

Transparent Forwarding: First Steps

Paul L. McCullough

System Concepts Laboratory
Xerox Palo Alto Research Center/Northwest

Abstract

Traditional object-oriented systems tend to be single-user. As we move from personal to interpersonal computing, we must look for ways to extend our programming paradigms. This research extends the Smalltalk-80 system to send messages transparently to objects residing on remote machines. We discuss two models for remote message sends, describe our current implementation, and suggest areas for future research.

1.0 Introduction

The Smalltalk-80 [1] message-passing paradigm is simple: an object sends a message to another object and receives a result object in return. In extant Smalltalk systems, senders, receivers, and messages themselves all reside in a single object space on a single machine. This work expands the Smalltalk paradigm to include the sending of messages to remote objects, that is, to objects residing in a different object space on another machine.

A trend in the use of workstations involves moving from private to shared information and moving from personal to interpersonal systems and interfaces. In terms of an object-oriented system, this can be supported by sending messages across machines in as transparent a manner as possible. Transparent message passing facilitates access to objects across machine boundaries without placing extra conceptual burdens on the programmer.

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 0-89791-247-0/87/0010-0331 \$1.50

In this paper, we describe the goals of this research, a simple model, problems with the simple model, a revised model, a description of our current implementation, and areas for future research.

2.0 Goals

The basic goals of this research are:

- a) any Smalltalk message may be sent to an object residing on a different machine,
- b) the source code should not reflect whether one is sending a message to a local or remote object (transparency so that programmers need not concern themselves about how the application is distributed),
- c) the resulting system may be used on any Smalltalk-80 workstation and, thus, there will be no changes to the virtual machine,
- d) the work will be done in the Smalltalk-80 programming language with little or no change to the compiler,
- e) there should be only minimal changes to the virtual image,
- f) the communication takes place over the Ethernet,
- g) that neither security nor concurrency need to be any better than in the current single-machine system.

3.0 Related Work

This research is related to Remote Procedure Call [2, 3]. RPC differs in that it is typesafe, that fairly extensive changes are needed to both the compiler and binder, and that the RPC system provides automatic generation of stub programs. Unlike RPC, this research attempts to make only minimal changes to the Smalltalk environment. This work is similar to that of Decouchant [4] but differs

because his work involved extensive modifications to the underlying virtual machine. Purdy, Maier, and Schuchardt [5] have addressed some of these issues in their system which connects a Smalltalk-80 system with an external object-oriented database system.

4.0 Transparent Forwarding -- A Simple Model

First, we present a simple model for Transparent Forwarding based on the use of proxy objects and the encapsulation of Smalltalk messages in network packets. This simple model raises several issues which will be addressed in a revised model presented later.

Any Smalltalk object residing on a machine may be represented by *proxy objects* on other Smalltalk machines. The object which a proxy represents is called a *remote object*. Any message to a proxy object is forwarded via the network to its remote object. Just as in a normal message send, the remote object will respond to the message by executing an associated method. The result object from the method execution is then returned to the proxy via the network, and the proxy returns the result to the original sender of the message.

Proxy objects are implemented by instances of class *Proxy*. Proxy's behavior is implemented using the "does not understand" feature of the Smalltalk-80 interpreter. Whenever a Smalltalk interpreter encounters a message whose selector cannot be found in the message dictionary of the receiver's class or any of its superclasses, the interpreter creates an instance of class *Message*. This *Message* instance holds the message selector and arguments of the message whose selector was not found. Next, the interpreter sends a **doesNotUnderstand:** message to the original receiver with the new *Message* as an argument. In the Smalltalk hierarchy, the class *Object* has a method for **doesNotUnderstand:** which invokes the debugger. Class *Proxy* overrides the **doesNotUnderstand:** message with a method which forwards the original message to the remote object. In order to capture all the messages to the proxy in this way, class *Proxy*'s message dictionary should not contain any other selectors and it should not have any superclasses.

The implementation of **doesNotUnderstand:** in class *Proxy* creates an Ethernet packet containing the original message and forwards it to the machine containing the remote object. The *Proxy* contains information in its instance variables about where the remote object resides. It is easy to encode the selector of the message in the Ethernet packet since it is purely textual. Storing the arguments to the message is a bit more difficult because they are structured objects of arbitrary complexity. *Proxy*

uses a structure storage package developed by Chris Hibbert to produce a linearized representation of each argument.

Figure 1 illustrates the flow of a single message send across machine boundaries. Upon receipt of a message at a proxy object, the following actions occur.

- a) The proxy instance forms a representation of the message (both the selector and the arguments) suitable for transmission to the remote machine,
- b) the proxy transmits the representation, and the sending context blocks waiting for a result.
- c) The remote object receives the representation of the message,
- d) extracts the message selector and arguments,
- e) and executes the message send as though it originated on the same machine.
- f) The result of the message send is then acquired,
- g) and a representation of it is formed and returned to the originating machine.
- h) The proxy object, using the returned representation, reconstructs the result object and returns it to the sender of the original message send.

This simple model illustrates the fundamental ideas behind transparent forwarding but it raises several significant issues. We will discuss problems involving object identity, primitives, efficiency, garbage collection, and the Smalltalk debugger. These issues will be addressed in a revised model for transparent forwarding.

In single-machine Smalltalk systems, the message == is utilized extensively to quickly determine if two variables refer to the same object. Often this test is implemented by comparing the object pointers referred to by the variables. In a multi-machine Smalltalk system, one of the variables may be a proxy for a remote object, and comparison of the object pointers will be inadequate. One way to implement the same semantics in a multi-machine Smalltalk would be to declare a very large object space which is somehow managed across machines. Of course, such a decision violates our goal of minimal change to the virtual machine to implement Transparent Forwarding. So, another solution must be sought.

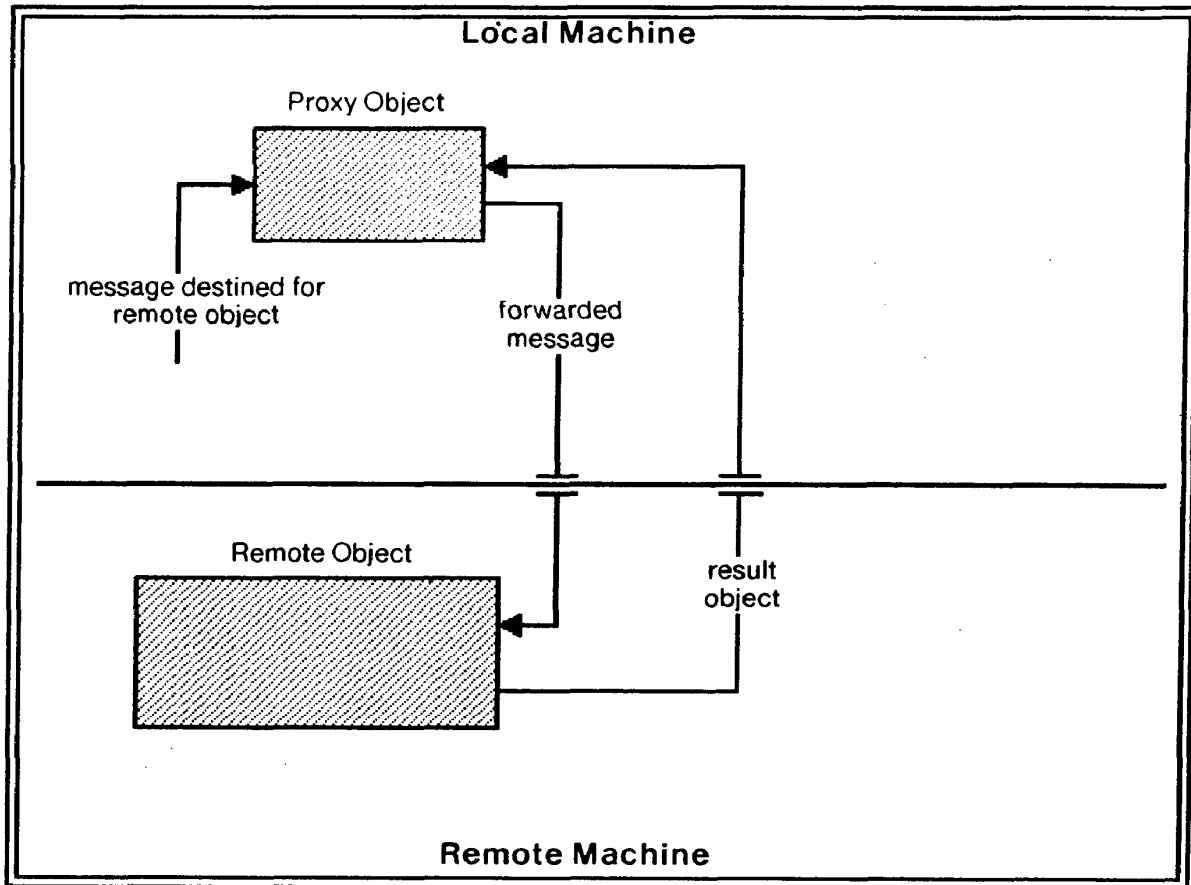


Figure 1: Message Send/Return for the Simple Model

A related problem arises because we have disjoint class hierarchies on several machines. How can we ensure that an object will have the same functionality when viewed from another machine's perspective? Should we automatically distribute the results of compilations to all participating machines?

Another problem with proxies is that many Smalltalk primitives (for example, BitBlt) expect their arguments to be well-behaved (that is, the arguments are expected to be of certain classes or in a specified range of values). Finding a proxy in the place of such an argument will cause primitive failure because the virtual machine will have no idea how to resolve the proxy. How may we facilitate better interaction between primitives and proxies?

In order to be useful, any programming mechanism must have acceptable performance. Involving a network in remote message sends is certainly going to cause them to be slower than their counterpart single-machine message

sends. How can we make remote message send more efficient?

The simple model presented above does not deal with cross-machine garbage collection. In order to deallocate a shared object we must ensure that there are neither local nor remote references to the object. How may the local machine efficiently determine if there are outstanding remote references to an object?

Forwarding is based on the Proxy instance understanding only the message `doesNotUnderstand:` and having no superclasses. Unfortunately, the current Smalltalk debugger relies on classes having valid superclass fields so that messages like `instVarAt:` can function (this is primarily caused by the fact that class `Object` has extensive protocol which is inherited by all other classes). During the debugging of the forwarding code itself, the problem under perusal will only be aggravated by attempting to forward messages sent by the debugger to Proxy objects. How should we deal with this tension

between a desire for transparency and the apparent need for modifications to the debugger?

5.0 Transparent Forwarding -- A Better Model

We now describe some solutions to the problems described in the simple model. These solutions lead us to an improved model for the transparent forwarding of messages. We begin with a solution to the efficiency problem

5.1 Migration

If we send several messages to a remote object, the overhead of linearizing and transmitting messages may become quite high. It might be better to move the object from the remote machine to the local machine so that future message sends occur locally. We term such movement *migration*. The reader should note that migration may violate the goal of transparency: the source code (or perhaps some higher function) must somehow reflect the need to move an object from one machine to another. The idea of migration leads to questions about whether arguments should be passed by copying, by proxy, or by actual movement. Additionally, we should consider read-only variables and concurrent access to shared objects.

First, let us examine various parameter passing schemes which we might use. There are four possibilities: pass-by-value, pass-by-reference, pass-by-proxy, and pass-by-migration. Pass-by-value is fairly simple: we ship a representation of the object to the remote machine which in turn reconstructs the object. Because the compiler prevents assignment to formal parameter variables, pass-by-reference does not occur in Smalltalk, and we need not consider it. Pass-by-proxy is also fairly simple. Rather than linearizing the object for reconstruction, we pass a proxy for the object and any messages which are then sent to the proxy are automatically forwarded to the remote object. In pass-by-migration, we move an object from one machine to another, leaving a proxy object in its prior home.

Immutable (or read-only) objects are interesting because they make safe the replication of objects on multiple machines. In a shared environment, replication of objects provides two useful benefits. In the event of network partitioning essential information is available on more than one network node, and the time required to execute a computation which previously involved the network will be drastically reduced. This important issue does not occur in the current single-machine Smalltalk systems and

has not yet been well addressed in the present work but will no doubt be an important issue in the future.

By fiat, instances of the classes `SmallInteger`, `LargePositiveInteger`, `LargeNegativeInteger`, `Float`, `Character`, `String`, `Symbol`, `UndefinedObject`, `True`, and `False` are passed by value from one machine to another. With the exception of strings, this policy follows the Smalltalk conventions that these objects are immutable. Strings are passed-by-value for efficiency reasons. To determine how to pass an object, the forwarding system consults a table. Although this table is currently defined at class initialization time, it would be easy to add protocol to allow run-time changes depending on the specific needs of an application. It is also possible to add a capability to specify the manner in which individual objects should be treated if their desired behavior is different than other instances of the same class. Such a table may help to minimize the aforementioned tension between source code transparency and the need to specify which parameter passing mechanism we would like to use.

5.2 The PolicyMaker

Recall that we want Proxy objects to be very simple, implementing only the message `doesNotUnderstand:`. Yet their implied behavior is that they forward messages to remote objects and somehow determine which parameter passing mechanisms to employ. To address this tension between the desire for simplicity and the need for fairly complex behavior, we introduce a new entity which serves as a central clearinghouse to control all issues dealing with remote objects. This centralized control is represented by the class *PolicyMaker*. Each time a message is sent to a proxy, the proxy asks the PolicyMaker to deliver the message to the remote object. In this way, individual proxies need not record the current network location of a shared object: that is the responsibility of the Policymaker. PolicyMaker responsibilities also include the decision on whether to pass objects by value, by proxy, or by migration and whether to forward a message to a remote object or whether to migrate the object to the local machine for execution. Additionally, the PolicyMaker keeps track of open connections between machines. For each connection to a remote machine, the PolicyMaker creates an instance of class *TransporterRoom*. The TransporterRooms take care of communications protocols between machines as well as the linearization of messages and objects.

5.3 Primitive Solutions

There are several problems where primitives interact with proxy objects: object identity, class identity, and primitive failures.

To address the object identity problem, we would like to have unique identifiers which span machines. The unique identifier of an object and its proxies would then be identical. Where then should we keep such an identifier? Adding an extra field to the object header (fields in the object header are not directly accessible from Smalltalk) of every object violates our goal of an unchanged virtual machine. An approach briefly considered was to add a user-accessible field to the object body which would hold the unique identifier. This was deemed undesirable for two reasons: having the value be changeable by the user seemed dangerous, and not all classes have pointer-bearing instances. Such classes would thus have to have their unique identifiers contained in a table.

The current solution, which has its own associated problems, is to require each remotely accessed object to have a unique identifier (called a glop or global object pointer) which is maintained in a table, the Glop table. The Glop table, which is invisible to the user, maps object pointers to glops. An object's object pointer can be found by sending the object the message `asOop`. The `==` message in class `Object` is then modified to test for equal glops. One additional change is required for this solution. The current Smalltalk-80 compiler emits a special bytecode for `==` so that it is never looked up via the normal method lookup scheme. A simple change to the compiler causes a normal message send bytecode to be emitted rather than the special bytecode. The performance degradation from this change, while significant in the `==` benchmark, tends to be only slightly perceptible in general system interactions. This degradation could be significantly reduced by redefining the `==` primitive in the virtual machine. Currently, the code which maintains the Glop table is implemented in Smalltalk and not in the virtual machine, resulting in non-atomic table maintenance. Pushing Glop table management into the virtual machine seems the only long-term solution.

To determine if two classes on different machines implement the same message protocols, we currently test for equality of their names. This test, while strong enough for a prototype, is inadequate for a production system. This is an area for future research.

In order to avoid primitive failure in the presence of proxies, affected methods which invoke primitives are

recoded. Should the primitive fail, the `PolicyMaker` is asked to migrate the object to the local machine and the primitive is then retried.

5.4 Debugger Revision

As previously discussed, there is a tension between our desires for minimal changes to the system and the need for the debugger to know whether it is dealing with normal Smalltalk objects or proxies. A workable solution appears to be allowing the debugger to detect when it is dealing with a proxy object and to have it somehow query the user as to whether debugging messages should be sent to the remote object. Smalltalk inspectors, which allow the user to view and change the internals of Smalltalk objects, are similar to the debugger and also suffer from the need to know whether they are dealing with proxies.

5.5 Distributed Garbage Collection

Along with the idea of shared, distributed objects comes the problem of distributed garbage collection. How can the local system economically determine whether there are any remote references to a local object? We can actually use the virtual machine's object reclamation mechanism to do part of the work for us.

If an object has ever been shared with another machine, it has an assigned glop which will stay with the object for the remainder of its existence. The glops are maintained in a global 'Glop table' (of object table size) which is indexed by the value returned from sending the message `asOop` to the shared object. Each table entry is either nil (meaning no glop assigned) or else the corresponding object's glop.

We can arrange to keep the Glop table up-to-date by modifying the (fortunately few) messages in the system which invoke the object creation primitives. Each time an object is created the Glop table is accessed and the appropriate entry is set to nil, indicating that this object has no glop. If we later hand out a remote reference to this object, we assign it a glop in the Glop table. To ensure that the shared object, while still being referenced by a remote machine, is not reclaimed locally by the virtual machine's reclamation mechanism, we add an extra reference to the shared object. To provide this increased reference count, each `TransporterRoom` maintains a dictionary of shared objects which are known to be (or more precisely, to have recently been) referenced from another machine, and this dictionary serves the purpose of adding the extra reference to the shared object. It is up to the remote machine to tell us when it no longer has any proxies for the shared object.

Now shift your point of view to the remote machine. Whenever a proxy no longer has local references, we want it to be reclaimed by the virtual machine reclamation process. Each PolicyMaker maintains a dictionary which maps glops to their respective proxy objects. In order not to create extra references to proxies, this dictionary refers to them by the value returned from sending them the message `asOop`. Thus, when there are no longer any references to a proxy object it will be reclaimed automatically via normal virtual machine memory management. Because this happens outside the scope of the Smalltalk language, there is no way to arrange for the glop-to-proxy map to be automatically updated as a side effect of reclamation. Instead, this updating must be done periodically by the PolicyMaker.

To determine if a glop-to-proxy map entry still denotes a valid proxy, the PolicyMaker checks the Glop table entry corresponding to the proxy. If the entry is nil, the proxy has been deallocated (i.e. some new object has been created with that Oop) and the map entry is invalid. If the glops match, the PolicyMaker further checks the validity of the map entry by sending the message `asObject` to the value in the Glop table entry for the proxy. If this results in an error (i.e. the proxy has been deallocated and no object is currently allocated for that Oop), the map entry is invalid, and the Glop table also needs to be updated.

Periodically, a background process of the PolicyMaker scans the glop-to-proxy dictionary and tests each entry with the above validity check. If an entry fails the test, it means that all references to the proxy were destroyed and the proxy object freed. The glop that was mapped to that proxy is remembered and the appropriate local TransporterRoom is notified. The local TransporterRoom then informs its counterpart TransporterRoom (on the other machine), which in turn drops its reference to the shared object.

Now shift your point of view back to the original machine. Should the reference count of the shared object now drop to zero it will, of course, be reclaimed by the normal virtual machine mechanism. In a single-machine Smalltalk system, freeing one object can cause many other objects to be freed. In this multi-machine system, reclaiming a proxy can cause a similar cascading effect: the local and remote machines may bounce back and forth informing one another of newly unreferenced shared objects. As mentioned previously, a portion of this glop management is somewhat unsafe: glop management should be done at object creation time, atomically and within the virtual machine. In actual practice, no problems have been encountered.

Objects participating in cross-machine circular structures are not reclaimed by this scheme, although a simple mark-sweep cross-machine collector seems feasible.

5.6 Flow of Control

Let us now examine the flow of control involved in sending a message to a remote object (see figure 2).

- a) Someone sends a message to what they believe to be a local object but which, in reality, is an instance of Proxy,
- b) the Proxy instance appeals to the system's PolicyMaker,
- c) the PolicyMaker determines whether the arguments of the message should be sent by copying or by proxy to the remote object,
- d) the PolicyMaker finds or establishes a connection to the remote machine, and
- e) the TransporterRoom linearizes and transmits the message to the remote machine.

Upon receipt at the remote machine:

- a) the TransporterRoom reconstructs the message, and
- b) sends a `perform:WithArguments:` to the receiver.
- c) The PolicyMaker considers the resulting object (see figure 3) and determines whether to return it by value or by proxy, and
- d) the TransporterRoom linearizes the result object and returns it.

On receipt of the return linearized return value at the local machine,

- a) its TransporterRoom reconstructs the result object, and
- b) returns it to the sending context.

6.0 The Current Implementation

The current implementation consists of several new classes and a few changes to existing methods in the system.

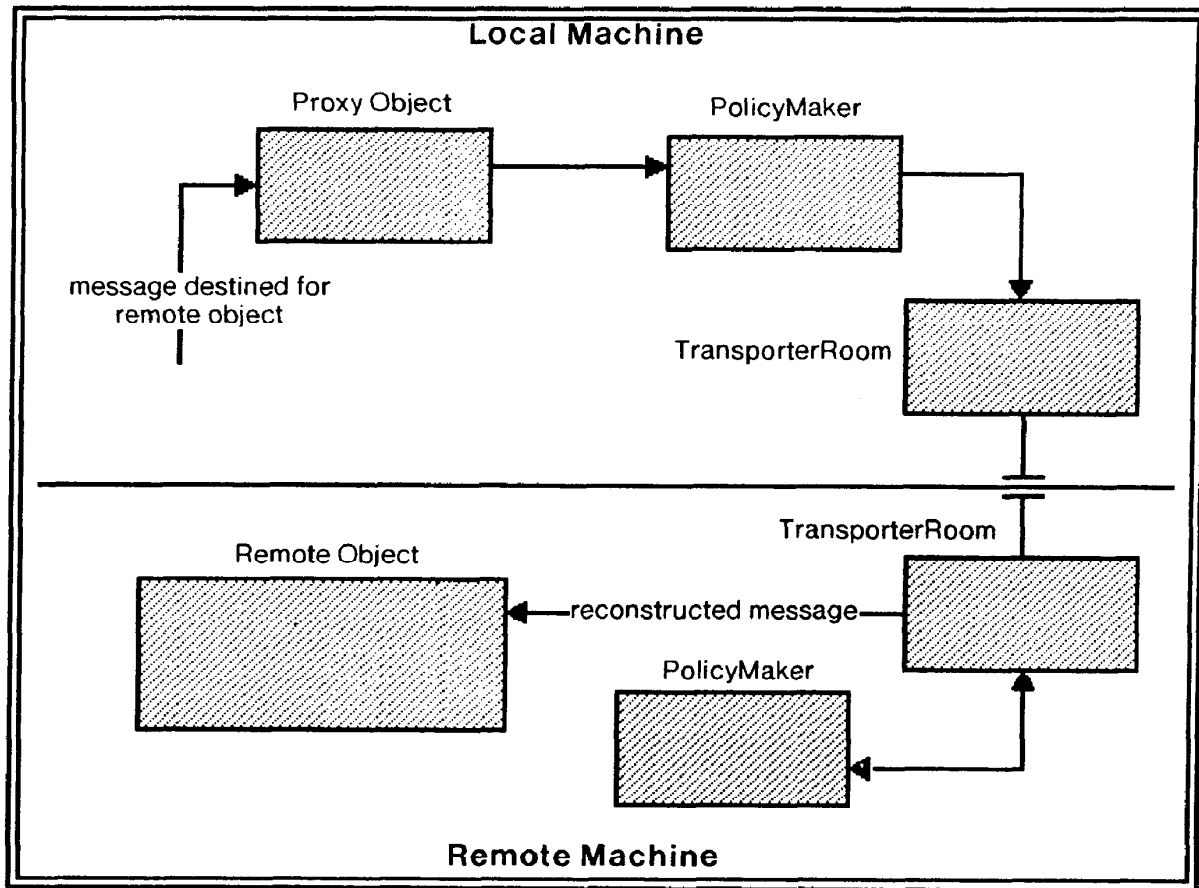


Figure 2: Message Send from Local to Remote Machine

6.1 New Classes

MySemaphore

MySemaphore, a subclass of Semaphore, provides mutual exclusion for critical regions of code. It mainly exists because of design problems in the virtual machine specification of Semaphores. (These problems have to do with an interaction between semaphores and user interrupts. If a process is blocked waiting on a semaphore and is interrupted by the user, it will proceed as though the semaphore had been signaled when, in fact, it has not.) Additionally, two new messages **signal:** and **wait:**, are defined. Each takes a string as an argument and, by enabling some code in MySemaphore, records in a global object the exact order of waiting, acquisition, and release of mutual exclusion semaphores. This approach has made the debugging of occasional deadlock problems much simpler.

Proxy

As previously mentioned, instances of Proxy should implement only the message **doesNotUnderstand:**. Unfortunately, a few other messages are necessary. These messages are necessary to set and retrieve instance variables of the Proxy object. The selectors for these messages are all preceded by the word 'secret' (e.g., **secretTransporterRoom**) and are intended only for use within the forwarding system, not by other classes. A few other messages are also defined by Proxy. For example, some messages dealing with object identity (e.g., **==** or **~**) need not be forwarded to the remote machine. Additionally, a few messages dealing with the Smalltalk user interface, such as **inspect** or **basicInspect**, are overridden. Smalltalk objects are often located in structured objects such as dictionaries by utilizing a hash value based on the object's object pointer. Since this value will be different for the proxy and the actual object, messages dealing with hashing such as **hash** present some interesting

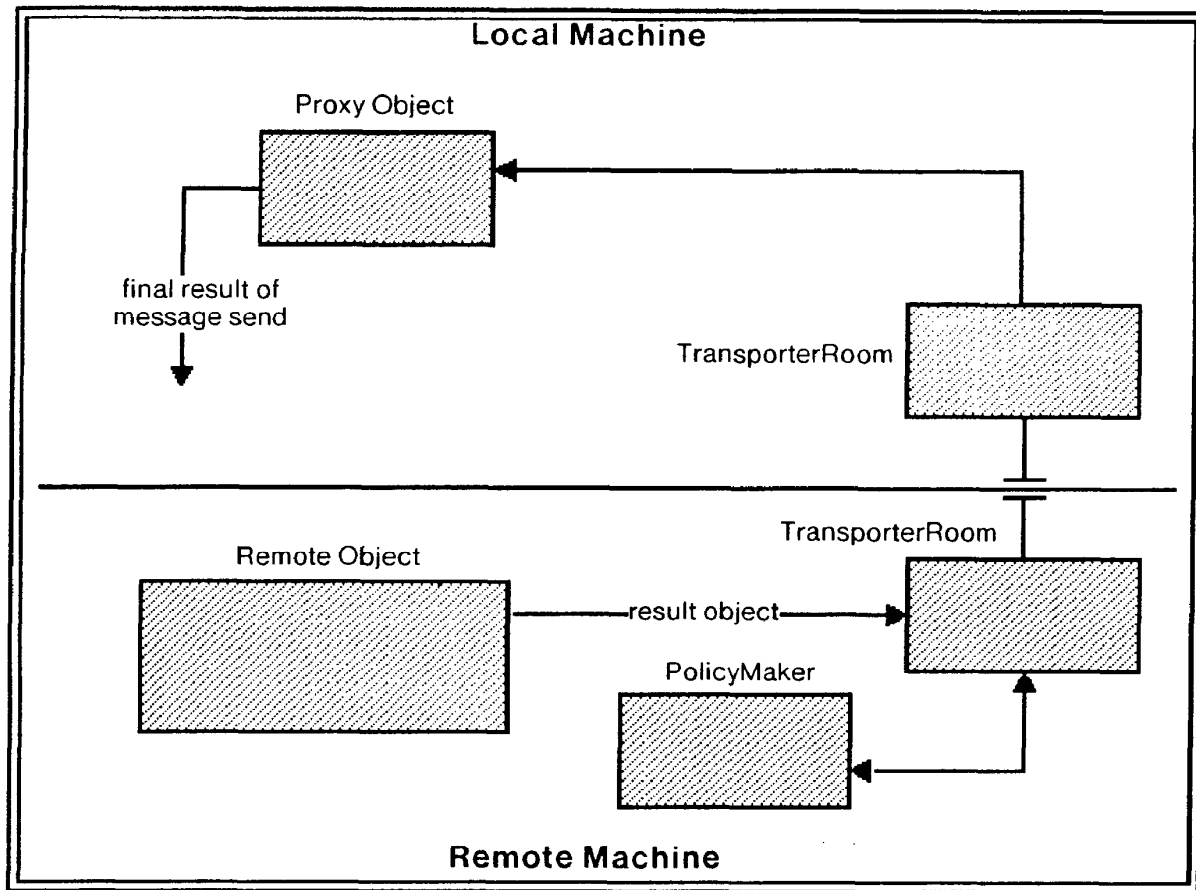


Figure 3: Returning Result Object to Complete Message Send

problems. The PolicyMaker has a table of classes which are known to implement **hash**. When that message is sent to a Proxy, the PolicyMaker consults its table and decides whether to forward the message to the remote machine (in the case of classes which inherit but not override the **hash** message) or to perform it locally (N.B., the current system does not automatically update this table of classes known to implement **hash**).

Finally, in the current implementation, Proxy is a subclass of Object (as mentioned earlier, this is necessary because the debugger fails when encountering classes with nil superclass fields). When the forwarding system is installed, Proxy overrides all methods defined in class Object, replacing them with an appeal to the PolicyMaker for assistance.

TransporterRoom

Instances of the class TransporterRoom are the primary agents in the communication protocol between any two

machines and handle the linearization and reconstruction of messages and objects. When the PolicyMaker is started on a machine, an instance of TransporterRoom is created and placed in listening mode awaiting a connection from another machine. When such a connection occurs, each machine forks a dispatch process. This process loops continuously to field requests from the remote TransporterRoom. It forks a process to handle each such incoming request (it is essential, of course, to fork the requests otherwise the dispatch process may become blocked and a multi-machine message send deadlock will occur).

6.2 Modifications to Existing Methods

In addition to the new classes discussed above, a few changes are necessary to existing methods. These are:

VariableNode class initialize

Change == from a special bytecode to a normal message send bytecode.

Behavior

Change basicNew and basicNew: to nil out the appropriate glops array entry.

Change the manner in which two classes check for equality

ContextPart

Modify the printing of contexts so that it is clear in the debugger when you are dealing with a proxy.

Object

Change the definition of ==.

Add messages for isProxy, secretGlop, and secretAsOop.

6.3 Communications Protocols

When several machines are sharing objects, each has one instance of TransporterRoom per remote machine. These objects manage the communications protocols and field requests from their respective remote machines. Currently, the various types of requests are:

locate a global,

send a message,

reply to a message send,

create an extra reference to an object,

destroy an extra reference to an object,

reply to a message send with a third party object,

and acknowledgments to the above requests.

We now consider each of these requests from the point of view of a TransporterRoom which has just received the request. Note that each argument is received in the linearized form and each result object is transmitted similarly.

Locate a global

inputs: a Symbol

output: glop of global object

Assuming that one starts from an image which contains no references to remote objects, one must begin by somehow obtaining a handle on an object residing on a remote machine. A remote machine is making such a request of us, so we reconstruct the global name, access our Smalltalk dictionary and return either a proxy or the actual object (depending, of course, on the policies established by our PolicyMaker).

Send a message

inputs: the glop of the receiver

argument count

each argument

A proxy on the remote machine has been sent a message and the message has been forwarded to us for execution. Reconstruct the message and any arguments, then perform the message send returning a result to the remote machine. Note that performing the message send may entail this machine forwarding messages to the requesting machine or still other machines.

Reply to a message send

output: the result of a prior message send

In response to a prior "Send a message" request, a result object is now being returned from the remote machine.

Create an extra reference to an object

inputs: the glop of the object

output: an acknowledgment

A remote machine is about to be given a proxy for an object residing on this machine. Record this fact.

Destroy an extra reference to objects

inputs: any number of glops

output: an acknowledgment

The remote TransporterRoom has recently performed a proxy reclamation and these glops were found to no longer be needed. Remove them from the local table of objects known by the remote machine. After the last such glop has been processed, this machine will notify its local PolicyMaker that the remote machine just finished a proxy reclamation so that it may initiate a local reclamation.

Reply to a message send with a third party object

inputs: the network address of the remote machine
the glop of the object
output: an acknowledgment

The local machine (call it machine *L*) has previously transmitted a `send a message` command to a remote machine (call it *A*). *A* is about to return the result object, but finds that it is a Proxy for an object residing on another machine (call it *B*). Machine *L* must contact *B* and ask it to create, on *L*'s behalf, an extra reference for *B*'s object. After an acknowledgement from *B*, tell *A* to proceed and it now safely returns a Proxy for the object residing on *B*. This message is necessary to prevent a race condition in the following scenario:

A hands out (to *L*) a proxy to an object residing on *B*.

A drops all references to its Proxy for the object.

A runs its proxy reclamation process and tells *B* it no longer needs its object.

B drop all references to the object and it is reclaimed by the virtual machine.

L now informs *B* that *L* needs a forwarder to its object (which no longer exists).

The communications protocol between two TransporterRooms is fairly simple. Each request consists of several fields. Every request has a command byte specifying the type of the request, and a request identification (a sequentially allocated integer). Depending on the type of request, additional fields may be present. For example, the variable portion of a `send a message` request consists of a glop for the receiver, the number of arguments, and then each argument. The actual transmission of packets is the responsibility of Ethernet protocols included in our research system.

7.0 Future Research

There are two areas requiring immediate attention. We need to determine how exception handling should be propagated between machines and, perhaps more importantly, we need to define the desired semantics for snapshots. Currently a Smalltalk user makes a snapshot when it is desirable to provide a checkpoint of a certain system state. What should such an action mean in a world of shared objects?

Single-user Smalltalk systems generally have a need for increased robustness. Transparent forwarding puts additional pressure on this need because of network partitioning and crashed nodes. In the presence of network difficulties, the current Smalltalk system's belief that a message send will produce a result object is doomed to disappointment. Additional work will need to be done in both forwarding and the rest of the system to change such expectations.

There are certainly other design choices which could have been made in this research. The primary change we would make is to have one PolicyMaker per Proxy object. This would simplify the instance variables and protocol of the proxy and some of the bookkeeping within the PolicyMaker because each PolicyMaker could keep track of the manner in which its particular proxy should be passed as an argument or treated as a message receiver (rather than maintaining this information in dictionaries). There would likely still be a global PolicyMaker to maintain network connectivity information and to enforce global policy decisions. An interesting change would be to make the transparent forwarding code recursive on itself. Currently, requests are transmitted from one machine to another and each TransporterRoom decides, based on the command byte, whether the request is a remote message send or is a request by action by the forwarding system. By dispensing the glops for the TransporterRoom and PolicyMaker objects, all requests would become first-class message sends and the communications protocols would be simplified.

In the area of migration, an interesting question is: at what depth should we stop the migration of objects? That is, if we are asked to pass an array to another machine and somehow discover that we should pass it by migration do we migrate the array substituting a proxy for each array element (so-called *shallow* migration), or, pass each array element by also migrating it (so-called *deep* migration). If we choose deep migration do we also pass the objects referred to by the array elements by deep migration? At what objects do we stop the deep migration? How may we parameterize such migration depth? We expect to address these issues in the near future.

Now that we have started to share objects, we need to examine how to manage the sharing. Are there approaches from the database and distributed systems fields which apply to a shared object-oriented world? Since each object is an instance of a class, how can we keep class protocols up to date across machines? If one user changes a method definition in a class, should that change be reflected in other machines? This is a deep

issue touching upon sharing, object identity, version management, and immutability of objects.

If we allow changes to the virtual machine, several parts of the implementation change in significant ways. Much of the work related to multi-machine garbage collection could be done more efficiently by pushing this task into the virtual machine and, of course, putting the code for the == primitive back into the virtual machine would speed up that operation. Adding support in the virtual machine for immutable objects would enable local caching of shared objects by individual workstations. By making use of the networking code in the underlying operating system it is expected that we would experience a significant performance improvement in inter-machine communication.

Most importantly, by changing the virtual machine, we can produce truly transparent forwarders which are completely invisible from the Smalltalk level and which are useful in both this and numerous other applications. These other applications include better Smalltalk processes, lazy and eager evaluation, and a more general concept of providing "wrappers" around objects. We are pursuing the necessary changes to the virtual machine to experiment with such forwarders.

8.0 Conclusions

One of the most pleasant outcomes of this work was that in real applications of forwarding few problems were encountered. An existing application, an on-line shared phone directory, utilized a home-grown communications protocol to enable shared access. By changing one line of code we bypassed the old protocol, and enabled direct, simpler, shared access from remote machines via forwarding. The forwarding system performed correctly on the first attempt. Another experiment, opening a browser on a remote model, required no source changes and worked on the second attempt.

There were also some interesting surprises along the way. The user interface for Smalltalk was built with the assumption that one user would use the interface at a time. What happens when you open a browser on one machine and open a browser on a second machine *but on the same model*? Generally it works, but a few strange effects are observed. Because the Browser model is poorly factored, any attempt to pop-up a menu on a remote machine causes the browser on the machine containing the model to pop-up the menu. Similarly, inspecting an object will cause the inspector to show up on the machine where the object resides rather than on the machine where the user requested it. There are two

problems here, neither are related to Transparent Forwarding. The first is that the Model-View-Controller (MVC) paradigm generally assumes there will be only one view per model; the second is that some user interface code violates the MVC framework.

From a fairly simple idea, this work has provided a wellspring of insights and new ideas. It has, for example, illustrated areas in which the single-user mind-set of existing Smalltalk systems may need to change. And it has also provided us with new ways to view both information sharing and encapsulation.

9.0 Acknowledgments

I would like to thank many people for their assistance in both the research and the preparation of this paper. Bob Flegel early saw the advantage of this approach to object sharing and suggested many improvements to this paper. Randy Smith and Mark Miller contributed ideas which led to the early designs for forwarding. Frank Zdybel took a keen interest in the work and made suggestions which have much improved this exposition. Alan Purdy helped to clarify early ideas and pointed out areas which needed clarification in the paper. Dave Robson also contributed ideas for the organization of this paper. Chris Hibbert supplied the off-image object storage mechanism. I would also like to thank the System Concepts Laboratory for supporting this work.

10.0 References

- [1] Adele Goldberg and Dave Robson, Smalltalk-80: The Language and its Implementation, Addison-Wesley, 1983.
- [2] J. E. White, *A High-level Framework for Network-based Resource Sharing*, AFIPS Proceedings of the National Computer Conference, pages 561-570, June 1976.
- [3] Bruce Jay Nelson, *Remote Procedure Call*, Xerox PARC, CSL 81-9.
- [4] Dominique Decouchant, *Design of a Distributed Object Manager for the Smalltalk-80 System*, Conference Proceedings, OOPSLA'86
- [5] Alan Purdy, David Maier, Bruce Schuchardt, *Integrating an Object Server with Other Worlds*, ACM Transactions on Office Information Systems, V5, #1, January 1987

Smalltalk-80 is a registered trademark of The Xerox Corporation.