



The Design and Implementation of Distributed Smalltalk

John K. Bennett
Department of Computer Science FR-35
University of Washington
Seattle, WA 98195

Abstract

Distributed Smalltalk (DS) is an implementation of Smalltalk that allows objects on different machines to send and respond to messages. It also provides some capability for sharing objects among users. The distributed aspects of the system are largely user transparent and preserve the reactive quality of Smalltalk objects. Distributed Smalltalk is currently operational on a network of Sun workstations. The implementation includes an incremental distributed garbage collector and support for remote debugging, access control, and object mobility. This paper concentrates on the important design issues encountered and some of the more interesting implementation details. Performance measurements of the current implementation are included.

1 Introduction

Smalltalk [Ingalls 78, Goldberg 83] is a language and highly interactive programming environment originally developed for the Xerox family of personal workstations and now implemented on a variety of different hosts. The Smalltalk environment provides a single user with access to a single object address space. Multiple processes within Smalltalk are scheduled to run on a single processor. Although Smalltalk host machines can be interconnected with high bandwidth net-

works, only rudimentary support exists within Smalltalk for cooperation among users, and no support exists within Smalltalk for object sharing between users, communication between objects that reside on different machines, or cooperation between processes on several machines.

Distributed Smalltalk (DS) provides improved communication and interaction among geographically remote Smalltalk users, direct access to remote objects, the ability to construct distributed applications in the Smalltalk environment, and a degree of object sharing among users. Applications of Distributed Smalltalk include mail systems, remote computation servers, and remote file servers. The significant features of Distributed Smalltalk include:

- user-transparent remote invocation
- automatic creation of return references for remote message arguments
- support for remote process debugging
- an incremental distributed garbage collector that does not impact or restrict the method used for local reclamation
- support for the control of remote access (node autonomy)
- a naming mechanism that does not rely on the internal structure of the object memory of the host virtual machine
- support for object mobility (move and copy primitives)
- reactive remote objects

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.

© 1987 ACM 0-89791-247-0/87/0010-0318 \$1.50

- support for message interaction among heterogeneous (even non-Smalltalk) hosts

Distributed Smalltalk is not a direct extension of Smalltalk from one user on one machine to many users on many machines. Several characteristics of Smalltalk interact to make this approach undesirable. For reasons that we will describe in detail later, Distributed Smalltalk retains a logically distinct address space for each user rather than attempting to unify the users' object memories. Thus, in characterizing Distributed Smalltalk, the preceding list of "things it does" should be augmented by the following list of "things it does not do":

- Only modest support is provided for object mobility. When an object moves, its class must be present at the destination. Distributed Smalltalk provides some assurance, *but does not guarantee*, that the required class, if present, is compatible with the class from which the designated object was instantiated. Providing such a guarantee is difficult. Distributed Smalltalk attempts to compromise between functionality, efficiency, ease of implementation, and semantic clarity.
- Limited support is provided for object sharing. Although remote objects are reactive, two objects on different hosts may not have the same class (since classes and instances must be co-resident). Distributed Smalltalk does not provide an object server for persistent objects. There is no mechanism analogous to the Eden *checkpoint* [Almes 85] that "snapshots" the active state of an object into a passive, long-lived representation. Such a mechanism is not found in current Smalltalk implementations, but would be useful in Distributed Smalltalk.
- Smalltalk is a powerful programming environment that provides users with enough capability to shoot themselves in the foot. Distributed Smalltalk gives careless users some capacity to shoot others.
- Currently, object names do not possess either system-wide or for-all-time uniqueness. This decision was made for efficiency reasons (and to simplify the implementation). In contrast to the previous three points, which represent design decisions, lack of name uniqueness is a characteristic only of the current implementation of Distributed Smalltalk. Name uniqueness could be provided by implementing object names used for

remote naming as a machine ID / time-stamp pair.

Distributed Smalltalk is implemented on top of a prototype version of the **PS** Smalltalk interpreter [Deutsch 84]. The implementation is written primarily in Smalltalk. The system currently runs on a network of Sun-2 workstations linked by a 10 Megabit/second Ethernet and exhibits satisfactory performance. Distributed Smalltalk is built around a basic mechanism for remote object communication that transparently forwards messages to, and routes replies from, remote objects. Special objects in the address space of both the sender and the receiver handle the details of this interaction.

This paper concentrates on the important design issues encountered and on some of the more interesting implementation details. Results of performance measurements of the current implementation are also presented.

2 Design Issues

Distributing a single-user programming environment like Smalltalk poses two kinds of problems: those associated with increasing the number of processors and address spaces, and those associated with increasing the number of users. In distributing Smalltalk, the problems associated primarily with distribution include naming, concurrency, error reporting, and garbage collection. The primary multi-user problems are node autonomy and the interaction of inheritance and reactivity.

2.1 Inheritance and Reactiveness

Inheritance is a fundamental property of Smalltalk that allows objects to acquire behavior from other objects. Objects that describe behavior are called *classes*; objects that acquire behavior and that have state are called *instances*. All objects are an instance of some class. The classes form a *hierarchy*. Each subclass in the hierarchy may add to or modify the behavior of the object in question and may also add additional state. One of the advantages of inheritance is that it supports what is known as *differential programming*: "I want an object like that one, but with these changes". A major disadvantage of inheritance is the potentially awkward separation of object behavior and object state. This separation makes questions like "Has object *foo* been modified?" difficult to answer. This is because an instance acquires its behavior

from its class and from the entire superclass hierarchy of that class. Changes to instance behavior that occur within superclasses are difficult to detect efficiently. Detecting changes in object behavior is necessary to support certain mechanisms for object mobility.

A system is *reactive* to the degree that objects in the system can be easily presented for inspection or modification. Smalltalk is highly reactive in that all objects in the system can be so presented. This fact can have major impact on the performance of Smalltalk systems since things such as stack frames are full-fledged objects that can be displayed and modified during execution¹.

In designing Distributed Smalltalk, we found the interaction of inheritance and reactivity to be a major source of difficulty. Neither concept scaled well from a single-user, single-address-space environment to a multiple user environment. We observed that the manner in which these mechanisms are supported affects the character of the user interface, object mobility, and system performance and reliability. Seven design alternatives were evaluated:

1. **Disallow remote classes (i.e., require that classes and instances be co-resident.)** This approach impacts object mobility. Instances can only move to hosts with compatible classes. Ensuring class compatibility is hard.
2. **Make classes immutable and allow only the creation of new subclasses.** This choice eliminates the class compatibility issue but may seriously degrade the reactivity of the system. The impact of this alternative on the "typical" Smalltalk programmer is unknown.
3. **Maintain a "master" copy of the class hierarchy and cache copies of classes on machines where they are used.** There are several problems with this approach: maintaining cached copies of classes in a consistent state when the master copy is updated, deciding when it is appropriate to update the master copy, and the impact on system reliability and performance of locating the master copies on a single machine.
4. **Provide multiple copies of system classes on every machine and treat these copies as a replicated distributed database.** This

¹ For efficiency reasons, some Smalltalk implementations have two representations for stack frames, a hardware-compatible format used during normal execution, and a Smalltalk object format used when required. Conversion between these two formats is transparent to the programmer.

approach inhibits reactivity. Class modification by one user usually should not be immediately reflected by changes in class behavior for all users. This approach also exhibits poor performance.

5. **Require that an object's class pointer always point back to the class from which the object was created.** One problem with this idea is that classes, like instances, are objects that may move to other machines, thus requiring a pointer forwarding mechanism or some technique for informing all instances when a class moves. This is likely to be a costly operation.
6. **Provide a means for class pointers of objects that are being moved to point back to the creating class definition when appropriate, but allow class pointers to "splice in" to the class hierarchy on the destination machine when possible.** This method implies that some classes are replicated and some are not. One of the problems with this approach is determining when it is appropriate to splice a class pointer into the destination machine class hierarchy.
7. **Require that users mutually agree that classes are compatible before allowing an instance of that class to be moved and spliced into the destination class hierarchy.** The principal problem with this idea is developing the mechanism by which users can agree on class compatibility. This approach also exhibits the undesirable property that system behavior is dependent on user sophistication.

These alternatives were evaluated on the basis of semantic clarity, ease of implementation, reactivity, mobility, performance, and reliability. Based on these criteria, options (3) and (4) were clearly poor choices. We rejected option (7) because it required an unreasonable level of user cooperation. When examining the remaining choices, we considered two major design alternatives: the desirability of allowing a class and its instances to reside on different hosts, and the need to support class replication. We found these two characteristics interrelated.

Since the behavior of a Smalltalk object is described by the object's class hierarchy, this hierarchy must, in principle, be accessed for each message received by the object, in order to locate the *method* containing the

code that is executed upon receipt of that message². In a distributed system allowing remote classes, the process of message lookup may have unacceptable overhead if a remote class hierarchy must be frequently accessed or if accessing this hierarchy becomes a system bottleneck. Thus, from a purely performance-oriented perspective, classes and instances may often need to be located on the same host even if remote classes are allowed. This is possible in a distributed Smalltalk system only if classes can be replicated in some manner (since we may want objects on different hosts to exhibit identical behavior). If remote classes are allowed, the reactive property of Smalltalk objects also compels class replication. If classes were not replicated, the modification of a system class by one user could cause immediate and unexpected system behavior for another user. A third reason for class replication in a system allowing remote classes is reliability. If all classes are located on a single host then the failure of that host or the network renders every host unusable. Thus support for both co-location and class replication is required.

Class replication in a reactive system can also create problems. If users on different machines are allowed to create incompatible versions of a system class, considerable confusion can ensue if instances of these classes move between machines.

Our early research favored option (6). This alternative would have provided a single logical address space where all objects were functionally part of the same class hierarchy and where all objects were equally reactive to all users. The problem of multiple users sharing a single address space was not novel, however the interaction of inheritance and reactivity in such a system resulted in an object model that possessed several internal inconsistencies. In particular, no satisfactory mechanism was found that correctly determined when to "splice-in" to the remote class hierarchy and when to "point back". The information about the class hierarchy required to make this decision was in each case obscured by the inheritance and subclassing mechanisms.

Option (5) was rejected for performance reasons. Although option (3) may in fact be a satisfactory alternative, it was rejected because the potential impact of immutable classes on the Smalltalk user was not clearly understood.

The current implementation requires that classes

²In practice a data structure called a "method cache" is usually employed to cache the class-message-method association. PS employs a more exotic scheme involving the use of an inline cache and self-modifying code [Deutsch 84].

and instances be co-resident (option 1) and accepts the resultant weak support for object mobility. The implementation does not attempt to unify the users' object memories into a single address space. Rather, each user is given a logically distinct address space. The implication of this decision is that the scope of inheritance is restricted to each user's address space. This restriction does not appear to unduly limit the development of distributed applications.

2.2 Naming

Many Smalltalk implementations use a level of indirection known as the "object table" to translate object pointers ("oops" in Smalltalk parlance) into physical memory addresses. (In systems without an object table, oops are real address pointers.) In addition to this local mapping, Distributed Smalltalk must provide a mapping from local oops to remote objects. It is also necessary to keep track of local objects that are remotely referenced so they won't be mistakenly reclaimed as garbage (since the external reference may be the only reference). This information is required in order to forward references if an object moves. Distributed Smalltalk requires a level of indirection to map an object identifier known by the system (that can be passed between hosts) to a local memory address (that may change periodically as a result of compaction or *become* operations). In systems that do not use a local object table this level of indirection is essential. In Smalltalk systems that do utilize an object table, it is possible to bypass this indirection to improve performance since the host-oop pair is an adequate system identifier.

2.3 Garbage Collection

The Smalltalk object memory resides in a heap. Space is allocated from the heap when objects are created. Deallocation is performed only by the system and only when objects are no longer referenced. This condition is detected either by continuously keeping track of the number of references for each object (reference counting) or by periodically determining which objects are still referenced and deleting those objects that are no longer referenced (garbage collection).

In a distributed environment, traditional reference counting is likely to exhibit excessive overhead due to the presence of references spanning multiple machines. Maintaining reference counts in this situation requires that machines inform one another when updating is required. An unpredictable number of network transmis-

sions may be generated whenever the reference count of an object drops to zero since the object may be part of a chain of objects with only one remaining reference.

Traditional mark-and-sweep is also inappropriate for a distributed environment. It is unreasonable to expect processing to come to a halt on every machine in the system until the marking phase completes.

A garbage collector with the following characteristics is required:

- No non-garbage object will ever be accidentally reclaimed.
- Any garbage created during one collection cycle will be reclaimed no later than at the conclusion of the next collection cycle.
- The garbage collection process must be able to proceed in parallel with other system activity, including the creation of new objects and new garbage.
- The collector must be robust enough to handle such problems as cyclic structures of garbage that span multiple machines and system or network crashes that occur during a collection cycle.
- The collector must not overwhelm the system while executing.
- It should be possible to utilize the local system garbage collector (or reference counter) for objects that are only locally referenced, and to restrict the purview of the distributed garbage collector to objects that are remotely referenced. The local collector must be prohibited from reclaiming an object that is only remotely referenced. This approach has the added benefit of eliminating the need for the distributed garbage collector to worry about compaction.

Since the marking or counting phase progress should be incremental, the reclamation phase of the collector should also proceed incrementally. It should also be possible to reclaim local garbage (objects that have never been referenced remotely) independently of the reclamation of system garbage.

2.4 Concurrency

Communication and cooperation between objects supported on remote processors requires an underlying communication mechanism that provides synchronization between the processes on each processor.

Since the semantics of Smalltalk message passing imply a blocking send, Distributed Smalltalk follows a variation of the traditional remote operation model [Spector 82]: the sending process blocks, the receiver process performs the operation, a value is returned, and the sending process then resumes. The variations have to do with the point in time at which the receiver process is created and whether or not values are returned. In Distributed Smalltalk the receiver process is created upon message receipt and a value is always returned. The default return value is a pointer to the receiver.

2.5 Error Reporting

Smalltalk provides a highly interactive error reporting and analysis capability. When an error occurs, the process that was executing at the time is suspended and the user is notified through a *notifier* window. A notifier contains a simple description of the error and a view of the state of the suspended process. The user may perform in-depth diagnosis of the error through a *debugger* window, or may elect to discard the notifier and the suspended process.

Any distributed implementation of Smalltalk should provide essentially the same functionality. Interaction with remote objects during error handling should be as similar as possible to interaction with local objects. Remote error reporting is difficult since the process that gets suspended when an error occurs is executing on a different processor than the process that gets notified that the error has occurred. Distributed Smalltalk provides the means to debug local and remote processes with the same mechanism.

2.6 Node Autonomy

The final issue influencing the design of Distributed Smalltalk was the question of machine ownership and protection. Since Smalltalk was designed as a single-user system, no protection mechanisms were included. In a distributed environment, the "owner" of a workstation may wish to exercise autonomy over that workstation's resources by restricting access to objects and machine resources that he or she considers private. An example of such a resource might be the display bitmap. This restriction may take the form of complete denial of access or the substitution of a surrogate object for those local objects the user wishes to protect. Distributed Smalltalk provides the means for a user to exercise both kinds of autonomy.

3 Implementation

In this section we will describe how the design decisions discussed in Section 2 are implemented in Distributed Smalltalk.

3.1 An Overview of DS

The sole paradigm for object interaction in Smalltalk is message passing. A sender sends a message to a receiver to cause the receiver to perform some action. Distributed Smalltalk retains this paradigm by inserting a message forwarding and reply service between the sender and receiver. This service is transparent to both the sender and the receiver. From the perspective of both the sender and receiver, local object interaction is identical to remote interaction. Two major components comprise this forwarding service: proxyObjects and the RemoteObjectTable. ProxyObjects represent remote receivers to the local sender, while the RemoteObjectTable represents remote senders to the local receiver.

A proxyObject represents a remote object to all objects in the local address space. There is one proxyObject per host per remote object referenced by that host. In this way, proxyObjects cause a remote object's message interface to appear to local objects as if the remote object were locally resident. ProxyObjects redefine the *doesNotUnderstand:* message of Object. This is the primary message defined for proxyObjects; messages sent to proxyObjects are intended to fail. Smalltalk responds to this failure by sending the message *doesNotUnderstand:* to the receiver with the message that was not understood as an argument. ProxyObjects' response to the *doesNotUnderstand:* message is to forward the original message to the RemoteObjectTable on the appropriate machine. The location of the remote object is part of the internal state of the proxyObject. The details of proxyObject/RemoteObjectTable communication are part of their private interface.

Since ProxyObject is a subclass of Object, messages defined by Object must be handled differently. Methods that are looked up normally can be redefined by proxyObjects to forward the message as with *doesNotUnderstand:*. However, some class-selector pairs are designated as "no-lookup" by the Smalltalk virtual machine. An example of a no-lookup class-selector pair is the selector `==` defined for class Object. When the Smalltalk interpreter encounters a no-lookup class-selector pair, the interpreter jumps directly to the routine which implements the corresponding primitive in-

stead of performing the normal method lookup. These no-lookup special selectors cannot be redefined within the Smalltalk environment.

Handling no-lookup methods intercepted by the virtual machine requires that we replace references to these methods in the source code with references to methods that invoke the forwarding mechanism. For example, `==` becomes `EqualEqual`. `EqualEqual` and `==` are functionally equivalent, they just differ in how they are accessed. `EqualEqual` is defined in class ProxyObject to forward the `==` selector to the target remote object. This source code scanning and "purification" is provided as a user utility and is performed when source code is migrated to another host by Distributed Smalltalk.

The RemoteObjectTable is responsible for receiving and replying to messages forwarded by proxyObjects. There is one RemoteObjectTable per host. It is the sole instance of class RemoteObjectTable. The RemoteObjectTable can be thought of as a set of extensions to the object tables (if present) of all remote machines. The RemoteObjectTable keeps track of all local objects that are remotely referenced. When the RemoteObjectTable receives a message from some proxyObject, it schedules a process that will contain the execution context of the actual message receiver by sending the message *perform* to the receiver with the forwarded selector and arguments (if any) as arguments to the *perform* message. The value returned by the *perform* message is returned to the remote sender in the reply message constructed by the RemoteObjectTable. The RemoteObjectTable also manages the local requirements of system-wide garbage collection on an incremental basis. The RemoteObjectTable ensures that objects that are only referenced remotely are not mistakenly garbage collected by retaining a reference to all such objects.

The remainder of this section examines some of the details of remote object interaction, object mobility, garbage collection, remote debugging, and node autonomy.

3.2 How Remote Objects Interact

ProxyObjects forward messages by invoking the *remoteSend* primitive. There are seven arguments to this invocation:

messageType An integer that distinguishes user (message forwarding) from kernel (system support) messages. The messageType also specifies the format of the argument string.

remoteHost A string designating the target host.

targetObject An integer encoding of the target object oop.

argumentString A string that encodes all of the forwarded message arguments.

selector A symbol specifying the message selector.

replyArray An array oop provided by the sender into which the receiving RemoteObjectTable will deposit the reply.

replySemaphore A semaphore oop that the sending process waits on after performing the send. This semaphore will be signaled upon receipt of the reply or after the message timeout period has elapsed.

ReplyArrays and replySemaphores are necessary because there may be several threads of control performing remote sends on a single host. DS keeps track of this situation by having each thread provide its own replyArray and replySemaphore. The replyArray has three fields: a messageType, a valueType, and a returnValue. The returnValue is an oop for either a string encoding of the literal value or for a proxyObject for an object on the receiver's host. DS creates ProxyObjects for both arguments and return values automatically.

Argument strings can have two formats. The first format is appropriate only for certain types of literals and is the result of sending the Smalltalk message *storeString* to the argument array. The second is a more efficient encoding called an *etherString*. *EtherStrings* can be used to encode arbitrary argument arrays. *EtherStrings* encode immutable literals as a representation of their value. Arrays encode as an *etherString* consisting of a sequence of *etherStrings*. All other objects encode as their host-ooop pair. This pair is used on the receiving host to create a proxyObject that points back to the argument object.

The RemoteObjectTable has associated with it three processes: the messageProcess, the userProcess, and the kernelProcess. The messageProcess waits for any Distributed Smalltalk message to the host. Upon receipt of such a message, the messageProcess constructs a messageArray containing the *remoteSend* parameters. The messageProcess then deposits the messageArray into the synchronized queue associated with either the userProcess or kernelProcess, as appropriate. The userProcess and kernelProcess block until a messageArray is deposited into their queue. When a

messageArray is received, the process performs the indicated action, and then blocks waiting for the next messageArray.

3.3 Object Mobility

Although Distributed Smalltalk requires classes and instances to be co-resident, certain classes are replicated at each host. Most system classes are replicated in this manner. Prior to moving an instance, DS performs a basic check to ensure class compatibility. This check consists of determining whether the remote class and local class have the same class name, superclass name, internal format, kind of subclass, instance variable names, class variable names, and method dictionaries (the remote method dictionary must subsume the local dictionary). These tests are necessary but not sufficient to ensure class compatibility. If any test fails the any remaining tests are short-circuited.

Certain classes, such as class Class, are assumed to be compatible and are not checked. Objects that are immutable literals (like SmallInteger) are not moved at all; they are reinstantiated on the target host. For those classes that are checked, there are three cases to consider:

1. The required class is already present and is compatible.
2. The required class is present but is determined to be incompatible.
3. The required class is not present.

In case (1), DS proceeds normally. In case (2), the move fails and the user is notified of the error. In case (3), the user is asked whether the desired object's class should be moved. If the response is affirmative, the object's superclass is checked for compatibility. This procedure continues up the class hierarchy until class Object is reached, however, class Object may not be moved.

3.4 Distributed Garbage Collection

In Distributed Smalltalk, garbage collection is carried out in parallel with normal system activity. No changes are made to the existing local garbage collecting schemes. DS is quite general in this regard; one host may employ reference counting and another generation scavenging. System wide collection is performed by *prevention*: objects that are not garbage are prevented from being collected by the local collector by having a reference to these objects stored in

the sending process

```
Set defaultColor to new
Set extRefCount to zero
For each proxyObject FOO do:
[Send IncExtRefCount to FOO's host]
Forget all objects whose extRefCount is zero
Set defaultColor to normal
```

the receiving process

```
(upon receipt of the IncExtRefCount message)
Increment extRefCount by 1
```

Figure 1: Fast Garbage Collection Algorithm

the RemoteObjectTable. Objects are reclaimed by the system garbage collector by deleting this reference and allowing the local collector to reclaim them.

Distributed Smalltalk actually employs two system garbage collectors. The primary difference between the two collectors is that one is far more efficient than the other but does not reclaim cycles of garbage that span multiple hosts. The other detects and reclaims such cycles but is comparatively slow. Both collectors are user utilities. Either may be initiated by any user from any host. Once started, attempts by other users to initiate garbage collection are ignored.

The fast algorithm is shown in Figure 1. The basic algorithm is mark-and-sweep, with all proxyObjects considered to be in the root set. All communication between hosts during garbage collection uses kernel messages. By changing the default color to *new*, the collector will thereafter avoid newly created objects that may briefly exhibit inconsistent reference information. When the marking phase concludes, the external reference count will be valid (except for the possible existence of cycles). Any object with an external reference count of zero may be removed from the RemoteObjectTable. (This does not necessarily reclaim the object, since it may be referenced locally.)

The primary difference between the cycle algorithm and the fast algorithm is that the cycle algorithm must, in addition to constructing the external (remote) reference counts, also construct the internal (local) reference counts in order to determine which objects are potentially part of a distributed cycle of garbage. Fortunately, the only objects that must be examined are those objects in the RemoteObjectTable that are only referenced by the RemoteObjectTable (no other local reference) and which also contain pointers to proxyOb-

jects. The cycle algorithm always begins by employing a variant of the fast algorithm. This potentially reduces the workload for the cycle collector by eliminating the "easy" garbage first.

External reference counts are computed by counting the total number of remote proxyObjects that refer to an object in the RemoteObjectTable. Internal reference counts are obtained directly from the virtual machine if the host uses reference counting, and by some implementation-dependent means if the host does not employ reference counting. In the worst case, all objects in the local object space must be enumerated in order to search for references to proxyObjects. If access to the virtual machine is available, this penalty can be avoided. For example, in a copying garbage collector, all proxyObjects could be placed in a separate space so that existing inter-space reference information could be used to compute internal reference counts for the required objects. Complete reference count information is actually not required. We only need to know if the external reference count greater than zero and if the internal reference count is greater than one.

Once the internal and external reference counts have been computed, any object whose external reference count is zero may be removed from the RemoteObjectTable. If the external reference count is greater than zero, the object is potentially part of a distributed cycle and must be included in the final phase of the cycle algorithm. In this phase, objects whose internal reference count is greater than one are placed in the root set and another mark-and-sweep cycle is performed. Each root set object in the RemoteObjectTable is examined for pointers to proxyObjects. Kernel messages are sent to all remote objects so identified that cause those objects to be marked and searched for references to proxyObjects. This process continues until all objects accessible from the root set have been marked as valid. As with the fast algorithm, objects created during the marking phase are not examined or disturbed by the collector. At the conclusion of this phase, any unmarked object is removed from the RemoteObjectTable (and is, in fact, garbage).

3.5 Remote Debugging

When an error occurs, Smalltalk systems suspend the active process and report the error to the user. In the local case the processes associated with the sender and the receiver of the message that caused the error are on the same host (they are often the same process). In the remote case the process that detects the error is the one associated with the message receiver but

the process that effectively caused the error (the one associated with the sender) is on another host. It is the host on which the sender process executes whose user should be notified of the error. This mechanism is facilitated as follows: When an error occurs the system attempts to suspend the receiver process. If this process is not associated with some remote node, the receiver process (perhaps identical to the sender process in this case) is suspended and debugging proceeds at the local user's discretion. If the receiver process is associated with some remote host, the sender process on the remote host is notified of the error and the local receiver process is suspended. No user interaction occurs on the receiver process' host. When the sender process is notified of the error, it is suspended, the user is notified, and debugging proceeds normally. If the user elects to proceed, both the sender and receiver processes are resumed. The debugger interface was modified to provide simultaneous access to both the sender and receiver processes. Thus a user can inspect and modify the call stack of both sender and receiver during debugging.

3.6 Node Autonomy

Node Autonomy is supported primarily by three messages to RemoteObjectTable:

- *EnableRemoteAccessTo:*
- *DisableRemoteAccessTo:*
- *Agent:Of:*

DisableRemoteAccessTo: sets a flag in the RemoteObjectTable that causes attempts at remote access of the argument object to fail. The RemoteObjectTable detects this condition and returns an *access denied* message to the remote sender. *EnableRemoteAccessTo:* restores the accessibility of the argument object. The default condition of objects in the RemoteObjectTable is to have access rights enabled. *Agent:Of:* also denies remote access to a local object but, in addition, the RemoteObjectTable is provided with an agent object. This agent represents to the remote user a message interface similar to that of the object being represented. This allows a local object that the local user wants protected to be represented by a different object. The interaction of the agent and client object is entirely up to the local user. Remote senders are usually unaware of the presence of an agent unless the agent fails to respond to some message defined by the original object.

An example of the use of agents might be the creation of an agent for the display bitmap that allows remote users to interact with a small portion of the actual display. Two users employing these agents would be able to exchange visual information while each maintains overall control of his or her respective "desktop".

3.7 Interaction with PS

An alternative to implementing Distributed Smalltalk on top of PS would have been to implement DS directly in a new or existing virtual machine. This alternative was rejected because achieving satisfactory performance of a Smalltalk virtual machine is itself a difficult implementation problem. In retrospect this proved to be a wise choice; the basic system was operational in much less time than it would have taken to develop a virtual machine from scratch. There were several implications and benefits of the decision to use PS:

- The PS virtual machine is complex and its components are tightly interlocked. Modification of the PS virtual machine was ruled unwise early in the development cycle.
- The excellent performance of PS allowed the development and testing of most of the system code within Smalltalk.
- The ability to add user primitives in PS proved to be very valuable.
- PS provides access to the underlying Unix system calls. Although this interface is rather inefficient, having access to these calls reduced the number of new primitives that had to be added to support Distributed Smalltalk.
- A surprising feature of the PS kernel is its use of polling for I/O event detection. The reason appears historical: early Unix implementations on Suns incorrectly handled asynchronous signals, resulting in incorrect restoration of registers when a separate signal stack was used³. This problem was transparent to Unix but wreaked havoc within PS. To avoid this problem PS used polling. The use of polling places an upper bound on system IO performance associated with the dead time spent waiting to poll and determines the nature of the

³The problem has been corrected in release 3.2 of the Sun operating system.

message interface within the RemoteObjectTable. Since Unix provides very little buffer space within the kernel, synchronized queues had to be constructed within the RemoteObjectTable for buffering incoming messages. Manipulating messages through these queues consumes a significant portion of the total time required for a remote invocation.

4 Performance

Table 1 shows the measured performance of Distributed Smalltalk for various types of messages. The measured system consisted of two Sun-2 diskless workstations connected via the primary departmental 10 megabit/second Ethernet.

The table is divided into three sections. The first section shows the performance of the PS system for local messages with various argument types. All components of the send and reply time, and the time required to compute a standard result, are included in these figures. Since the implementation of Distributed Smalltalk does not affect the local performance of PS in any way, these figures also reflect the local performance of DS.

The second section of the table shows the performance of DS for remote messages with various argument types. Comparison with the local case shows that remote operations are slower than local operation by a factor of roughly 1000. No optimization of DS has been attempted thus far. Some will clearly be possible, but an order of magnitude of performance improvement is probably not possible. To our knowledge, most distributed systems, except ones supporting a highly optimized, microcoded remote procedure call, exhibit a performance difference between local and remote invocation of between two and three orders of magnitude. This difference between local and remote invocation performance must be taken into account when designing distributed applications. Distributed Smalltalk is no exception to this rule.

For comparison purposes, the third section of the table shows local performance for two other MC68000 implementations of Smalltalk: Tektronix Pegasus 4404 and Berkeley Smalltalk on a Sun-2. PS is a highly optimized implementation, and the remote performance of DS is only a factor of 300 worse than the local performance of Berkeley Smalltalk. The initial remote performance of DS is less than a factor of 100 worse than the local performance of many of the early implementations of Smalltalk.

Message Type	Number of Arguments	Send/Reply Time (milliseconds)
PS Local	0	0.13
PS Local	1 Lit Value	0.14
PS Local	1 Oop Arg	0.13
DS Remote	0	136.0
DS Remote	1 Lit Value	140.2
DS Remote	1 EtherString	138.1
Tek 4404 Local	1 Lit Value	0.24
BS Local	1 Lit Value	0.49

Table 1: Timings of Various Message Types

In examining the remote invocation time, we found the largest component to be the time spent process switching within Smalltalk (more than half). The next largest component was the polling dead-time previously discussed (about one fourth). Network communication in Distributed Smalltalk uses UDP datagrams. Less than one fourth of the remote invocation time is attributable to network communication. A "remote" invocation sent by a host to itself is about 30 milliseconds slower than the same invocation sent between two machines, indicating the presence of multiple processes competing for scarce CPU cycles.

5 Conclusions

5.1 Summary

The goal of this research was to develop a system that would permit (and encourage) the construction of distributed applications within the Smalltalk environment. That goal was achieved. Key ideas underlying the design and implementation include proxyObjects, the RemoteObjectTable, co-residency of objects and classes, and a remote message invocation mechanism that does not adversely impact local performance. An important conclusion of our work, evident in our design, is that the mechanisms of class inheritance and reactivity do not appear to be well suited to building distributed systems. In this regard, the *prototype* object model appears to be a much better choice than an object model based on subclassing. Lieberman discusses several problems with subclassing and inheritance when he makes the case for *delegation* as the more general alternative [Lieberman 86]. Our results support his conclusions, particularly in multi-user and multiprocessor systems.

A great deal of credit must go to the designers and implementors of Smalltalk for the success of this

project. The richness of the Smalltalk programming environment greatly facilitated the development of Distributed Smalltalk. The excellent performance of PS engineered by Peter Deutsch made the resulting DS system attractive to users.

5.2 Related Work

Distributed Smalltalk derives many of its specific ideas from the work of others. Some of these ideas are summarized below.

- ProxyObjects are similar in function to Decouchant's *proxy* [Decouchant 86]. ProxyObjects, however, are full-fledged Smalltalk objects, while Decouchant's proxies are part of the private data of the "Object Manager" portion of his resident virtual machine.
- A number of researchers have observed the problems associated with subclassing and inheritance [Borning 86, Lieberman 86, Synder 86, LaLonde 86, Sandberg 86]. These efforts helped identify coping with inheritance in Distributed Smalltalk as a major area of concern.
- The distributed garbage collector presented here is a distant descendent of the Kung-Song/-Chansler collector used in STAROS [Kung 77, Chansler 82, Gehringer 81]. Distributed Smalltalk, like STAROS, implements no policy regarding when garbage collection should take place. Using reference counts to detect distributed cycles was suggested by Vestal [Vestal 87].
- Vegdahl identified some of the difficulties associated with moving object structures between Smalltalk systems [Vegdahl 86]. Distributed Smalltalk uses techniques similar to his to handle complex structures.
- Distributed Smalltalk shares many historical and philosophical roots with Eden [Almes 85] and Emerald [Black 86]. Local experience with these systems provided the yeast for many of the ideas presented here.

5.3 Future Research Directions

Distributed Smalltalk provides fertile ground for further research in several areas:

- There is considerable room for improving the performance of Distributed Smalltalk. Beyond obvious optimizations, ideas in this area include

modifying the virtual machine to use an asynchronous I/O model, integrating local and system garbage collection, and building a native distributed virtual machine. The functionality of Distributed Smalltalk could possibly be improved by the inclusion of the asynchronous message sends of ConcurrentSmalltalk [Yokote 86] or an object server for long-lived and offline objects [Wiebe 86].

- Emerald [Jul 87] incorporates several concepts that might be applicable including the notion of "attachment" (attached objects move together), and "call-by-move" and "call-by-visit" (internode call-by-value and call-by-value-result, respectively). These concepts could be incorporated into Distributed Smalltalk without changing the user semantics by building them into the private proxyObject/RemoteObjectTable interface. The migration of active processes supported by Emerald would require significant support from within the Smalltalk virtual machine.
- Experimentation with other alternatives for addressing inheritance and reactiveness is an interesting area for further research. Examples include exploring the feasibility of immutable classes and developing robust mechanisms to allow objects to plug into remote class hierarchies. An alternative might be to distribute a prototype-based Smalltalk such as the *exemplar* system developed at Carleton University [LaLonde 86].
- Work is underway to develop single-user, multiprocessor Smalltalk systems [Ungar 86, Thomas 86]. These systems should be an interesting decoupling of the problems associated with distribution and multiple users.
- The design of Distributed Smalltalk avoids specifying policy where possible. When usage data has accumulated, it would be interesting to attempt to specify policies for garbage collection and object migration. Based on some metric (such as object interconnectivity and size), such an object migration policy might attempt to migrate objects to the "best" host in order to load balance or reduce network overhead [Eager 86, Korry 86].
- Finally, advantage has not yet been taken of the potential for heterogeneity provided by Distributed Smalltalk. We plan to use this support

to add Tektronix Pegasus workstations to the system in the near future. The construction of application-specific servers that provide a standard Smalltalk message interface is quite simple and could significantly expand the domain of Smalltalk application programs.

Acknowledgements

Alan Borning, Edward Lazowska, and Henry Levy provided extensive reviews of early versions of this paper. This work was supported in part by the National Science Foundation under Grant DCR-8420945. Computing equipment was provided in part by DECwest Engineering.

References

- [Almes 85] Guy T. Almes, Andrew P. Black, Edward D. Lazowska, and Jerre D. Noe. The Eden System: A Technical Review. *IEEE Transactions on Software Engineering*, SE-11(1):43-59, January 1985.
- [Black 86] Andrew Black, Norman Hutchinson, Eric Jul, and Henry Levy. Object Structure in the Emerald System. In *Proceedings of the First ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 78-86, Portland, Oregon, October 1986.
- [Borning 86] Alan H. Borning. Classes Versus Prototypes In Object-Oriented Languages. In *Proceedings of the ACM/IEEE Fall Joint Computer Conference*, November 1986.
- [Chansler 82] Robert J. Chansler Jr. *Coupling in Systems with Many Processors*. PhD thesis, Department of Computer Science, Carnegie Mellon University, August 1982.
- [Decouchant 86] Dominique Decouchant. Design of a Distributed Object Manager For The Smalltalk-80 System. In *Proceedings of the First ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 444-452, Portland, Oregon, October 1986.
- [Deutsch 84] L. Peter Deutsch and Allan M. Schiffman. Efficient Implementation of the Smalltalk-80 System. In *Proceedings of the Eleventh ACM Conference on Principles of Programming Languages*, pages 297-302, Salt Lake City, Utah, 1984.
- [Eager 86] Derek L. Eager, Edward D. Lazowska, and John Zahorjan. Adaptive Load Sharing in Homogeneous Distributed Systems. *IEEE Transactions on Software Engineering*, SE-12(5):662-675, May 1986.
- [Gehring 81] Edward F. Gehring and Robert J. Chansler Jr. *StarOS User and System Structure Manual*. Carnegie-Mellon University, 1981.
- [Goldberg 83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, Menlo Park, California, 1983.
- [Ingalls 78] Daniel H. H. Ingalls. The Smalltalk-76 Programming System: Design and Implementation. In *Proceedings of the Fifth ACM Symposium on Principles of Programming Languages*, pages 9-16, Tucson, Arizona, January 1978.
- [Jul 87] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. *Fine-Grained Mobility in the Emerald System*. Technical Report 87-02-03, Department of Computer Science, University of Washington, Seattle, Washington 98195, February 1987.
- [Korrry 86] Richard Korrry. *A Load Sharing Algorithm for a Workstation Environment*. Technical Report 86-10-03, Department of Computer Science, University of Washington, Seattle, Washington 98195, October 1986.
- [Kung 77] H. T. Kung and S. W. Song. An Efficient Parallel Garbage Collection System and Its Correctness Proof. In *Proceedings of the Eighth Annual Symposium on the Foundations of Computer Science*, pages 120-131, October 1977.
- [LaLonde 86] Wilf R. LaLonde, Dave A. Thomas, and John R. Pugh. An Exemplar Based Smalltalk. In *Proceedings of the First ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 332-330, Portland, Oregon, October 1986.

- [Lieberman 86] Henry Lieberman. Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems. In *Proceedings of the First ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 214-223, Portland, Oregon, October 1986.
- [Sandberg 86] David Sandberg. An Alternative to Subclassing. In *Proceedings of the First ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 424-428, Portland, Oregon, October 1986.
- [Spector 82] Alfred Z. Spector. Performing Remote Operations Efficiently on a Local Computer Network. *CACM*, 25(4):246-260, April 1982.
- [Synder 86] Alan Synder. Encapsulation and Inheritance in Object-Oriented Programming Languages. In *Proceedings of the First ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 38-45, Portland, Oregon, October 1986.
- [Thomas 86] David A. Thomas, Wilf R. LaLonde, and John R. Pugh. *Actra-A Multitasking/Multiprocessing Smalltalk*. Technical Report SCS-TR-92, School of Computer Science, Carleton University, Ottawa, Ontario, Canada K1S 5B6, May 1986.
- [Ungar 86] Dave Ungar. 1986. private communication.
- [Vegdahl 86] Steven R. Vegdahl. Moving Structures Between Smalltalk Images. In *Proceedings of the First ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 466-471, Portland, Oregon, October 1986.
- [Vestal 87] Stephen C. Vestal. *Garbage Collection: An Exercise in Distributed, Fault Tolerant Programming*. PhD thesis, Department of Computer Science, University of Washington, January 1987. Department of Computer Science Technical Report 87-01-03.
- [Wiebe 86] Douglas Wiebe. A Distributed Repository for Immutable Persistent Objects. In *Proceedings of the First ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 453-465, Portland, Oregon, October 1986.
- [Yokote 86] Yasuhiko Yokote and Mario Tokoro. The Design and Implementation of ConcurrentSmalltalk. In *Proceedings of the First ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 331-340, Portland, Oregon, October 1986.