

DESIGN OF A DISTRIBUTED OBJECT MANAGER FOR THE SMALLTALK-80 SYSTEM

D. Decouchant

Laboratoire de Génie Informatique, IMAG, Grenoble
Université de Grenoble
BP 68, 38402 Saint-Martin d'Hères, France

-0-0-0-0-

ABSTRACT

This paper describes the design of a distributed object manager which allows several Smalltalk-80 systems to share objects over a local-area network. This object manager is based on the following principles : location transparency and uniform object naming, unique object representation and use of symbolic links for remote access, possibility of object migration and distributed garbage collection. A version of the object manager has been implemented and is currently being integrated on a two nodes configuration.

1. INTRODUCTION

The object model has received considerable attention in the recent years as a means for structuring distributed systems. The design of an experimental investigation of an object-based distributed system can be conceived along two main lines :

- Design a new language (or even a whole operating system) based on an object model, and implement such a language (or at least the kernel of a distributed operating system).
- Extend an existing object-based language with the facilities which would allow to use it in a distributed environment.

The first approach has been taken by several projects (e.g. Argus [Liskov82,Liskov84] Eden [Almes85]). These projects are large-scale, resource-consuming efforts. In an environment with limited resources, we decided to take the second approach, starting with an existing language. The choice of Smalltalk-80¹ was motivated by two considerations :

- Smalltalk-80 may be regarded as a paradigmatic language for the definition of an object model.
- The language and its implementation are well-documented, and the source code of an implementation of a Smalltalk system (Berkeley Smalltalk) was available.

The objective of the project is to design a distributed object manager, which could eventually be used on a network of personal workstations. This manager would replace the original object manager of the Smalltalk-80 system, thus allowing the users of several such systems to share objects, to perform remote execution, or to store objects in a repository.

The general layout of the object manager is represented on Figure 1. In the original Smalltalk-80 system, object management is integrated in the byte-code interpreter, which runs on a (centralized) operating system. Our design separates the interpreter from the object manager, while preserving the interface between the interpreter and the virtual image. The object manager is now distributed on several nodes of a local-area network.

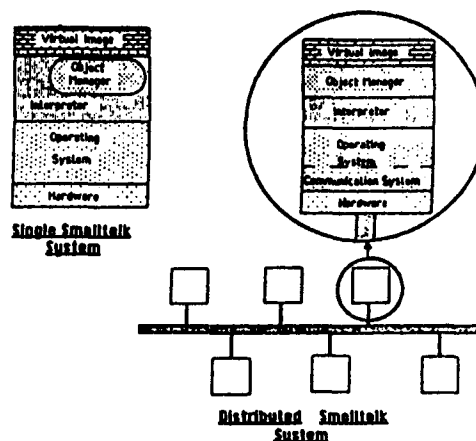


Figure 1. Organization of the distributed Smalltalk system.

This paper is organized as follows. Section 2 presents the general problems of object management in distributed systems : naming, sharing, migration, garbage collection. Section 3 describes the design principles of the distributed object manager. The main implementation features are described in Section 4. Some conclusions and directions for future work are presented in Section 5.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

2. PROBLEMS OF DISTRIBUTED OBJECT MANAGEMENT

The development of distributed systems has partly been motivated by the desire to extend the limited set of sharable resources of a particular computer system to a large, possibly unlimited, set of network resources. Resource sharing poses the problems of naming, protection and consistency. While these problems have been resolved to a satisfactory extent for centralized systems, they have not been adequately resolved in the area of distributed systems. In particular, the Smalltalk-80 system, initially designed for personal workstations, has no built-in facility for managing resources in a network of Smalltalk systems.

In a Smalltalk system, the elementary manageable resource is an object. As a result, resource sharing in a Smalltalk network boils down to the distributed management of objects. Distributed object management can be achieved by a collection of local cooperating object managers. Each local object manager runs on a particular workstation, is in charge of the objects that are local to the workstation and cooperates with remote object managers to achieve the distributed sharing of objects.

Problems that result from object sharing, such as naming, protection and consistency, are amplified by the facilities inherent to distribution. In particular, facilities such as remote access, dynamic migration of objects and dynamic connections and disconnections of workstations dramatically increase the complexity of a distributed management of objects. In the following paragraphs, attention is focused on the problems we have identified so far.

2.1. Naming

In general, naming provides users with a mechanism for designating objects without any concern with their physical location. Similarly, naming in a distributed environment allows objects to be designated without specifying their actual site of residence. Such object naming is a type of virtual naming (as opposed to physical naming).

Although location independence is generally desirable, virtual naming takes away from the user the possibility for direct control on object locations. For example, a user may want to force the migration of a particular object to a specific site for explicit transfer or for taking advantage of effective parallelism. In short, an appropriate naming mechanism should be location independent while providing the user with facilities for controlling object locations when needed.

2.2. Access

Access to objects is either local or remote. An access is local when the object is on the same site as the caller. It is remote when the object is located on a distinct site from the caller. Problems related to local accesses have been resolved satisfactorily. Remote access can be handled in two ways: either an effective remote access is performed and the object does not migrate, or a local access is performed and the object is forced to migrate round trip between the calling and the residence sites. Round trips can be avoided by migrating a copy of the object. Object duplication induces an overhead whose cost in terms of space and computation may exceed migration costs (in particular Smalltalk objects are small). The choice between actual remote accesses or local accesses with migration depends on communication load, objects size and application requests.

2.3. Protection

Protection is usually concerned with the detection of unauthorized accesses. It may also be involved in the control of excessive consumption of resources.

Unauthorized accesses may be performed by a program which, maliciously or not, has confiscated some "strategic" data belonging to another program: to do so, the malicious program first reads data and acquires the capabilities of the victim program. Then, it can access some secret information or randomly perform some kind of sabotage. In distributed environments, the problem is amplified by resource migration. In particular, a resource may migrate to a new site which then must be in charge of protecting this resource.

Protection in general purpose systems such as time sharing systems, is difficult to achieve. In these systems, entities are organized as large collections of information and tied down to a unique entity, the process. As a result, once the capabilities of a process have been confiscated, unintended accesses can easily propagate through a large amount of information. Conversely, in object-oriented systems such as Smalltalk, the fine-tunable granularity of data and programs in the form of independent objects, provides a natural framework for protection.

Control of excessive consumption of resources remains an important problem.

2.4. Memory allocation and Garbage Collection

In the previous paragraphs, we observed that object-oriented systems are well-suited to the management of resource sharing. However, these systems are intensive object consumers: objects are allocated and released with high frequency. Moreover, they are organized in graph structures that express reference relationships. As a result, a Smalltalk system can rapidly be congested by unused objects if these are not reclaimed by a garbage collector.

Garbage collection consists in detecting unused objects and then reclaiming the memory space allocated to these objects. It implies costly graph traversing that may be prohibitive in the absence of optimization. In addition, garbage collection is interleaved with foreground processings. This interleaving may result in unexpected long reply times for the operator. Ungar in [Ungar84a] has tackled the problem and proposed the "generation scavenging". Generation scavenging is based on a hierarchy of object generations: if an object survives from a garbage session, it is made older and is classified in the immediate upper generation. It is no longer subject to on-line reclaiming if it belongs to the root generation. This solution has been investigated for centralized systems but needs be adapted to deal with migrations.

So far, we have identified the problems posed by the management of objects in a local network of Smalltalk workstations. In the next section, we present the design principles that we adopted to solve these problems.

3. OBJECT MANAGER DESIGN PRINCIPLES

Our object manager is implemented as a collection of local cooperating object managers. Each local object manager runs on a particular workstation, and provides the programmer with a collection of primitives. These primitives allow to name and share objects without being aware of their actual physical location. In addition, the object manager is able to cope with dynamic features such as

migration, garbage collection and connections/disconnections of sites. This section describes how this is done.

3.1. Basic Design Decisions

Remote references and "proxy" objects

The driving design principle is physical location transparency. An object name is always a pointer to a local object. More precisely, if an object *O1* holds a name *N* to reference another object *O2*, *N* always identifies an object localized on the same site as *O1*. For remote references, we introduce a particular type of object, the "proxy", that locally represents a remote object. As shown in figure 2, *O2* is located on site *B* and a proxy object is created on the site *A* of *O1* to serve as a local representative of the remote *O2*. In addition, if *O3* at site *A* also references *O2*, it uses the same proxy object as *O1*. This solution has been adopted to ensure correctness and homogeneity of the Smalltalk-80 image. The migration algorithm detailed in the next section also verify this property.

A proxy is part of the private data of the object manager. It has two fields that describe a remote object: the residence site of the remote object and the virtual pointer to the object in the residence site. When the referenced object migrates, the contents of the referencing object is not modified. The proxy is updated accordingly by the object manager. In addition to physical location invisibility, the proxy preserves the local consistency of reference counts. A proxy is functionally equivalent to a Unix¹ link except that a proxy is not visible to the programmer. It is a private data structure which is handled by the object manager like any Smalltalk object for swapping, garbage collection and memory relocation.

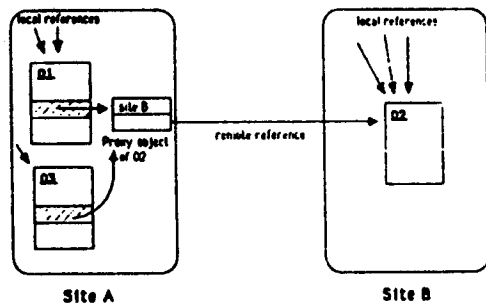


Figure 2. Proxy object usage for remote access.

Given the notion of proxy for physical location independence, we need now to be precise about how to solve the following problems:

- 1) Migration of an object in the distributed system.
- 2) Reference to a proxy object by another proxy object.
- 3) Distributed circular structures reclamation. This is an important task of a centralized object manager and this problem is increased by system distribution.
- 4) Consistency of shared object access and privileges of different systems on shared objects.
- 5) Dynamic system connections and disconnections protocols, and consistency of others systems connected to the network.

¹ Unix is a trademark of ATT-Bell Laboratories.

Object migration

Several cooperating Smalltalk systems may need to share objects to implement a distributed application. Sharing can be handled through remote or local access. Remote access is performed through the "proxy" mechanism. For efficiency purposes, access to a remote object may be transformed into a local access by copying or migrating the desired object.

In an Smalltalk environment, object duplication is not appropriate. Given object size and the high rate of object creation and deletion, the maintenance cost for managing of duplicated object would be prohibitive. The migration of an object is independent from changes in its ownership.

Object migration consists in moving a real object from a workstation to another. A real object is an entity explicitly manipulable by the user. When migrating a real object is replaced by a proxy object in the source workstation.

A real object may come back to its previous site, then migrate again. Therefore, when migrating an object it is necessary to check the possible existence of a proxy object for the migrated object. If a proxy already exists, the real object and its proxy are exchanged.

The real object moves to its new site and takes the proxy object's name. A proxy object replaces the real object on the old system and references the real object through the network.

In addition, when a migrating object is shared between more than two systems, all its proxy objects on these systems need be updated. This updating is performed according to an appropriate protocol.

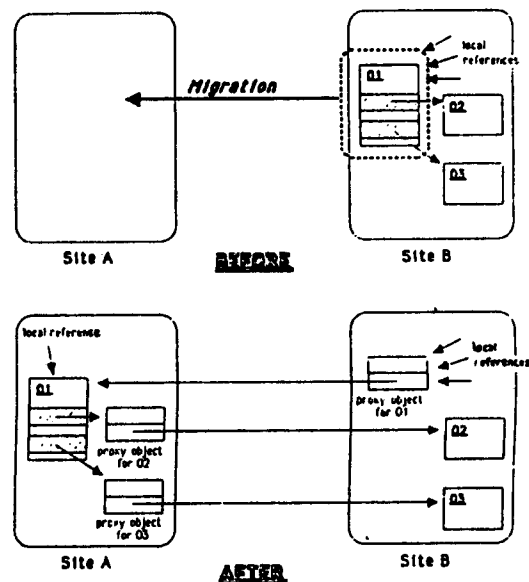


Figure 3. An example of object migration.

For this protocol, each local system maintains a sorted list of the local names of its proxy objects. By doing so, each local system is able to respond to the question: "does a proxy object locally exist for a given object?"

Storage management and garbage collection

Functional and object-oriented languages impose a heavy load on storage management. In Smalltalk, memory requests are very frequent and fragment the memory in a vast number of small pieces. Paging systems are ill-suited to this type of requests. Therefore, the object is the unit for storage management. The main storage is allocated according to a buddy-system scheme. Free memory blocks are split up and recombined according to the following rule:

$$L(i) = L(i-1) + L(i-k)$$

Where $L(i)$ represents the block size of level i , and k is a parameter (the "splitting order") which allows to choose between several algorithms :

$k=1$ Binary buddy system: $L(i) = L(i-1) + L(i-1)$

The block size series is:

1 1 2 4 8 16 32 64 128 ...

$k=2$ Fibonacci buddy system: $L(i) = L(i-1) + L(i-2)$

The block size series is:

1 1 2 3 5 8 13 21 34 55 79 ...

...

$k=6$ Buddy system of order 6: $L(i) = L(i-1) + L(i-6)$

The block size series is:

1 1 1 1 1 1 2 3 4 5 6 7 9 12 16 21 27 34 43 55 71 ...

...

Large values of the splitting order allow a finer control on the size of the blocks. Since most Smalltalk objects are small, a fairly high order ($k=9$ or 10) may be a good choice. This solution and its advantages have been presented in [Peters77, Shen74].

The method allows efficient allocation and liberation and reduces internal and external fragmentation.

Internal fragmentation results from allocating memory in predefined block sizes. More precisely, it results from allocating a larger block size for a request for which the size does not exist in the buddy-system series.

External fragmentation results from splitting the free memory into blocks which can be recombined only if they are "buddies".

Our implementation of storage management is based on the following decisions:

- a hashing-function over the pointer value of the objects, for the topography function
- LRU for the replacement algorithm.

We need now to consider the effect of dynamic linking, remote object access, and object migration on the storage management policy.

In a Smalltalk system an object exists if its reference count (i.e. the number of objects that are currently using it) is different from zero. This implies garbage collection for the reclamation of storage. In a distributed Smalltalk system, an object may be used by local or remote objects.

In order to maintain the reference mechanism for managing object existence in a distributed system, it is necessary to introduce a new reference count for each real object: the remote reference count. The value of this count is the number of proxy objects which reference the real object through the network. It is less or equal to the real number of referencing objects. The figure 4 illustrates this property.

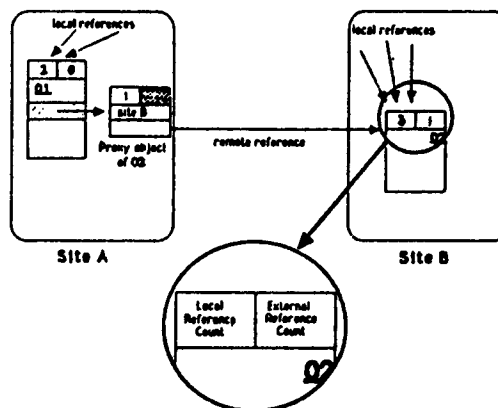


Figure 4. Possible states and reference counts of an object.

Our approach introduces two levels of garbage collection: local and distributed storage reclamation.

Local reclamation.

In section 2.4, we introduced a convenient solution for storage reclamation on a centralized Smalltalk system: Generation Scavenging [Ungar84b]. This method is based on the observation that most objects either die young or live forever. An object gets older by moving it to an older "object generation" if its storage is not reclaimed. This storage reclamation algorithm matches well object evolution. This algorithm has been implemented on the Berkeley Smalltalk (BS) version of Smalltalk-80 [Ungar84a]. We are using this implementation on a Sun-2 workstation. As we already mentioned, this solution is convenient for local garbage collection, but not for distributed garbage collection.

Remark:

In this object memory organization, main memory garbage collection is not crucial because object is the single management entity. However, garbage collection is required by secondary memory management.

Distributed reclamation.

A major problem with garbage collection is reclaiming distributed circular structures. As described in figure 5, distributed circular structures may result from proxy objects usage and objects migrations. An elementary solution consists in performing a distributed marking. Each system marks all of its accessible objects in its virtual image. Whenever a proxy object is found, a marking request

is transmitted to the site of residence of the real object. While remote marking may be done in the background, collecting is an expensive operation which holds up the entire network. This real object constitutes a new starting point of the marking process for its site.

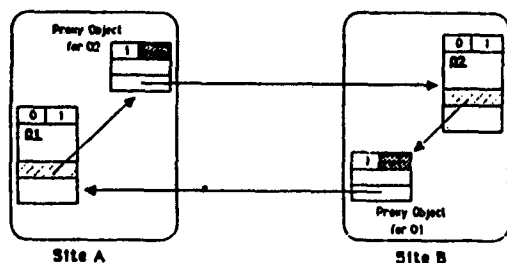


Figure 5. Example of distributed circular structures.

Currently, no better solution to this problem has been investigated. It constitutes an interesting and important topic for future research.

Dynamic connections and disconnections

One of the requirements for our distributed system is that individual workstations may be dynamically connected or disconnected. In order to ensure the global consistency of the system, specific operations must be performed at connection or disconnection time.

When a new site is connected, it broadcasts its readiness to communicate with the rest of the network. Thus, connection protocol is simple and easy to implement.

System disconnection is more difficult for two major reasons.

- First, the disconnecting system may currently support objects owned by other systems. These objects need to be sent back to their owners, or to any other system ready to receive them for later use.

Sending back objects is necessary to ensure a graceful disconnection; For traumatic disconnections such as software or hardware errors, no satisfactory solution is available.

- More problematic is the case where objects owned by the disconnecting system are currently present on other systems.

Access protection

Access protection ensures that any access to an object is made in conformity with some stated rules of usage. Three levels may be defined:

- Object protection provided by the Smalltalk access rules (i.e. the methods defined by the object's class), which is equivalent to abstract data type protection. This protection is ensured by the Smalltalk interpreter and will not be considered further.

- Object rights for remote access control.

- Protection of a site from unwanted remote access.

Object migration requires strict access control. To implement it, two additional fields are associated with each object. One of them

represents read and write capabilities on this object by foreign systems. The second one is the name of the owner system. For homogeneity reasons, a proxy object also has these two indicators.

Thus, an access is correct if

- either the accessing system is the owner of the object,
- or the access rights are compatible with the attempted access if the system is not the owner system.

To protect each system against unwanted intrusion, we decided not to use the standard remote procedure call but to perform remote object access through a unique server process on each workstation. This process controls the validity of the access and performs the access for the client system.

3.2. Interface specifications

The object memory level provides an interface to the objects that make up the Smalltalk-80 virtual image. The interpreter uses object pointers to access the objects. An object pointer is the system name of each object and it is invisible at the application level.

The interface provides the following classes of primitives :

- Internal and external reference counting
- Access to the class pointer and length of an object
- Access to elementary values (word, byte, integer)
- Object creation
- Object migration
- Enumeration of the instances of a class

A complete list of the interface specifications is given in the Appendix. Modifications of this object memory interface are still possible.

3.3. Organization of the object manager

Three processes cooperate to perform the object management functions: the network manager, the main memory manager and the secondary memory manager (see figure 6). In addition, several others processes may be present on a site: Smalltalk interpreter processes which access the objects to perform Smalltalk actions. These processes may be viewed as clients of the system.

The network manager (NM).

The network manager is the master-process of a Smalltalk site. It ensures consistency of the shared objects with the other network sites and controls the local processes.

The main memory manager (MMM).

The main memory manager is in charge of the object management in main memory (relocation, swapping, free space management, etc) and garbage collection. It resolves object faults by allocating free space for the missing object and sending an object load request to the secondary storage manager. After an object has been loaded, the main memory manager reactivates the Smalltalk interpreter.

ter process which has caused the object fault.

The secondary memory manager (SMM).

This process takes care of the object management in secondary storage. This storage is actually represented by two files, one of which contains the Smalltalk object table and the other one the object space.

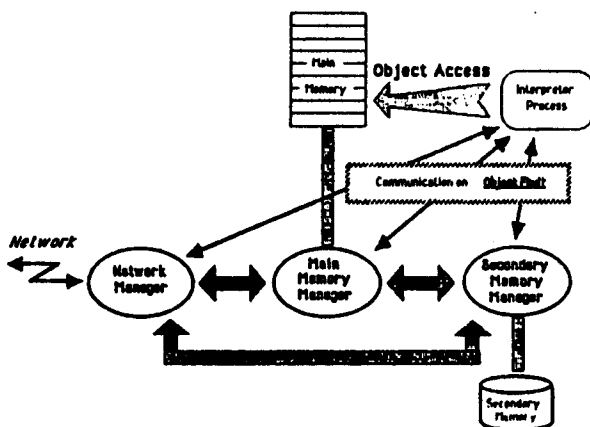


Figure 6. System processing model.

4. IMPLEMENTATION

In the above design, the objects located in main memory must be made accessible to all processes. Our implementation is made on top of the Unix 4.2 BSD, and this system does not allow memory sharing between processes. Consequently it is not possible to implement this design by allocating one Unix process for one Smalltalk process.

The solution is to manage several "lightweight processes" in one single Unix process. Thus we have implemented a Unix process multiplexor (MUX) which takes care of the internal scheduling among several sub-tasks (lightweight processes). These processes communicate by messages through mailboxes.

MUX provides some primitives similar to those of Unix system:

- process creation.
- process destruction.
- process interrupt.
- process resume

and some input/output primitives.

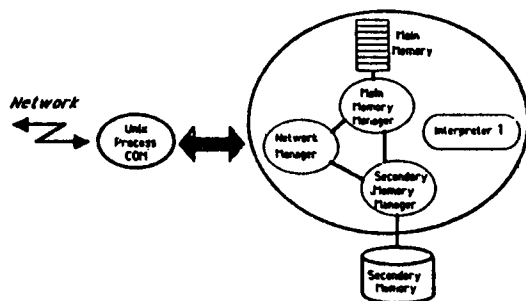


Figure 7. System implementation.

An additional Unix process (COM) is introduced to buffer the network communications of this implementation. This buffering is necessary because network communications processing may lock the receiver. This is a constraint imposed by Unix communications. These two Unix processes (COM and Smalltalk implementation) communicate by a socket connection. This connection is dynamic and allows several Smalltalk implementations to run on the same computer. Thus network communications may be simulated by pipe communications and workstation connections by socket connections.

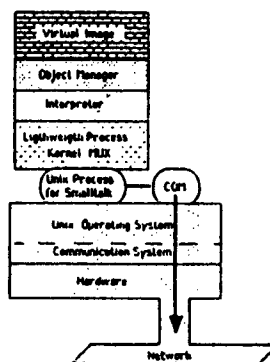


Figure 8. Layered architecture for a Smalltalk-80 workstation

The current status of the project is as follows. The system is being developed on a Sun 2/50 workstation, under Unix 4.2 BSD. The main part of the object manager has been implemented and tested. Object access and migration are running between two simulated sites implemented on the Sun. The next step is to connect the Sun to a Vax 11/780 also running Unix 4.2, through an Ethernet network. This will allow effective object migration between the two sites.

In parallel, we are modifying the BS Smalltalk-80 interpreter in order to include the object manager. The final step will be to design an interface enabling the user of the Smalltalk system to use object sharing capabilities; this interface will be in the form of Smalltalk classes.

5. CONCLUSIONS

We have designed for a distributed object manager that allows several Smalltalk systems to share objects on a local area network. This design is based on the following guidelines:

- Location transparency, based on a uniform naming scheme,
- Single copy objects, with remote access through symbolic links ("proxy" objects),
- Object migration, involving link updates
- Distributed garbage collection

The implementation of the system is under way. A complete version of the object manager is used on a two-node configuration, and shall be integrated in the Smalltalk-80 system. Preliminary experiments seem to show that the functionality of distributed object management can be achieved with tolerable cost.

Acknowledgments

I would like to thank Sacha Krakowiak for his support and guidance at the beginning of this project, and Joelle Coutaz for her unvaluable help with the English syntax and style. Christian Tricot designed and implemented the MUX process manager and Gerard Vandome provided helpful advice on Unix communications.

References

- [Almes85] G.T.Almes, A.P.Black, E.D.Lazowska, and J.D.Noë : *The Eden System: A Technical Review* ; IEEE Transactions on Software Engineering, no. SE-11,1, IEEE, january 1985. (pp.43-59).
- [Goldbe83] Goldberg A., Robson D. : *"Smalltalk-80: The language and its implementation"* ; Addison-Wesley Publishing Company, Palo Alto Research Center, 1983.
- [Liskov82] B.H.Liskov : *On Linguistic Support for Distributed Programs* ; IEEE Transactions on Software Engineering, no. SE-8,3, IEEE, may 1982. (pp.203-210).
- [Liskov84] B.H.Liskov : *Overview of the Argus Language and System* ; Programming Methodology Group Memo 40, MIT Laboratory for Computer Science, february 1984.
- [Peters77] Peterson J.L., Norman T.A. : *Buddy Systems* ; Communications of the ACM, vol. 20, no. 6, june 1977. (pp.421-431).
- [Shen74] Shen K.K., Peterson J.L. : *"A Weighted Buddy System for Dynamic Storage Allocation"* ; Communications of the ACM, vol. 17, no. 10, october 1974. (pp.558-562).
- [Ungar84a] David Ungar, Ricki Blau, Peter Foley, Dain Samples, David Patterson. : *Architecture of SOAR: Smalltalk on a RISC* ; IEEE, Ann Arbor, MI, June 1984. (pp.188-197).
- [Ungar84b] David Ungar. : *Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm* ; Symposium on Practical Software Development Environments, ACM Software Engineering, Pittsburgh, April 1984. (pp.157-167).

APPENDIX : Object manager interface specifications

Notations:

The interface to the object memory uses zero-relative integer indices to point to an object field.

An italic parameter is a pointer to a real object or to a "proxy" object. A non italic parameter is necessarily a pointer to a real object.

As previously stated, a proxy object is made of two fields: the name of the remote site and the object pointer to the real object on this site.

Internal reference counting.

procedure increaseReferencesTo (*objectPointer*)

Add one to the internal reference count of the object pointed to by "objectPointer".

procedure decreaseReferencesTo (*objectPointer*)

Subtract one from the internal reference count of the object pointed to by "objectPointer".

Similarly, it is necessary to define the procedures which modify the external reference count of an object.

External reference counting.

procedure increaseExternalReferencesTo (*objectPointer*)

Add one to the external reference count of the object pointed to by "objectPointer".

procedure decreaseExternalReferencesTo (*objectPointer*)

Subtract one from the external reference count of the object pointed to by "objectPointer".

Class Pointer access.

function fetchClassOf (*objectPointer*) ---> *classPointer*

Return the object pointer of the class-describing object pointed to by "objectPointer".

Length access.

function fetchWordLengthOf (*objectPointer*) ---> *size*

Return the number of fields (word or pointer) in the object pointed to by "objectPointer".

function fetchByteLengthOf (*objectPointer*) ---> *size*

Return the number of bytes fields in the object pointed to by "objectPointer".

Object pointer access.

function fetchPointer (*objectPointer* , *fieldIndex*) ---> *valuePointer*

Return the object pointer found in the field numbered "fieldIndex" of the object associated with "objectPointer".

procedure storePointer (*objectPointer* , *fieldIndex* , *valuePointer*)

Store the object pointer "valuePointer" in the field numbered "fieldIndex" of the object pointed to by "objectPointer".

Word access.

function fetchWord (*objectPointer* , *fieldIndex*) ---> *valueWord*

Return the 32-bit numerical value found in the field numbered "fieldIndex" of the object pointed to by "objectPointer".

procedure storeWord (*objectPointer* , *fieldIndex* , *ValueWord*)

Store the 32-bit numerical value "valueWord" in the field numbered "fieldIndex" of the object pointed to by "objectPointer".

Byte access.

function fetchByte (*objectPointer* , *byteIndex*) ---> *valueByte*

Return the 8-bit numerical value found in the byte numbered "byteIndex" of the object pointed to by "objectPointer".

procedure storeByte (*objectPointer* , *byteIndex* , *valueByte*)

Store the 8-bit numerical value "valueByte" in the byte numbered "byteIndex" of the object pointed to by "objectPointer".

Integer access.

Integers are encoded directly as object pointers with a 1 in the low-order bit position and value stored in the high-order 31 bits. These objects are instances of the class `SmallInteger`. The next four functions manipulate this particular type of objects.

function integerValueOf (*objectPointer*) ---> *value*

Return the value of the instance of `SmallInteger` pointed to by "objectPointer".

function integerObjectOf (*value*) ---> *objectPointer*

Return the object pointer for an instance of `SmallInteger` whose value is "value".

function isIntegerObject (*objectPointer*) ---> *boolean*

Return "true" if "objectPointer" is an instance of `SmallInteger`, false if not.

function isIntegerValue (*value*) ---> *boolean*

Return "true" if "value" can be represented as an instance of `SmallInteger`, false if not.

Object creation.

function instantiateClassWithPointers (*classPointer* , *instanceSize*)
---> *objectPointer*

Create a new instance of the class whose object pointer is "classPointer" with "instanceSize" fields that will contain pointers. Return the object pointer of the new object.

**function instantiateClassWithWords (classPointer , instanceSize)
 ---> objectPointer**

Create a new instance of the class whose object pointer is "classPointer". The "instanceSize" field contains a 32-bit numerical value. Return an object pointer on the new object.

function instantiateClassWithBytes (classPointer instanceByteSize) ---> objectPointer

Create a new instance of the class whose object pointer is "classPointer" with room for "instanceByteSize" 8-bit numerical values. Return an object pointer on the new object.

function createProxyObject (site , remoteObjectPointer) ---> localObjectPointer

Create a proxy object for a remote object whose the object pointer is "remoteObjectPointer" on site "site". Return (local) object pointer of the new "proxy" object.

Pointer swapping.

procedure swapPointersOf (firstPointer , secondPointer)

Make "firstPointer" refer to the object whose object pointer was "secondPointer" and make "secondPointer" refer to the object whose object pointer was "firstPointer".

Instance enumeration.

These two functions enumerate the instances of a class. If "classPointer" is a proxy object pointer, these functions enumerate local instances of a remote class sharing the object.

function initialInstanceOf (classPointer) ---> firstInstanceObjectPointer

Return the object pointer of the first instance of the class whose object pointer is "classPointer" in the defined ordering (e.g., the one with the greatest object pointer).

function instanceAfter (objectPointer) ---> nextInstanceObjectPointer

Return the object pointer of the next instance of the same class as the object whose object pointer is "objectPointer" in the defined ordering.

Object migration.

procedure migrateObject (objectPointer.toSite)

This procedure causes the object whose object pointer is "objectPointer" to migrate to the site "site". If this object is an proxy object, this procedure fails.