment its data over multiple locations, for availability, performance and/or fault tolerance. We address the referencing needs of such *fragmented objects* (FOs) later in this document (see Section 6).

2.3 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, which type-checks the binding and the remote interface, and returns what is needed to create the proxy and its connection, possibly redirecting it to another atom. On the client side, instantiation method new is upcalled with the information returned by the target. This type-checks the local interface and returns the proxy, instantiated with its connection. As a side-effect, code for the proxy may be dynamically linked.

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. With the proposed binding protocol, any object is free to implement its own group, persistency, type and class management.

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

Objects are shared dynamically and at a fine granularity: the typical size of a shared data object in existing persistent object systems is known to be of the order of tens, sometimes a few hundreds, of bytes [1, 2]. References are used heavily and must be cheap.

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,

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

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 [12] or CLOS [10, 13], and in C++ extensions such as SOS/C++ [11]). Run-time instantiation of a class may require dynamic linking of the code.

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

[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 Lynx [18], SOS/C++ [11] and Network Objects [5]), a unique type identifier,

presumed type reference. We do not define conformity, because it is language dependent.

The preceding object model is rather strong, because it assumes classes and type references are available at run time. Languages such as Emerald and CLOS support this model. A standard C++ environment does not, although extensions have been proposed that do [11].

If the all the features of our object model are not available, they can be emulated by user-level conventions and discipline. Alternatively, the binding protocol can be run in a more restricted object model, giving up either dynamic type checking, or dynamic selection of the proxy class, or both.

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). Another example is a replicated object, formed of the set of replicas. A remotely-accessed object is similarly logically composed of the "real" atom (the server) and the remote access stubs. We call any such distributed group of atoms a "fragmented object" (FO).

A FO is a logical entity only; its physical representation is the set of its fragments. The only way to access an FO is through one of its fragment atoms. Binding a reference to an FO yields a local atom, the caller's *proxy* for the FO [19]. 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). A bound handle supports invoking the reference's target; binding prepares for invocation. Binding may be explicitly performed by the application or implicitly upon a fault.

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

or some other compact representation (more in Section 7).

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. The file descriptor can be further bound by mmap, setting up the internal mapping tables, and yielding a virtual address where the file is mapped. Thus the access chain is composed of a sequence of handles and objects, possibly of different kinds.

The later binding is allowed to occur, the more flexible the system. Binding may occur end-to-end in one operation, or a bind may be only partial, necessitating another bind later. 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. The fault handler repairs by calling bind under the covers, and restarts the faulting access. Faulting can make binding transparent to clients. This is typical of virtual memory systems (page faults) and persistent object systems (object faults).

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. The check is done by hardware in the case of virtual memory, or in software for object faulting.

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 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. Runtime 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.

4.5 Bound and unbound handles

An unbound reference is one that has not yet been checked and/or connected. An unbound handle identifies its target (in order to do the binding) but not necessarily in the most efficient way, and the target may change during the binding. It is associated with the information (location, access rights, and/or type) that is to be checked in the course of binding.

A reference may be partially bound, *i.e.,* only some of the attributes are known to be established. For instance, the access chain is incomplete; or, only part of the target's interface has been validated.

A bound handle contains the address of the next element of the *access chain*, often an OS-provided open channel address.

Different kinds of handles often co-exist within a single system, for instance those containing pathnames and those containing addresses. Binding often takes one kind of handle (say, a pathname) and yields another (say, a file descriptor number).

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 (application-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 (presumed-type-ref, application-specific-args)
→ proxy-class-ref, continuation-ref, initial-data

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 (presumed-type-ref, continuation-ref, initial-data)
→ 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.

The type of the application-specific arguments is determined by the presumed type (since accept-bind is part of the target's interface). However, since binding has not occurred yet, a presentation protocol for arguments has not yet been chosen; therefore, there must be a single universal presentation protocol for accept-bind arguments. If selecting a presentation protocol is desired, this can be done, either by adding a protocol-type argument, or by using statically-typed stubs, or by adding another phase to the bind, after instantiating the selected proxy class. The latter can be ensured by accept-bind returning an unbound reference. We will see examples in Sections 6.4 and 6.5.

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.

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 type of the initial arguments is not specified here, but is known by the class. This need not be type-checked since the class and the arguments are sent together.

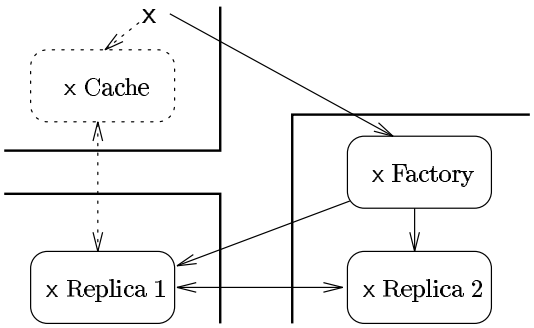

Figure 1: Fragmented object example. *Client has reference to Factory; at binding, factory initializes Cache, redirects client to Cache, redirects cache's reference to Replica 1.*

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.

6.1 Referencing an FO

In order to reference FOs, we consider four options. We reject two traditional options, because they do not support our model adequately: referencing individual fragments and system-resolved group references. The third option combines the previous two. Our fourth, preferred mechanism relies instead on an object-provided accept-bind and redirecting references. We now examine and compare these options.

1. Most existing systems support references to individual atoms only. In such a system, each fragment of an FO has its own reference. By exporting the reference of an appropriate fragment to each client, it is easy to control which client binds to which fragment. However, this is early binding (to be avoided), and fails its purpose if clients pass references to one another. Furthermore, clients do not see the FO as an encapsulated unit, but instead have the visibility of the multiple fragments.

2. Some systems, such as the the V-System [8] or Isis [4], reference a group as a single unit. The system chooses what fragment a reference binds to. Encapsulation is enforced. But the programmer of the FO cannot easily distinguish among fragments; and has no control over a client's binding. A particular FO cannot select its own binding policy.

3. A third option, used in SOS [22] and in Gaggles [7], combines the above two. Here, a fragment carries both a fragment identifier and an FO identifier; if a client uses the FO identifier then binding will go to an arbitrary fragment; if a fragment identifier is used, then binding will yield that fragment. This behaviour is confusing to users.

4. 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 [6]; see right part of Fig. 2.[4]

[3] We use the word factory differently from Meyer [14] 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

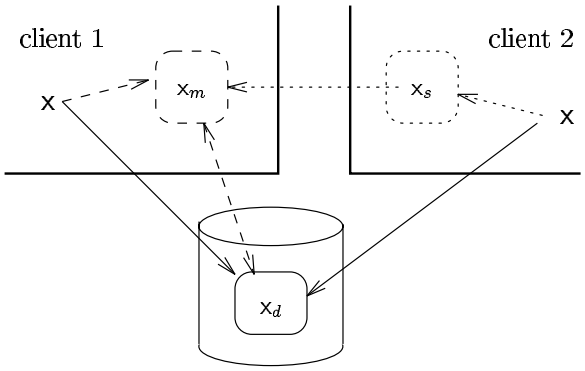

Figure 2: Persistent object and remote access examples. *When binding Client 1's reference,* x_d *is copied to* x_m*, and the reference redirected to* x_m*. When Client 2 binds, its reference is redirected to remote-invocation stub* x_s*, itself bound to* x_m*.*

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.

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. If multiple presentation protocols are to be supported (as in LRPC [3]), the stub generator will generate multiple corresponding stub and scion classes; one will be selected at run time.

Binding in existing RPC systems performs a subset of the above protocol. The "application-specific" arguments would include an authenticator for the caller. As the presumed type they pass a "version number" set by the writer of the interface, that should be changed whenever the interface changes (but this is only a convention, relying on manual action). Most RPC systems support only a single stub class per type; so, they do not neeed to return a proxy class reference. however, if more than one presentation protocol is supported, they return an indication of the selected protocol, that is like our proxy class reference.

---

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.

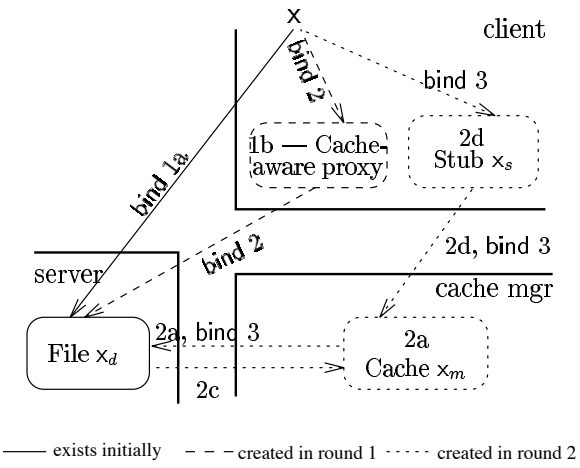

Figure 3: Cache manager example (pessimistic implementation). *Round 1: client binds file reference (1a), yielding cache-aware proxy with continuation to server (1b). Round 2: proxy instantiates empty cache* x_m*, passing file reference (2a); proxy binds to file (2b), passing reference to* x_m *(2c);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

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 [15], 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

Multiple caches of the same object must be kept mutually consistent. Many different consistency policies are possible; most require more than one round of binding.

The simplest policy is no consistency at all. This is appropriate for mostly-read objects such as a class.

Another simple policy is to ensure that no more than a single cache exists for any one object at any particular time, as illustrated in Fig. 2. When the server has allocated one cache x_m (left-hand side of the figure), it redirects all future bindings along the call-back to that cache, allocating a remote-access stub x_s to

the new client (on the right of the figure). The server serializes bind requests and remembers cache allocations and associated call-backs.

A widely-used policy in distributed file systems, such as AFS [15], is to allow multiple caches to the same object but ensure mutual consistency. When an application commits by "closing the file," updates are sent from the client cache to the server, and propagated to other caches along the call-backs.

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 [17] 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 call-back 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 [17], 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

The definition of a handle is recursive, since a handle contains a type reference for its presumed type, *i.e.,* another handle.

At first glance, the recursive definition of a handle seems to lead to an impossibility. Handles can be implemented in practice however, because (i) not all handles need be implemented identically, and (ii) the recursion may terminate after any suitable degree.

Remark (i) means that the presumed type handle embedded within an ordinary handle may be very short. This takes advantage of the fact that there are fewer types than ordinary objects; types can be identified relative to the universe of types, whereas objects are identified relative to the (larger) universe of all objects.

Remark (ii) means for instance that when type checking is static, the presumed type may be omitted from handles.

Let's pause a moment to look at this argument the other way around. 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 same idea could possibly type check across different programming languages by recursing to a "super type system", provided for instance by a "universal" interface definition language.

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.

In general, different representations of handles can co-exist when appropriate, if there is no risk of confusion. For instance, a type can be identified by its string name, or by its registration number within a type registry; this assumes that a normal handle will never appear in lieu of a type handle. (Alternatively, a type identification can consist of a hashcode computed from its interface —assuming a single universe-wide hashing function— as in Lynx [18], SOS/C++ [11] or in Network Objects [5].)

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 [9].

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 [22]). 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.

7.3 Grouping

Grouping makes possible a further simplification. For instance, one could assume that all the type references in a particular module or space are relative to the same schema. Then a single reference to the schema for the whole module replaces individual references per handle.

If there is a single schema for the whole system, its reference can eliminated altogether.

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.

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] Malcolm Atkinson. Discussion at Int. Workshop on Distributed Object Management, Edmonton Alberta (Canada), August 1992.

[2] R. Balter et al. Principes de conception du système d'exploitation réparti Guide. Rapport Guide-R1, Laboratoire de Génie Informatique, Grenoble (France), April 1987.

[3] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. Lightweight remote procedure call. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 102–113, Litchfield Park AZ USA, December 1989. ACM.

[4] Kenneth P. Birman. Replication and fault-tolerance in the ISIS system. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, Orcas Island WA (USA), December 1985.

[5] A. Birrell, G. Nelson, S. Owicki, and E. Wobber. Network objects. In *Proc. 14th Symp. on Operating System Principles*, Asheville NC (USA), December 1993. ACM SIGOPS.

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

[7] Andrew P. Black and Mark P. Immel. Encapsulating plurality. In *European Conf. on Object-Oriented Programming*, Kaiserlautern (Germany), July 1993.

[8] David R. Cheriton and Willy Zwaenepoel. Distributed process groups in the V Kernel. *ACM Transactions on Computer Systems*, 3(2), May 1985.

[9] Digital Equipment Corporation, Hewlett-Packard Company, HyperDesk Corporation, NCR Coporation, Object Design, Inc., and SunSoft, Inc. The Common Object Request Broker: Architecture and specification. Technical Report 91-12-1, Object Management Group, Framingham MA (USA), December 1991.

[10] Richard P. Gabriel, Jon L. White, and Daniel G. Bobrow. CLOS: Integrating object-oriented and functional programming. *Communications of the ACM*, 34(9):29–38, September 1991.

[11] 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.

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

[13] G. Kiczales, M. Theimer, and B. Welch. A new model of abstraction for operating system design. In *1992 Int. Workshop on Object Orientation and Operating Systems*, pages 346–350, Dourdan (France), October 1992. IEEE Comp. Society, IEEE Comp. Society Press.

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

[15] 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.

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

[17] 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.

[18] Michael L. Scott and Raphael A. Finkel. A simple mechanism for type security across compilation units. *IEEE Transactions on Software Engineering*, 14(8):1238–1239, August 1988.

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

[20] Marc Shapiro. A fault-tolerant, scalable, low-overhead distributed garbage detection protocol. In *Tenth Symp. on Reliable Distributed Sytems*, Pisa (Italy), October 1991.

[21] 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. Also available as Broadcast Technical Report #1.

[22] 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.