DISTRIBUTED ENVIRONMENTS BASED ON OBJECTS:
UPGRADING SMALLTALK TOWARD DISTRIBUTION

A. Corradi, L. Leonardi, M. Zannini

Dipartimento di Elettronica, Informatica e Sistemistica - Università di BOLOGNA
2, viale Risorgimento, 40136 BOLOGNA, ITALY - E-mail: Boari@BODEIS.cineca.it

ABSTRACT
The paper describes a Distributed Environment based on Objects (DEO). DEO configures an environment split in several contexts. The requirements of DEO are: transparent sending of requests to remote objects that reside in different contexts, possibility of replicated objects in each context and migration of objects from one context to another. DEO provides several kinds of replicated objects with regard to their consistency protocols. Consistency is based on the reconsideration of the equality relationship. The goal of the described implementation is compatibility with Smalltalk. No essential modifications to the standard virtual image are introduced. The interpreter has not been changed at all. Only a few primitives have been added to the virtual machine.

1. INTRODUCTION
Distributed applications are more complex than non-distributed counterparts. However, distributed environments are an area that has been and must be considered, both to face new requirements and to handle with efficiency the resources a distributed architecture can make available.
The object methodology has been adopted for solving problems in the distributed field mainly for dependability sake. Argus [17], EDEN [2], ISIS [6], Concordia [10] are examples of this trend.
While the object paradigm is still under definition, it seems to include the reconsideration of the equality relationship. The goal of the described implementation is compatibility with Smalltalk. No essential modifications to the standard virtual image are introduced. The interpreter has not been changed at all. Only a few primitives have been added to the virtual machine.

The paper describes a distributed environment based on objects: Distributed Environment Objects (DEO). DEO consists of several extended Smalltalk machines residing on different nodes in a network. In particular, the project guideline is the compatibility with the existing Smalltalk services with the goal of transparent use of existing functionalities. DEO stretches to the limit the concept of compatibility with the Smalltalk environment to determine the areas where (possibly minimal) incompatibilities must be inserted.

2. DEO: A DISTRIBUTED ENVIRONMENT BASED ON OBJECTS

2.1 Object taxonomy
The paper deals with distribution in accord with a protected domain approach [14]. The total object space is divided into different contexts [11]. Each context can either correspond to a single machine or more contexts can be grouped into a single machine. An object (generally) belongs to a specific context. Within one context, the communication is the usual adopted by Smalltalk: we call it intra-context message passing.
Of course, the communication can take place even from one context to another. This happens via inter-context references. References to objects in other contexts are via proxy objects that locally stand for the remote object [11]. An object can request a service of an object residing in another context only if it knows the remote object via a local representative, a proxy (see figure 1).
A context space offers:
- local objects, i.e. objects that reside in the same context;
- proxy objects. They stand for objects that reside in other contexts and act as keys to other contexts.

Note that there is one and only one proxy object for a specified remote object within a context (proxy/context rule).
- replicated objects. Objects are replicated when a copy of them is present within each context. Of course, replicated objects do not belong to a specific context. Replication introduces the need of recognizing copies of the same object. At object creation, the copies must be identified as part of the same replicated object [4], [6]. In the following, the problem of partially replicated objects also emerges. The abstract services any machine furnishes motivate the existence of replicated objects. All the objects that map local resources, e.g. I/O handlers, file systems, etc., are examples of replicated objects. They must exist within any context to make the resources available. Moreover, in any Smalltalk environment, there are peculiar resources, such as the hierarchy of standard system classes that we can consider as replicated objects.
The proxy mechanism does not introduce any load on the intra-context message passing. Only objects that are engaged in inter-context communications can experience delays. Replication can sometimes transform services that are local to a context into services local to the server context. Moreover, the migration of objects can sometimes reduce overhead (due to non-local operations) by moving objects that heavily communicate to the same context of residence.
consistency protocols. A reference to a replicated object must be collected. It has been described in [20]. This issue is beyond the scope locally found) or a local object must be identified in the server effective practice favours local communications whenever they are reached via a local table. Then, it cannot be prey cated and neglect the deallocation issue. The deallocation phase can trasparently forward the request for an object operation from the client context. They reference must be translated and adapted to the new context. Either proxies are found locally in the server respectively. This makes worth to analyze the parameter passing phase.

By the use of proxy, inter-context communications can be handled transparently. Of course, they cost more than local ones. A cost-effective practice favours local communications whenever they are possible and recurs to inter-context message passing only when it is necessary. There is a context switch for any inter-context communication. This switch implies different context visibilities, of the client and of the server respectively. This makes worth to analyze the parameter passing phase.

In an inter-context communication, one can pass as parameters:

- Local objects. Their validity is strictly intra-context with reference to the client context. They references must be translated and adapted to the new context. Either proxies are found locally in the server context or, if they do not exist yet, they must be created.

- Proxy objects. For any remote object a reference valid in the server context must be produced. Either a proxy must be created (or again locally found) or a local object must be identified in the server context.

- Replicated objects exist in any context, but can follow different consistency protocols. A reference to a replicated object must be dealt with in a special way (see 2.4).

2.3 Object Identity and Equality

Within a distributed object framework, there is a need of reconsideration of some primitive operations. These operations are closed related to the identity of objects. We are interested in the identity and the equality relationships [16], [8]. The identity relationship ( = = ) states whether two variables refer to the same object. Let us consider two variables a and b within a given object. As a matter of fact, we may use the Smalltalk definition: the identity relationship a = = b holds if a and b refer to the same (local) object.

Even in a distributed object system, the = = relationship is always solved locally. The above definition is valid also when a and b refer to proxies. Only if a and b refer to the same proxy, they refer to the same remote object and vice versa: this because of the proxy/context rule.

In a distributed object environment, the relationship of objects may behave differently than in standard Smalltalk. In fact, the equality relationship is established on the basis of the contents of the two objects. It cannot be generally solved within a single context [16]. To produce a correct equality relationship, we must compare internal object components. An object is composed of two parts:

- the behavior, i.e. the description of its state and of its operations.

This part is really contained in the class of the object and its superclasses.

- the state, constituted by the values of variables described by the behavior part. In general, a state variable can contain either a primitive value (e.g. an integer) or a reference to an object.

\[
\begin{align*}
&\text{OBJECT} \quad \text{state description} \\
&\text{behavior} \quad \text{operations}
\end{align*}
\]

Let us consider two objects, O1 and O2, in two different contexts. The object O1 is equal to the object O2 iff both their behavior and their state are equal. This definition is stronger that the need, but it can be of use in migration (see 2.6).

\[
\text{(DEF.1) } O1 = O2 \quad \text{iff} \quad \text{state} \quad (O1) = \text{state} \quad (O2)
\]

Equality of behavior requires that the objects have both the same state description and the same operations. Equality of the state implies that each of its variables contains the same value. Two state variables have the same value iff:

- either they contain the same primitive value (e.g., the integer 12); or
- they refer to the same object. In case this object is replicated, copies of the same replicated object must be referred (e.g., the replicated object Rep03; see in the following). In all other cases, the same object must be referred. This can happen either by having a direct reference to it or via proxy objects, depending on the context. This definition extends the shallow equality of the Smalltalk environment [12]. Within one context, the semantics of the Smalltalk = is preserved.

For our purposes, two objects can be tested for equality only if they have equal behavior. This means the immediate difference of objects whose object descriptions are different. Then, the following constraint holds:

\[
\text{(1) state} \quad (O1) = \text{state} \quad (O2) \quad \text{iff} \quad \text{behavior} \quad (O1) = \text{behavior} \quad (O2)
\]

Therefore, equality of behavior of two objects requires the equality of instance description parts at the class level. The instance description is part of the state of the class, when the class itself is considered an object. For the constraint (1), two classes can have equal state only if they have equal behavior, and so they are equal. Then, two objects can be equal only if their classes are equal:

\[
\text{(2) } O1 = O2 \quad \text{iff} \quad \text{class} \quad (O1) = \text{class} \quad (O2)
\]
2.4 Replicated Objects

An object is replicated if each context owns a copy of it. Copies are always recognized as part of the a single replicated object.

We define consistency protocols for replicated objects, depending on the relationship of equality specified. The kind of replicated object depends on how equality is imposed, either to both object parts or to one part or to none. Then, we deal with three relationships: TCRep (tightly consistent), CRep (consistent) and LCRep (loosely consistent) replicated objects.

The relationship TCRep provides close copy consistency, i.e. equality of both behavior and state, for the copies of the same replicated object. Two copies, A and B, of a replicated object are TCRep if A is equal B:

(DEF.1) \( \text{TCRep} (A, B) \iff A = B \)

The relationship CRep implies that copies of the same replicated object are consistent in behavior, but may assume different state values.

(DEF.2) \( \text{CRep} (A, B) \iff A \Rightarrow B \)

The relationship LCRep allows the greatest copy independence.

(DEF.3) \( \text{LCRep} (A, B) \iff A \Rightarrow B \) \& \( \text{behavior} (A) = \text{behavior} (B) \)

In DE0, replicated objects can belong to three different categories:

- tightly consistent replicated objects (TCRep objects);
- consistent replicated objects (CRep objects);
- loosely consistent replicated objects (LCRep objects).

A replicated object belongs to the category of TCRep objects if the TCRep relationship for its copies is maintained for all their lifetime. Consistent CRep objects are expected not only to have in accord with the same behavior, but also to assume the same state values. This requirement is important in case one wants to employ replication to face site crashes [10]. The implementation of this consistency can be solved in distributed environments by using complex synchronization and communication protocols [6].

A replicated object belongs to the category of CRep objects if any copy can change independently its state, but any change of behavior is propagated to the other copies. This propagation is done before any other operation can take place. Then, two copies of a CRep object, A and B, can either be CRep(A,B) or be in a closer relationship, i.e. TCRep(A,B).

A replicated object belongs to the category of LCRep objects if copies can update state and behavior independently. The updating of one copy does not affect other copies. Loosely consistent objects must not be affected by the behavior change. There is, however, no requirement of consistency. Therefore, two copies of a LCRep object, A and B, can be LCRep(A,B) or CRep(A,B), or TCRep(A,B).

In a DE0 environment:

a) Only a few system classes belong to the TCRep category. This requirement becomes important in case of object migration. The class Object, for example, must be a TCRep class. This is because all objects in a DE0 environment must have behavior by default at the same time. One of such classes generally updates itself at a slower rate than other ones. Then, the protocol to maintain consistency among copies is rarely activated. Let us note that DEO does not deal explicitly with the event of node failures as in [4], [6], [10].

b) All objects that model the user interfaces and, in particular, I/O devices belong to the CRep category. In fact, in each context the state can assume different values, but the behavior ought to be the same. Each keyboard, for example, maintains the same operation in any context, but the state is different depending on the user interactions.

c) LCRep objects can better adapt themselves to tailor each copy to its specific context of belonging. For instance, a printer can have different state and behavior depending on the context: in fact, a printer can own a "laser printer" that furnishes services different from the one of "normal" printer.

Let us note that all replicated objects are created in a TCRep relationship (see 2.5). They can evolve toward less tight consistency depending on the category of replication and on separate evolution of contexts. There is a relationship between the consistency of objects and of their classes. Replicated objects can only be created by replicated classes. In fact, each context, a replicated object must exist and so its behavior. Then, their class must be replicated in all contexts. Of course, replicated classes can also generate single (non-replicated) objects.

Copies of replicated objects can be TCRep or CRep objects only if their classes are equal and do not diverge:

TCRep \( (A, B) \) / CRep \( (A, B) \) = \( \Rightarrow \) class \( (A) = \) class \( (B) \)

Two copies of the same replicated object are TCRep if their states and behaviors are equal. These parts are described in the state of their classes. Then, their classes must have equal states. This for constraint (1) implies that they have equal behavior, i.e. they must be equal (apart from being replicated).

LCRep objects do not impose any constraint on their classes apart from being replicated. In fact, two copies of the same replicated object can be LCRep if their state and behavior are not equal. The description of instances at the class level can also be different. One can assume a negative constraint. Truly LCRep objects cannot be created from TCRep classes, because they could not vary independently their description part.

Replicated objects are recognized as such by any context. This recognition means a peculiar treatment when passing one such reference as a parameter. Replicated classes can structure a subtree starting from the class Object. In particular, the class LCRep inherits from the class Object. The class CRep derives from LCRep, and the class TCRep from the class CRep.

2.4.1 Passing Replicated Objects. Independently of the above distinction based on consistency, there are two kinds of replicated objects with regard to inter-context communications [20]. This categorization is important both for the request phase and for the parameter passing phase:

- translation-independent replicated objects. They represent the services local to any machine. Any context furnishes its own service.

- forwardable replicated objects. Despite of the principle of finding the more effective cost solution, there are services connected to a specific site, independently of the context (possible remote) where they have been requested. Then, the request is pinned down to its initial resources and it must be maintained independently of the number of nodes the request itself is visiting. A console object in a remote context is forwardable to make possible the communication of results to the user at the starting context.

An object within a context has always a single reference for translation-independent replicated objects and possibly several references to a forwardable replicated object.

When a reference to a forwardable resource is used or passed, it is dealt with the same as if it were a local one. A proxy must exists on the node of execution (or must be created) that refers to the object on the source node. In case of a translation-independent resource, the one local to the current context must be found and referred.
The behavior of an object is factored in several classes connected superclasses in the same context. Because of the inheritance relationship, the behavior is contained in its class. Then, any object and its class must belong to the one in a concentrated domain. An object here must always reside in the context of belonging of its class. An object has its behavior contained in its class. Then, any object and its class must belong to the same context. Because of the inheritance relationship, the behavior of an object is factored in several classes connected by a superclass relationship. Then, all involved classes must reside on the same node of the object. We call that the co-residence rule.

The create request produces an inter-context communication and the creator (e.g., Cl), the requestor must have a proxy for the class. In DEO, the class/instance relationship has a stronger meaning than the one in a concentrated domain. An object here must always reside in the context of belonging of its class. An object has its behavior contained in its class. Then, any object and its class must belong to the same context. Because of the inheritance relationship, the behavior of an object is factored in several classes connected by a superclass relationship. Then, all involved classes must reside on the same node of the object. We call that the co-residence rule.

The create request produces an inter-context communication and the new object is created in the context of Cl, because of the co-residence rule. The creator Cl receives - as the result of the creation - a proxy to refer the created object. In case Cl is a replicated class (and a local copy of C exists in the creator context), the distinction between translation-independent and forwardable replicated classes affects the creation. If the class is translation-independent, the creation is locally requested. In case the class is forwardable, the requestor must own a proxy for a copy of the class in another context as in the first case. The object is then created in the remote context.

Note that creating from a replicated class does not imply the replication of the object. There are replicated classes whose instances are single objects: for example, already-defined system classes from which one still wants to create single objects. We recall that a class completely specifies the behavior of an object. In particular, a class also describes whether its instances are replicated objects and of which kind. TCreP, CREP or LCRep.

The creation of a replicated object needs a creation of a copy in each context. A correspondence table must exist in any context to correctly associate copies of the same replicated object. This table is used by the consistency and the inter-context message passing protocols.

2.5 Object Creation

In DEO, the class/instance relationship has a stronger meaning than the one in a concentrated domain. An object here must always reside in the context of belonging of its class. An object has its behavior contained in its class. Then, any object and its class must belong to the same context. Because of the inheritance relationship, the behavior of an object is factored in several classes connected by a superclass relationship. Then, all involved classes must reside on the same node of the object. We call that the co-residence rule.

The create request produces an inter-context communication and the new object is created in the context of Cl, because of the co-residence rule. The creator Cl receives - as the result of the creation - a proxy to refer the created object. In case Cl is a replicated class (and a local copy of C exists in the creator context), the distinction between translation-independent and forwardable replicated classes affects the creation. If the class is translation-independent, the creation is locally requested. In case the class is forwardable, the requestor must own a proxy for a copy of the class in another context as in the first case. The object is then created in the remote context.

Note that creating from a replicated class does not imply the replication of the object. There are replicated classes whose instances are single objects: for example, already-defined system classes from which one still wants to create single objects. We recall that a class completely specifies the behavior of an object. In particular, a class also describes whether its instances are replicated objects and of which kind. TCreP, CREP or LCRep.

The creation of a replicated object needs a creation of a copy in each context. A correspondence table must exist in any context to correctly associate copies of the same replicated object. This table is used by the consistency and the inter-context message passing protocols.

2.6 Migration

An object migrates when it moves from one (source) context to another (destination context). Only non-replicated objects can migrate. In the following, we neglect the reasons that may lead to the decision of the migration. The number of movements, however, must be minimized to preserve efficiency. The strategies to avoid any thrashing are not subjects of the paper [15].

Object migration introduces the necessity of moving object information around. Again, one can distinguish state and behavior. Recall that the state is locally enclosed within the object, while the behavior is kept in its class and superclasses. The easiest case is when one must move only state information and the behavior is already present in the destination context (see figure 3). The object state must pass from one context to another. At the destination context, the migration of an object forces the creation of a new object that will receive the migrated state. This easiest case happens iff the class (and its superclasses) are replicated objects.

In case of a TCreP class, there is no problem. In all other cases (i.e. CREP or LCRep classes), it is necessary to verify consistency. In case of class inconsistency, the migration can simply be refused.

Consistency algorithms can be of various levels:

- CREP classes are tested for their states, in particular their descriptive part. One may start with controlling the number of instances variables and operations as in [5]. Then, class variables can be examined.
- Even inconsistent classes (e.g. LCRep classes) can be accepted: the classes in the source and the destination contexts can have a different number of instance variables. A new object can be created at the destination context even when there is a larger or smaller number of instance variables. Of course, this migration can cause loss of information when an object transits from one context to another. More complex is the case when one must pass also the object behavior contained in its related class and its superclasses. Of course, one could decide to avoid the migration of the object in the destination context when the behavior is not present. This decision can be taken for efficiency sake, but alternative policies are viable:
  a) Class Copy. The class must be copied to the destination context. This introduces classes replicated only in two contexts. Partially replicated objects require to be suitably dealt with.
  b) Class Movement. The class must be moved to the destination context. A class itself an object; so, it can migrate. This solution is recommended only if the migrating object is the only instance of this class. In fact, if that is not the case, the co-residence rule would imply a migration of all other instances.

Both solutions may recursively introduce the need of migration of all superclasses. Mixed solutions can even be preferred: for example, the direct class - of the migrating object - migrates; its superclasses are copied and then becomes (partially) replicated classes. Class migration implies the same actions described for objects. In particular, in a Smalltalk environment the migration of a class is simpler than the migration of a generic object. The tight connection of a class/metalclass pair precludes any propagation of the migration to any other object level.

DEO keeps migration transparent for the clients of migrated objects. When an object migrates, the source context creates a proxy that refers the migrated object. This serves to keep migration transparent to all local - to the source context - objects that refer the migrated object. Similarly, the references contained in the migrated object must be correctly translated. The rules for this translation are the same as for parameter passing.

All other proxies for the object must be requalified. In particular, if the destination context - where the object migrates to - contains a proxy for the migrated object, it must be removed. If an object O1 (in the context ctx1) has a proxy for an object O2 (in the context ctx2) and O2 migrates in another context (for example ctx3), the source context ctx2 requalifies all proxies for O2 by sending to all contexts the new location. When O1 refers to object O2, the latter is consistently found via the requalified proxy. The proxy in the source context ctx2 serves for local references to the migrated object O2. The migration can be considered completed only after the broadcasting of the requalification message.
ferred. The processes executing within the migrating object must be favored. Only when all the processes have completed their actions, A possible solution in Smalltalk is to explore all existing processes A peculiar problem

Figure 3. Migration of object O1 from context1 to context2. A peculiar problem is if there are processes executing within the object that is being migrated. The existence of processes actively executing on one object can hardly be revealed within a traditional passive object environment as Smalltalk.

A possible solution in Smalltalk is to explore all existing processes (and method contexts) for a reference to the object itself. If any, apart from the current process, is found, the migration must be deferred. The processes executing within the migrating object must be favored. Only when all the processes have completed their actions, the object can be safely moved. This detection can postpone migration for long.

The solution of this problem is easier for:
1) active objects [1], [2], [3] that can decide how and when to execute operations. An object can block out operations until its migration is completed. Active objects can more easily moved and transferred because they may decide when and who to serve.
2) passive objects organized to serialize and block out operations. Objects can have lock functions, executed before any operation. These objects serialize the executing operations.

1. DEO IMPLEMENTATION

The DEO prototype has been implemented on SUN2/SUN3 workstations under Unix 4.2 BSD (with System V facilities). The Smalltalk system considered for the modification is the version of Smalltalk developed at the Berkeley University.

The goal of the DEO implementation is compatibility with the Smalltalk support. In particular, the standard virtual image has been minimally changed: only few additional methods have been inserted in the class Object and slight changes have been introduced to the inspector methods. The DEO virtual machine has several new primitives written in C, i.e. in the same language in which the virtual machine is written. These primitives mainly support network communications. No modifications to the interpreter has been necessary. The introduction of proxies makes transparent the use of remote resources and does not load the local operations. This guideline has been respected by any implementation of DEO.

3.1 Inter-Context References

A proxy represents a remote object within a context. DEO implements proxies as instances of the (new defined) class Proxy. A proxy must always forward requests of operations to the remote object it represents. Then, a proxy must not recognize any request (apart from few exceptions).

The state of a proxy is constituted by three instances variables (residence-host, class name and remote-address) capable of forwarding a message to the remote object. The class Proxy redefines the doesNotUnderstand: method, following the scheme suggested in [18] and [5]. In the standard Smalltalk virtual image, this method is furnished in the class Object to raise an error when an object receives a request impossible to serve. Then, when a method request reaches a proxy, the interpreter always invokes a method of name doesNotUnderstand: in DEO, the code of the doesNotUnderstand: method is in this case found in the class Proxy.

The class Proxy defines only a few methods that must be executed locally for debugging and inspecting purposes. Note the = means the test for the same proxy (or object) within one context. All other messages are forwarded to the remote object, with the advantage of a completely transparent forwarding. The result is also transparently given back to the client that waits for it.

In fact, when an object O1 sends a message to another object O2, it is not aware if O2 is a local or remote object (transparency). In case O2 is remote, O1 must own a reference to the O2 proxy in its context. O1 can possess this reference because of a previous communication.

An object O2 can initially export a reference in another context by using the message exportAsProxy:, added to the class Object. This operation has the effect of creating on the specified host a proxy for the exported object. Of course, duplication is avoided, by checking if it already exists.

At the remote site, the export operation inserts the proxy identification in a table, ProxyTable, that registers all the proxies of the specific context. This table is used:
- to avoid local garbage collection of proxies;
- to find an already locally existent proxy for a specific object (and translating to it, to avoid duplication).

At the exporting site, a similar table is defined, GlobalTable. It keeps track of remote referenced objects, so that they are not mistakenly destroyed by the local garbage collector, when remote references are still possible. The current prototype does not alter the strategies of the Smalltalk garbage collection.

3.2 Inter-Context Messages

Any communication that involves a proxy creation is dealt with in the same way as if it has been described in the implementation of the exportAsProxy: method.

3.2.1 Client Site. When a proxy P receives a request, it executes the doesNotUnderstand: method. The code forwards the request to the remote machine and waits for the coming back of the result.

The proxy P sends the message to the remote host by employing the
SUN RPC mechanism [19] (see figure 4), after a packing of the parameters.

DEO deals with the parameters of a remote request as follows. First, the proxy \( P \) must inspect the parameter type. A parameter can refer to:
- an object local to the client context;
- an object of another context, via a local proxy;
- a replicated object.

As matter of fact, there is a fourth kind of parameter, primitive values. DEO considers primitives: instances of SmallInteger, LargePositiveInteger, LargeNegativeInteger, Float, Character, UndefinedObject, True and False.

A proxy \( P \) translates any parameter in a triplet made of residence host, class name and local address:
- When a parameter is a proxy, it is dealt with at the client site to avoid a level of indirection at the server site. For this reason, the variable that identify the proxy parameter are read and packed in the remote message.

DEO processes similarly the forwardable replicated objects. For translation-independent replicated objects, instead, DEO states the residence host as the server site. The replication type is founded in a table, RepPolicy.

The proxy \( P \) packs all information to be given to the SUN RPC server: the remote object identity, the selector of the request, and all parameters. The SUN RPC is made available to the Smalltalk machine via a newly defined primitive.

A proxy \( P \) waits for the result on a (System V) private message queue (see figure 4), on which the remote object will send the answer. The binding between queues and proxies is established only on need. Allocation and deallocation operations of a private queue take place respectively before sending a remote message and after extracting the corresponding result.

3.2 Server Site At the remote site, the arrived RPC message is inserted in a dedicated system queue. A Unix background process (called Unix server, see figure 4) acts as a message collector in charge of accepting requests from the network.

Requests are extracted by a Smalltalk process (from now on called the BS process, see figure 4) that polls the system queue. The BS process unpacks the items part of the RPC message. Crucial operation is the creation of proxies, when needed. A parameter can also be a reference to an object local to the server context. On the other hand, a proxy for the parameter can be already present in the server context. In this case, the translation occurs by searching in the local ProxyTable for the local reference. In both cases, the BS process translates any parameter to the correct reference.

The creation of a Performer process avoids deadlock problems. In fact with this solution an agent can appear both as a server and a client in a chain of remote requests. The simplest example that involves only two objects is as follows: an object \( O1 \) requests an operation to a remote object \( O2 \). \( O2 \) requests a service of \( O1 \) in its turn; this service requires again an operation of \( O2 \). In case the \( O2 \) site would have employed the BS process not only to extract from the system queue, but also to serve requests [18], the BS process would be waiting for the result from the remote object \( O1 \). But no answer could arrive to \( O1 \) because the entire \( O2 \) context would be blocked. Any Performers is in charge of yielding back the produced result. This result is treated in the same way as a parameter in a remote request. Moreover, the same RPC low-level mechanism is used to send back the result message to the client site.

At the client site, the Unix server must accept the message that contains the reply from the network and inserts it in the specified private queue which the proxy waits from.

3.3 Migration

Object migration can only apply to non-replicated objects. Replicated objects, in fact, either translation-independent or forwardable, are present in any context.

3.3.1 Object Migration. The first step we have considered is the migration of an object whose class is present in the destination context. The class is then, replicated. Peculiar class consistency protocols have been neglected in the first prototype. Classes are then assumed either as TCRep or simply CRep or LCRep but with their state part still equal.

An object migration operation, called migrate:, has been inserted in the class Object. It can be explicitly invoked by any object that intends to change its allocation from its current context (source) to another context (Destination) specified by using any object resident in the destination context.

The migration requires treatment both at the source context and at the destination one. Migration does not introduce different network-related tools but completely reuses the already described servers.

3.3.2 Source Context. At the source context, the migration exploits the same mechanisms used for the remote object requests. It implies:
1) to treat the object state values according to same transfer protocols used for parameters. For instance, proxies are translated into references to their remote represented object. The translated state values are packed and the resulting migration message is sent to the destination context. A reply is waited for on a private queue. The reception of a positive answer means that the remote context has correctly created an object copy and consequent actions can occur. It is then possible to destroy the object on the source context. A negative answer (a time-out can be devised) blocks any further attempt of migration: the object location is kept unchanged and no destruction takes place at the source context.

2) The source environment must be able to refer to the migrated object. Then, the source context creates a proxy for the migrated object. To obtain this exchange of roles, we use the become: method (defined in the standard class Object). Then, the old object can, at the end of the migration, become prey of the local garbage collector.

3) Any other context must be able to find the migrated object. The new object location must be announced to all remote contexts that possess a proxy for it. A message that specifies the old proxy contents and the new one is broadcasted to all contexts. Only when they have all acknowledged the message and changed their ProxyTables, the requalification of all proxies of the migrated object can be considered complete as well as the migration phase.

3.3.3 Destination Context. A migration message is received by the BS process. When one such message is detected, a different process is created, a Migrator. Like the Performer, one Migrator is created for any migration phase. Its duty is to check for class compatibility.

Then, a new object of the compatible class is created. Its state is filled in according to the contents of the migration message. References to translation-independent replicated objects are translated to refer to the local copies, while objects local to the destination context do not need translation. Proxies for non local objects must be created. (Again the treatment is the same for remote message reception).

As the last step of migration, the new object must be available even for references local to the destination context. To shuffle the references for the migrated object and its proxy. Note that a proxy is created, if it does not exist yet. This has the effect of garbage collecting the old proxy after the updating of the local tables, ProxyTable and GlobalTable.
3.4 Class Migration

Any class can migrate by following the above sketched protocol. In particular, for the class state part, the instance variable names and the instance method code must be completely transferred, so to be used by instances. Then, the class must be inserted in the local class graph. This may trigger a similar treatment for its superclasses. The approach adopted in the prototype is to copy all superclasses, rendering them (at least partially) replicated (the chosen replication category is LCRep).

If a heavy use will suggest an excessive class replication in a system, this policy can be opportune changed.

4. CONCLUSIONS

In the paper, we have considered an extension of Smalltalk in the sense of distribution. Several Smalltalk machines coordinate their services to allow remote operations, replicated objects and migration of objects. Apart from the possibility of furnishing inter-machine services, we have considered new object categories based on replication and on the treatment while passing from one machine to another. The paper introduces a wide range of consistent replicated objects starting from tightly consistent objects up to loosely consistent objects where copies are simply identified as part of the same replicated object. This taxonomy helps in parameter translation and object migration. Crucial problem in the migration is the 'openness' of passive objects to processes that execute within them. This aspect needs further
The adopted design guideline is compatibility with Smalltalk, keeping minimal the changes to the virtual machine and image. The considered modifications succeeded in the minimization of overhead of remote services w.r.t. the local ones. Any local operation is not impaired by the introduced mechanisms that request remote services. While transparency has been followed as a main guideline for all introduced mechanisms, the migration policy has been left as a system decision. Any DE0 system can provide one of two scenarios. One where objects can migrate according to their will and another where context managers can decide the object policy of allocation on the basis of performance parameters [25]. Extensions that are within reach, such as asynchronous message passing, but of patent incompatibility, have been left aside.

REFERENCES