UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Rapports de Recherche

N° 2007

Programme 1

Architectures parallèles, Bases de données,
Réseaux et Systèmes distribués

FLEXIBILITÉ DES LIAISONS VERS DES OBJETS DE GRAIN FIN RÉPARTIS

FLEXIBLE BINDINGS FOR FINE-GRAIN, DISTRIBUTED OBJECTS

Marc Shapiro

Août 1993

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél.: +33(1)39 63 55 11

Contents

Flexible Bindings for Fine-Grain, Distributed Objects

Flexibilité des liaisons vers des objets de grain fin répartis*

Marc Shapiro

August 1993

Abstract

We examine the issues of referencing and binding to objects in large-scale distributed systems. Flexible binding mechanisms are necessary to support large-scale, long-running systems and applications. To assist object-oriented programs there is a need for cheap, fine-grain references, supporting objects of arbitrary type and distributed garbage collection. Support for large-scale systems includes referencing fragmented objects (groups), and federation of heterogeneous addressing mechanisms. Our object model separates logical objects from their representation atoms; binding a reference connects its source to an actual target atom; invocation calls a procedure of the bound atom. The primitive reference operations are duplication, binding, unbinding, redirecting, and invoking target. Binding may cause type-checking, instantiation of an atom and linking of the atom's class. To support programmer-defined policies, the target object may redirect the reference to any of its atoms, particularly at bind time. We specify a general (but simple) binding protocol that supports the above mechanisms; actual implementations can be derived from it, trading off completeness for performance.

Résumé

Nous discutons les problèmes des références et de la liaison aux objets dans un système réparti de grande échelle. Les systèmes et les applications de grande échelle et s'exécutant sans interruption nécessitent des mécanismes de liaison flexibles. Les programmes à objets nécessitent des références bon marché vers des objets de type quelconque et de grain fin,

*This research was supported in part by Esprit Basic Research Action BROADCAST 6370.

se prêtant au ramasse-miettes réparti. Le système étant de grande échelle, cela entraîne le besoin de référencer les objets fragmentés (ou groupes) et de fédérer des mécanismes d'adressage hétérogènes. Notre modèle fait une distinction entre objets logiques et leur représentation par des atomes ; lier une référence consiste à connecter effectivement la source avec un atome cible ; l'invocation appelle une procédure de cet atome. Les opérations primitives des références sont la duplication, la liaison, la déliaison, la redirection, et l'invocation de la cible. La liaison peut déclencher une vérification de type, l'instantiation d'un atome, et le chargement de la classe de l'atome. Afin de permettre au programmeur de définir les politiques de liaison, l'objet cible peut rediriger la référence vers un quelconque de ses atomes, ceci en particulier au moment de la liaison. Nous spécifions un protocole général mais simple de liaison, se prêtant aux mécanismes décrits ; on en dérivera des réalisations en faisant des compromis entre la complétude et les performances.

1 Introduction

We re-examine the issues of referencing objects in a distributed system, binding, and invocation. These issues have been studied extensively in the past (see for instance Saltzer [16], Needham [15] or Goscinski [9]). Some new considerations motivate the current paper.

Interest in garbage collection in distributed systems is relatively recent [12, 21]. Althouth garbage collection was a prime motivating factor for this work, it is out of the scope of this paper.

Large scale distributed systems and applications execute continuously. Object references are passed dynamically. New objects, and types appear and must be accomodated without interruption of service. This creates a need for dynamic instantiation of objects, dynamic linking of code, and dynamic type checking of references.

Objects can migrate, be copied (e.g. from disk to memory), replicated or fragmented in application-specific ways. For referencing a fragmented object, we propose to combine referencing a "factory" fragment, with the ability by the factory to redirect the reference to another fragment at bind time. The target object also has the option to redirect a reference even after binding, if necessary.

We specify an abstract binding protocol that provides for the above needs in a general and simple way. We believe it is complete enough to take into account the needs of most type systems and application needs. It is simple enough that performance, complexity and completeness trade-offs are clear.

To assume a homogeneous reference system (e.g. universally unique identifiers) is unrealistic in a large-scale system, such as a world-wide internet span-

ning untrusting corporations. Instead, we propose to use the native local addressing mechanisms and translate at the borders of the addressing domains. This is more efficient in the common case where the reference source and its target reside in the same addressing domain.

In our proposal, objects control transparency. Transparent access is usually a good thing but wired-in mechanisms will be inappropriate in some cases (for instance, a replicated file manager should control replica locations; or a document circulation system know the logical and physical layout of the office). Our proposal separates a client from the object it uses. A reference is transparent for its client, i.e. binding always yields a local object that is invoked in the same way, whether the initial target was local, remote, on disk, fragmented, etc. By default, binding is also transparent to objects, i.e. the default binder makes an object accessible wherever it resides, with no extra work. However, a target object has the option to replace the default binder in order to control the outcome of binding.

One can argue that objects are no different from classical OS abstractions such as files and processes. In practice there are some visible differences. Objects are fine grain and of arbitrary type. Since we are looking at large-scale systems that never stop, new types, classes, and objects appear. There is a need for dynamic type checking, dynamic instantiation, and dynamic linking. It is necessary to support fragmented objects. References should support automatic garbage collection as well as other resource management mechanisms such as clustering and load sharing.

These mechanisms might appear excessively complex to the reader. Our approach here is to emphasize completeness and correctness. By exposing these issues in their full generality, it can then clear where design trade-offs and simplifications can or must be made (e.g. when a dynamic check can be replaced by a static decision, and who makes that decision) and what are the consequences.

The discussion will proceed as follows. The coming section defines objects and references, and specifies the source end of the abstract binding protocol. Section 3 examines more closely referencing fragmented objects; it specifies the target end of the binding protocol. In Section 4, we briefly present a technique, called SSP Chains, that solves the problems raised previously. We conclude in Section 5 by recapitulating the main insights of this study.

2 Objects and references

An *object* is any entity of interest in the computer system. A *reference* designates some particular object, called its *target*. This paper concentrates on internal references (as opposed to symbolic or user-level naming).

The abstract concept of reference is implemented by a system-provided object called a *handle*. 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 2.3.

An objet is a logically encapsulated entity, composed of sub-objects. The bottom level objects (physical resources or system primitives such as transport connections) are called *atoms*. An atom is designated by its *address* (e.g. a memory address or a network address). The base level of invocation is calling an atom.

Ultimately, a handle evaluates to an address; invocation boils down to calling an atom. But reference systems decouple logical references from addresses; a layer of software interpretation can provide late binding, protection, encapsulation, transparency, or even more advanced features such as object versions, election, etc.[1]

2.1 Object model

Objects are shared dynamically and at a fine granularity: the typical size of shared atoms in exisiting persistent object systems is known to be of the order of 64 bytes [1]. The handle primitives are executed very frequently. References must be cheap.

Many interesting objects are represented by more than one atom, either in succession (e.g. persistent objects; see Section 2.4) or simultaneously (fragmented objects; see Section 3). In a way that will be described shortly, binding selects an appropriate target atom, and returns its address.

We assume very little about objects. Every object must accept upcalls to method accept-bind as specified in Section 3.2. The default accept-bind provides transparent remote access.

We assume the existence of *classes*, defined as objects that can create other objects at run-time. A class supports the upcall method new, specified in Section 2.6.

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 *dynamic type*. The holder of a reference will call at most

[1] In fairness, it should be noted that with some effort, it is possible to have similar features in a system that uses only addresses. For instance, linkers know how to patch the addresses in a binary file in order to reorder or select resources, and copying garbage collectors [23] similarly are capable of moving objects around dynamically by patching the pointers to them. However these techniques are complex and require very tight coupling with compilers.

the methods of the *static type* of the reference. The dynamic type of an object must conform to the static type of references to it.

We make no assumption about types. We only assume the existence of type references.[2] We do not define conformity because it is language dependent. We only assume that, given a static type reference, the new method of a class object can check that the instances it creates do conform.

2.2 The different levels of references

In this section we define the different levels of references and relate them to each other.

A number or string identifying an object (within some domain) is its name or *identity* (within that domain). An *address* is the identity of an *atom*. Every reference eventually evaluates to one or more addresses.

Addresses are efficient by definition. Strictly speaking an address is recognized by hardware and an atom is the corresponding hardware resource, e.g. a memory cell, a device, or a network attachment point. We also use the words address and atom more loosely, to mean an identity at some "bottom" layer of abstraction we are not interested in refining, e.g. a network transport address and the corresponding service access point.

Handles have a well-defined interface, described in Section 2.3. We distinguish a handle from an identifier (a bit-for-bit copy of the target's identity) because the latter don't have a well-defined interface.

A pointer is an identifier containing a memory address. Other examples of identifiers include a Unix pathname, a Unix file descriptor number, or a TCP socket number. Two examples of handles that are not identifiers are a Unix file descriptor itself, or a communication port.

Two different kinds of reference implementations co-exist in most computer systems: user-level names, for use by humans, variable-length character string identifiers (for instance Unix pathnames); and more efficient internal references, for use by programs, implemented as fixed-length handles. Binding (see Section 2.3.1) a user-level name yields an internal reference. This paper concentrates on internal references.

[2]Conceptually a type reference is the reference of a type object, e.g. one that contains a description of the type, but there is no obligation to keep such an object around at run time. The implementation of a type reference could just be a hash of its interface (as in Lynx [17], SOS C++ [8], and Network Objects [?]) or some other compact representation.

2.3 Handle primitives

The primitive operations on handles are listed below. This list is very general; we contend that all reference systems can be specified as a suitable combination of these primitives. Note the absence of an equality test, considered dependent on the semantics of the target.

The reference graph is a dynamic data structure. In addition to the obvious primitives to create or delete a reference, primitive duplicate enables sharing of the target; passing a reference as an argument or result duplicate's the corresponding handle.

These operations might affect the handle only. In some systems, however, they have further effects, on the reference system or the target. For instance in many systems, a handle cannot be created separately from its target. In a garbage-collected system, deleting the last reference to an object causes the target to be reclaimed.

2.3.1 Binding, faulting, redirecting

A handle must be bound before it can be used to access the target. Binding sets up a *access chain* (associated with a communication protocol along that path) to a particular atom of the target object. It yields a handle replacing the original.

For instance, in Unix the binding operation for pathnames is open, yielding a file descriptor. The access chain consists of the file descriptor number, the file descriptor containing a pointer to a memory-inode, and finally the memory-inode itself, containing a list of disk block addresses. 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.

Some systems support implicit binding through *faulting*: use of an unbound handle raises a fault. Execution diverts to a fault handler. The handler repairs by calling bind and restarts the faulting access. This is typical of virtual memory systems (page faults) and persistent object systems (object faults).

Operation unbind breaks an existing binding; the reference must be bound again before use. Unbind can be done implicitly by the system, e.g. using an LRU algorithm. In general whenever the correctness conditions established at bind time may have been violated, the binding should be broken and the reference bound again.

The later binding occurs, the more flexible the system. Furthermore, supporting mobile objects, machines, and applications, necessitates a redirect mechanism that dynamically selects a new target, possibly breaking existing bindings.

This same mechanism supports persistent and fragmented objects. Redirection is transparent to holders of the reference.

We will also show that, for flexibility, binding should be under the control of the target object.

2.3.2 Invoking target

Once a handle has been bound, its target can be invoked, 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.

Dereferencing is inseparable from invocation of the target. Our invocation primitive on a bound handle, invoke, takes a description of a method call of its target, and returns a description of the result.

The base invoke is a local procedure call. Here the operation description is virtual, composed of the address of the called procedure along with an argument stack.[3] In contrast, in the example of a remote procedure call, invoke has an actual representation, i.e. a message containing an operation code and a marshalled copy of the arguments.

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

2.4 Binding instantiates a local atom

The most interesting handle primitive is bind. In this section we will take a look at the basic requirements for binding. Later we will give a complete specification the binding protocol. The client side is specified, in Section 2.6, and the target side in Section 3.2.

When the target is local to the client, binding simply yields its address. Apart from this case, the two well-known cases to consider are a remote object and a persistent object.

When the target is remote, the standard approach is to instantiate a *stub* atom [4] in the client's address space (or find an existing stub for the same target). The reference is redirected to the stub; the client calls the stub's methods

[3] Take myhandle.invoke (f, arg1, arg2) to be equivalent to myhandle->f (arg1, arg2), in a C++-like notation.

locally. The stub hides the complexities of remote calls to the remote target. The stub contains the (network) address of a symmetrical atom, the *scion*.[4] A "stub generator" mechanically generates stubs and scions from interface specifications. A stub method marshalls arguments into a call message, sends the message, and awaits and unmarshalls the return message.

The second interesting case (see left part of Figure 1) is a object that persists, stored on disk, between access by different programs. To prepare for future invocations, bind copies the on-disk atom into a cache, an in-memory atom (or re-uses an existing cache if there is one). It may also set up an access path between the disk and the cache in order to propagate future updates.

Both cases above are special cases of fragmented objects (FOs). Another interesting FO is a replicated object composed of the set of replicas. We call *proxy* the fragment instantiated locally for use by a client [18]. The stub and the cache in the previous examples are examples of proxies.

An FO can implement various policies transparently to its clients. For instance Figure 1 shows a combination of persistence and remote access. The first binding instantiates cache x_m into memory, connected to the disk atom x_d. To avoid cache consistency problems, a later binding (on the right-hand side of the picture) sets up a remote invocation path to x_m through a stub proxy x_s.

A proxy has a dual rôle as object and as reference mechanism. For instance, a stub is a part of the access chain, and it converts a local address to a network address. But it is also an application object. It runs application-level code (the marshalling or cache methods) and doesn't support the handle interface of Section 2.3.

To summarize, the general case of binding instantiates a new proxy atom in the address space of the client (or finds an existing one). The handle returned by bind points to this atom. The access chain may continue further on from the proxy. The client invokes the object by calling the proxy locally. The proxy is free to return locally or to forward along the continuation of the access chain. The proxy and its descendants along the chain are logically part of the same target FO.

2.5 Bind-time operations

There are a number of pre-requisites to invocation. Type checking the reference is one that we mentioned earlier. We have just discussed instantiation of a proxy and setting up the access chain. If the class of the proxy was previously unknown, it is itself instantiated, and the corresponding code linked in. Since

[4] "**Scion** *n. 1. A descendant or heir. 2. A detached shoot or twig containing buds from a woody plant and used in grafting*" [14]. Birell [4] uses the term "server stub."

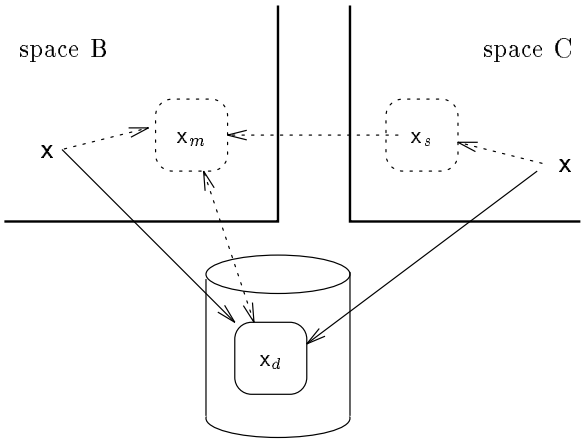

Figure 1: Persistent object example. *When binding B's reference, x_d is copied to x_m, and the reference redirected to x_m. Later, C binds to stub x_s, itself redirected to x_m.*

all these operations are expensive, we propose to execute them only once by combining them into the bind operation.

2.5.1 Dynamic loading and initialization

An object is instantiated by calling the method new of its class with appropriate arguments. We deduce from the previous discussion that the new method of a class for a proxy needs at least two arguments: data to initialize the proxy, and a handle for the continuation of the access chain.

For example, in Figure 1, the cache is created and loaded from disk by calling C_m.new (where C_m is the cache class) with a handle to x_d; a call to C_s.new with a handle to x_m creates stub x_s.

To select the appropriate class, the bind protocol provides a reference to the class object. (This is returned by the target-side binder, as we will see later.)

2.5.2 Dynamic type checking

The dynamic type of a target must conform with the static type of the reference.

Compile-time checking is the most efficient. But when objects are shared across separately compiled modules, the type information not available at com-

pile time. In this case, dynamic type checking is necessary. To amortize its cost, we propose to check early, at bind time.

Dynamic type checking is necessary in any system with dynamic sharing. Even Unix does some checking, e.g. open will fail on targets of type directory; the check is very simple because Unix implements only a small number of pre-defined types. RPC systems such as RPCgen [22] do a rudimentary type check by checking interface version numbers. Object-oriented databases [6] and persistent programming languages [2] store type definitions in a "schema" to check against bindings at run time; the CORBA specification similarly defines an Interface Repository [7].

Two things are needed in order to support bind-time checking. First, the compiler must store the static type reference in handles. Second, the binder must compare this static type with the dynamic type of the target. In our protocol, this is accomplished by new calling an language-specific type checker. The static type reference is a third argument to class method new.

2.5.3 Dynamic linking

It may be necessary to link in code for a previously-unknown class. We assume that this is done by instantiating the class object.

The class is known by the class reference returned by the target-side binder, as mentioned in Section 2.5.1. Binding to this reference will instantiate the class object (just like binding to any reference instantiates a local proxy).

This recursively calls the new method of a class managing classes, class class.

For instance the binding of the reference to x instantiates proxy x_m by invoking C_m.new. This is preceded by binding the reference to C_m, which upcalls class.new (..., C_m, ...). This method loads and links the C_m code and instantiates the C_m object. We will discuss in Section 2.7 how the recursion ends.

To simplify the programming model, we can suppose that binding the class reference occurs by faulting on the call to C_m.new.

The dynamic linking of unknown classes into a running application can be considered a security weakness, opening the possiblity of viruses. For this reason, classes should be loaded only from a trusted class repository, e.g. the "implementation repository" of the CORBA specification [7].

2.6 Binding protocol

We are now ready for an outline of the binding protocol.

The specification of bind follows directly from the previous discussion:

myhandle.bind (app-specific-args) → handle

Bind is a method of a handle. It may have application-specific and/or system arguments, such as an access mode, access control information, concurrency control information, etc. (We do not specify these arguments, which may vary between applications and between objects.) It extracts the static type from the handle, and sets up a path to the target. In a way that will be specified later (in Section 3.2), it passes the static type and the application-specific arguments, gets back a class to instantiate and the corresponding initial data, and calls new as specified next. It returns a handle bound to the instantiated atom.

The interface of a new method of some class myclass is the following:

myclass.new (static-type-ref, next-target-ref, initial-data) → myclass

The first argument is (the reference to) the static type of the reference being bound. New upcalls a language-specific type checker to verify conformity of the instance it is creating with the static type. The second argument is a reference to a further target, i.e. the continuation of this access chain, or null. The last argument is the initial data; its type is unknown by this specification but can be assumed correct and known by the class (since it was supplied with the class itself). Myclass.new() creates an object of class myclass, initialized with initial-data and connected to next-target-ref.

2.7 Ending the recursions

We now examine how the recursions in the binding protocol terminate, and what is the practical meaning of this termination.

The definition of a handle is recursive since a handle contains a type reference for its static type. The apparent impossibility is resolved by using a compact identification for a type, relative to a domain of type objects (and not relative to all objects), e.g. a registration number in a schema or type repository or a hash of the interface.

A handle to a type recursively contains a handle to the type's type. This is meaningful: a type's type might identify one of many schemas or checksum algorithms. The recursion continues if, say, the same system supports both schemas and checksums; then the check selects between these two algorithms. The recursion stops wherever the system designer decides to wire in a static decision. It is doubtful a real system would need any more than one to three levels. A type's type identifier is even more compact that a type identifier.

Grouping makes possible a further simplification. For instance, one could assume that all the handles in a particular module or space use 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.)

Binding instantiates an atom, possibly recursively binding its class reference. This recursion is meaningful, as a class manager might be necessary to link the class, etc. A simple way of stopping the recursion is to bind statically the class class, i.e. to arrange that the dynamic linker is statically linked into the address space. Alternatively, dynamic linking can be avoided entirely if all classes are known statically.

The binding protocol may also iterate. The access-chain continuation reference returned by the target-side binder may itself be unbound, in which case binding starts over. Iteration continues until either a null or a bound reference is returned.

3 Fragmented objects and the target side of the binding protocol

We have already seen two trivial examples of fragmented objects, a persistent object and a remotely-accessed object. A more serious example is a replicated object. A replicated object groups a set of replicas distributed across multiple sites. The object's value may be read from any replica; updates must be propagated to all replicas; concurrent updates and failures pose a consistency problem. Less extreme forms of replication exist, e.g. caching only the most-recently accessed data.

A fragmented object (FO) [13] is a logically single shared distributed entity grouping a number of fragments located on different sites. The fragments communicate by sharing a lower-level FO. Ultimately, the representation of the FO is the distributed set of atoms that make up its fragments and the system-provided communication channel between them. The fragments are not necessarily equivalent; for instance part of the object's data might be local to some particular fragment, whereas some parts may be cached at other fragments.

The appropriate binding policy for a particular FO depends on its intended use. For instance, a versioned-file FO, that encapsulates a set of versions, binds to the most recent version at the time of binding. A replicated-database FO could bind to an available consistent replica, even if not the most recent. An FO encapsulating a group of processes for reliable programming would bind to the current leader of the group.

3.1 Referencing an FO

In order to reference FOs, we consider three options.

In the first option, each fragment has its own reference. Then, by exporting a reference of an appropriate fragment to each client, it is easy to control which client binds to which fragment. However, this is early binding, and fails its purpose if clients pass references to one another. Furthermore, the FO is not accessed as an encapsulated unit.

The second option is to reference an FO as a single unit. When a client binds, the reference system chooses the appropriate fragment to bind it to: for instance, the closest one, or the quickest to respond. Encapsulation is enforced. But the programmer of the FO has no way of distinguishing among fragments; and has no control over a client's binding. A particular FO cannot select its own binding policy. This option is typical of systems that support process groups, such as ISIS [3].

In SOS, a general-purpose system supporting FOs [20], 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. Black's recent proposal [5] is similar. This behaviour is complex and proves confusing to users.

Our current solution is both simple and flexible. Referencing a fragmented object combines referencing a particular "factory" fragment, with the ability by the factory to redirect the reference to another fragment. We now explain this in more detail.

3.2 Binding protocol: target side

Every object has its own binder, the method accept-bind, upcalled by the system as the target side of the binding protocol. A default accept-bind is provided that sets up transparent remote invocation. Alternatively, an object can provide its own implementation of accept-bind. In this way it can set up an appropriate target atom and redirect the reference to it. This mechanism is sufficient to support a large variety of policies.

An FO is referenced through one if its fragments that we call its *factory*. (Any fragment will do, as long as it implements an appropriate accept-bind.) When binding the reference, the system calls the factory's accept-bind. Its interface follows:

myFactory . accept-bind (static-type-ref, app-specific-args)
→ myclass, next-target-ref, initial-data

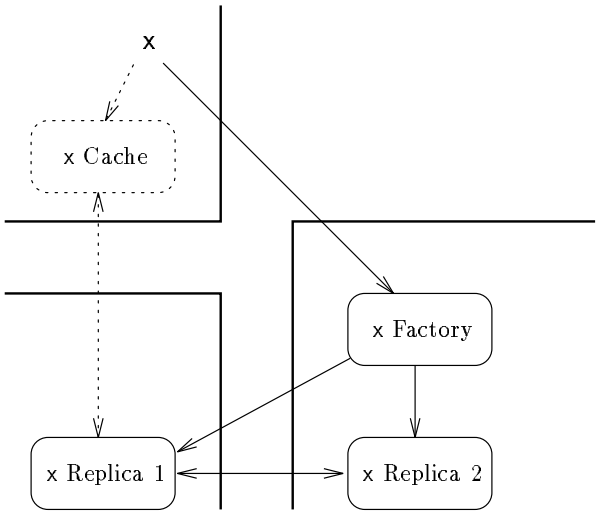

Figure 2: Fragmented object example. *Client has reference to Factory; at binding, factory initializes Cache, redirects client to Cache, redirects cache's reference to Replica 1.*

This method takes as arguments the static type of the reference and any application-specific arguments (as defined in the binding protocol, Section 2.6). It returns what is necessary at the source side to instantiate a proxy atom: its class, the access chain continuation, and initial data.

The static type is not strictly necessary for correctness, since the type check occurs at the source side, but may be useful for the factory to select the appropriate class. The result of accept-bind enables the calling bind to invoke myclass.new passing the reference next-target-ref and some initial data. The reference next-target-ref denotes the selected target (which may be, and probably is for many realisitic OFs, different from the factory).

The system-provided default returns a proxy class for the target, a reference bound to the target, and empty initial data. This supports transparent remote access.

3.3 Example of binding to a fragmented object

Let us illustrate our binding mechanism by an example (Figure 2), inspired by the Coda file system [10]. Consider a fragmented object x. The programmer of x wishes to implement the following fragmentation policy. There is a fixed set of replicas (two in the figure). Each client is assigned a particular replica

in order to balance the load per replica. Furthermore each client accesses a local atom, a cache of the object's recently-accessed data. A client reads data from its cache. An update is written through a cache to its assigned replica; the replica propagates the update to other replicas; every replica propagates back to its assigned client caches. An update may take some time to propagate everywhere. Thus, the state of x is spread over the set of replicas and caches.

We now show how the mechanism proposed above supports the proposed policy. To reference x, we will give out a reference to its factory xFactory. When a client binds its x reference, this will execute the method xFactory.accept-bind(...). The implementation of this method selects the least loaded replica, and creates a cache object co-located with the client. The client's reference is re-directed to the cache (in order to access it locally). The cache's reference is re-directed to its assigned replica (to resolve cache misses and in order to propagate updates). The cache's new calls its replica through this reference in order to pass a reference to itself, for call-backs. If the becomes unbalanced, a client can be assigned to a different replica by redirecting its reference to a factory. The reference is bound again before the next invocation.

The client is not aware of the fragmented and distributed nature of x, and encapsulation is enforced. Similarly, the cache is unaware of the existence of multiple replicas, and the coherency policy is encapsulated within the set of replicas.

In the end, we have built a well-encapsulated fragmented object with a demanding binding policy. This complex construction was obtained by a straightforward application of two simple mechanisms, the factory accept-bind method, and reference redirection. In Section 4 we will see that these mechanisms can be implemented efficiently. In particular the redirections can be short-cut efficiently and reliably.

4 A solution: SSP Chains

We propose a mechanism, based on chains of stub-scion pairs (SSP Chains), that efficiently implements the requirements set forth previously. SSP Chains are used, managed, updated, and deleted by well-defined protocols. We will describe only briefly the features of SSP Chains; a complete specification can be found elsewhere [19].

Rather than of imposing a single identification mechanism across the universe (such as unique identifiers [11]), we adopt a flexible and efficient approach, accomodating the multiple, efficient native addressing schemes, and providing translation at domain borders.

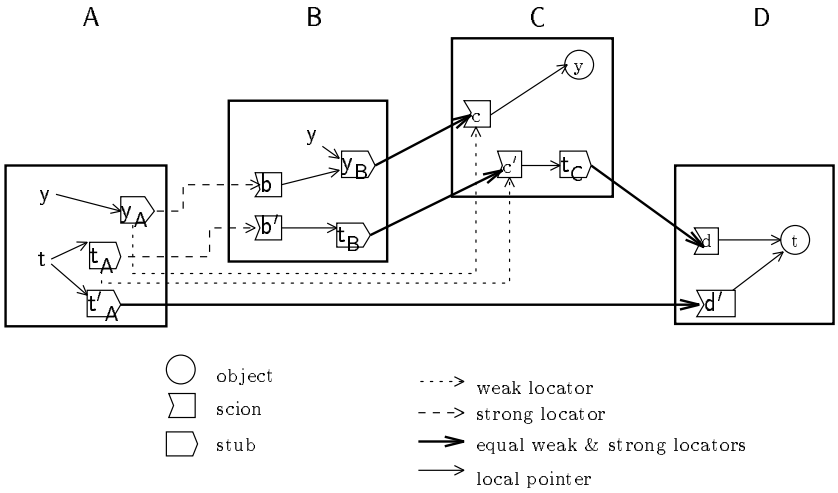

Figure 3: SSP Chain. *A reference to* y *has been sent to space* B*, then from there to* A*. Similarly, a reference to a* t *atom located in* C *was sent to* B *and* A*. This reference was subsequently redirected to a different* t *atom in space* D*. Binding it yields the direct chain through* t'_A *and* d′*.*

4.1 Summary of SSPs

A handle always contains a memory address. If the target is local (and this is expected to be the common case), the handle points to it directly. If it is in a different space, it points to a stub. Redirecting a reference just changes the handle to point to a different stub.

Sending a reference marshalls the pointer into a message; marshalling creates a scion (or selects an existing one). Unmarshalling creates the matching stub (or selects it), and returns a pointer to it. The remote reference is implemented by this Stub-Scion Pair (SSP). Passing a reference via multiple spaces creates a chain of SSPs with a pointer from each intermediate scion to a local stub, hence the name SSP chain.

A stub contains a "locator" for two scions, called the strong and weak locators. The strong locator contains the network address of the matching scion (the one in the space that sent this reference). The weak one contains the address of a scion closer to the target, if one can be known without extra messages (the one in the space that sent the reference initially).

A stub is either bound or unbound. An unbound stub supports only the single operation bind.

Binding an SSP Chain calls the accept-bind method of its target. This returns a continuation reference, either null, bound or unbound; in the latter case, binding iterates until a null or a bound continuation is returned.

A bound stub is fully typed, i.e. supports the interface of the static type of the reference. In a bound stub, the weak and strong locators are equal. A binding can be broken at any time, either by the source or the destination, forcing the source to rebind before the next invocation.

Distributed garbage collection relies on the invariant that (in the absence of crashes) there is an uninterrupted chain of strong locators between the source and target. Weak locators are used for communication and location, which rely on the invariant that the weakly indicated scion will not be collected, being protected by the strong chain.

4.2 Examples

We will illustrate SSPCs by two examples, shown in Figure 3. The remote reference to y located in space C, held by space B, is duplicated to space A; the strong chain will pass indirectly through space B. The weak locator will, however, point directly to a scion in space C. Note that the stub matching that scion lies on the strong chain and is in B, not A. The SSP Chain invariants ensure however that this scion will not be collected as long as a weak locator uses it.

Now consider redirection. Suppose an atom of t is located in space C; space A acquired a reference to t via space B. To redirect the reference to a t atom located in space D, all t pointers in C (in particular scion c′'s pointer) are redirected to the stub t_C. At the next use of the redirected reference the protocols will return an exception to the caller (with the location of the current target) forcing the caller to bind again.

The liberal use of indirect SSP chains makes it cheap to pass references in messages, while retaining useful invariants.

Two stubs received from different destinations are not directly comparable, even though they may refer to the same ultimate target. For instance, in Figure 3, two stubs in space A, t_A and t'_A, both refer to t in D. They cannot be compared or merged because the former's strong locator identifies space B whereas the latter identifies D. However, when t_A's long chain will be eventually short-cut, both references will use t'_A.

4.3 How binding short-cuts long chains

Binding sends a call message is sent along the weak locator, and the response carries the location of the actual target. This location is unmarshalled into a new stub; the one used for the call becomes obsolete. This, in Figure 3, binding the indirect reference to t from space A, through stub t_A, replaces it with the direct connection through stub t'_A.

Obsolete stubs and scions are cleaned up by the garbage collector. (The garbage collection protocol is described elsewhere [21]). If the new reference has been optimized to a single SSP, then binding is complete. Alternatively, the target may redirect the reference by returning a location to a scion located elsewhere; in this case binding iterates.

This description assumes that there is a direct connection between any two spaces, and that the space address suffices to identify the appropriate connection. Extensions to more complex configurations are out of the scope of this paper.

5 Conclusion

We have examined the requirements of references for objects in a large-scale distributed computer system. From these requirements we have derived a number of insights, which we now recapitulate.

Binding a reference yields new handle pointing to a local atom, which in turn may forward invocations along the access chain. Binding is a recursive mechanism that executes under the responsibility of the reference system but also upcalls user-level methods new and accept-bind. Flexibility requires that a binding can be redirected or broken.

Binding can cause dynamic type checking, dynamic linking of a class, and dynamic loading of the target atom. The general binding protocol specified supports, we believe, a large and useful variety of policies; we also pointed out where and how this abstract protocol may or must be simplified in order to implement real systems.

Fragmented objects, such as a replicated object, pose a difficult problem to the reference system. Our solution supports both encapsulation and user-defined binding, by combining references to fragments with redirection, under the control of the target object.

The mechanism of SSP Chains implements the above requirements efficiently.

The work described here is part of the specification of Perrault, an operating system framework for the distributed support of objects.

Acknowledgements

This work was supported in part by Esprit Basic Research Action "Broadcast," contract number 6370. Thanks to Karen Sollins for help in locating the relevant bibliography. Many thanks to Hank Levy and Laurent Amsaleg for their constructive criticism on an early draft.

References

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

[2] M. Atknison, P. Buneman, and R. Morrison. Binding and type checking in database programming languages. *The Computer Journal*, 31(2):99–109, 1988.

[3] 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.

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

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

[6] O. Deux et al. The O_2 system. *Communications of the ACM*, 34(10):34–48, October 1991.

[7] 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.

[8] 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.

[9] A. Goscinski. Naming facility. In *Distributed Operating Systems: The Logical Design*. Addison-Wesley, 1991.

[10] James J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, volume 25 of *Operating Systems Review*, pages 213–225, Pacific Grove CA (USA), October 1991.

[11] P.J. Leach, B.L. Stumpf, J.A. Hamilton, and P.H. Levine. UIDs as internal names in distributed systems. In *1st Symp. on Principles of Distributed Computing*, pages 34–41, Ottawa (Canada), August 1982. ACM.

[12] Barbara Liskov and Rivka Ladin. Highly-available distributed services and fault-tolerant distributed garbage collection. In *Proceedings of the 5th Symposium on the Principles of Distributed Computing*, pages 29–39, Vancouver (Canada), August 1986. ACM.

[13] Mesaac Makpangou, Yvon Gourhant, Jean-Pierre Le Narzul, and Marc Shapiro. Fragmented objects for distributed abstractions. In T. L. Casavant and M. Singhal, editors, *Readings in Distributed Computing Systems*. IEEE Computer Society Press, July 1993.

[14] William Morris, editor. *The American Heritage Dictionary of the English Language*. Houghton Mifflin, Boston, 1980.

[15] R. M. Needham. Names. In Sape Mullender, editor, *Distributed Systems*, ACM Press, chapter 5, pages 89–101. Addison-Wesley, 1990.

[16] Jerome H. Saltzer. On the naming and binding of network destinations. In P. C. Ravasio, G. Hopkins, and N. Naffah, editors, *Local Computer Networks*, pages 311–317. IFIP, North-Holland, 1982.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] Marc Shapiro, Olivier Gruber, and David Plainfossé. A garbage detection protocol for a realistic distributed object-support system. Rapport de Recherche 1320, Institut National de la Recherche en Informatique et Automatique, Rocquencourt (France), November 1990.

[22] Sun Microsystems. *Network Programming*, May 1988. Part Number 800-1779-10.

[23] Paul R. Wilson. Uniprocessor garbage collection techniques. In *Proc. Int. Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, Saint-Malo (France), September 1992. Springer-Verlag.