MASSACHVSETTS INSTITVTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A.I. Memo No. 1547 February 20, 1993

MIT SchMUSE: Class-Based Remote Delegation in a Capricious Distributed Environment

Michael R. Blair, Natalya Cohen, David M. LaMacchia, Brian K. Zuzga

This publication can be retrieved by anonymous ftp to publications.ai.mit.edu.

Abstract

MIT SchMUSE (pronounced "shmooz") is a concurrent, distributed, delegation-based object-oriented interactive environment with persistent storage. It is designed to run in a "capricious" network environment, where servers can migrate from site to site and can regularly become unavailable. Our design introduces a new form of unique identifiers called *globally unique tickets* that provide globally unique time/space stamps for objects and classes without being location specific. Object location is achieved by a distributed hierarchical lazy lookup mechanism that we call *realm resolution*. We also introduce a novel mechanism called *message deferral* for enhanced reliability in the face of remote delegation. We conclude with a comparison to related work and a projection of future work on MIT SchMUSE.

This report is an editted and reformatted version of a report that appeared in the Proceedings of the 1995 Lisp Users and Vendors Conference, Cambridge MA, August 14–18, sponsored by the Association of Lisp Users (ALU). It was voted and selected best paper from the Student Presentation Contest by the conference attendees.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-83-K-0125. Computer facilities to support the prototyping of this project were also provided, in part, by a generous equipment donation from Hewlett-Packard Company to the MIT Department of Electrical Engineering and Computer Science.

1 Introduction

MIT SchMUSE is a Scheme-based [Clinger & Rees 91] [IEEE 91] Multi-User Simulation Environment based on a concurrent, distributed, delegation-based object-oriented language with a persistent object store. It was originally conceived as a teaching vehicle for the introductory programming course at MIT [Abelson & Sussman 85].

The project's main goal was to provide a Scheme-based concrete introductory case-study of computer systems issues such as concurrency, distributed computing, persistence, recoverability, transactions, reliability, and delegation-based object-oriented programming system (OOPS). We did so under the guise of implementing a multi-user interactive adventure game, mainly because that is a common setting very familiar to most students. It was not our intention, however, to limit ourselves to this application domain. For example, we envision the SchMUSE as being a practical prototype system in which to implement a distributed interactive virtual office space or a distributed object-oriented database. Nonetheless, for reader accessibility, the examples in this report reflect a simulated world setting. Moreover, this report focuses exclusively on the novel features of the SchMUSE language design and its functionality rather than on distributed systems issues *per se*. To wit, we view our primary contribution as having explored several novel language systems issues that arise in distributed *object delegation* with regards to language implementation rather than in having tackled the distributed systems issues that arise merely by virtue of being a distributed computing environment, such as message transaction atomicity and concurrency coordination.

We chose to implement our system in MIT Scheme [Hanson 91], a dialect of the Lisp family of programming languages. Our object system is modelled after the delegation-based OOPS style of [Adams & Rees 88], primarily because that is the language in which our course is taught. *None of our results are specific to* Scheme*:* we could equally as well have implemented this system in Smalltalk-80 [XLRG 81], CLOS [Bobrow *et al.* 90], Dylan [Apple 92] or even C++ [Stroustrup 86].[1]

1.1 Application Context

The laboratory setting in which we envision MIT SchMUSE being exercised has greatly influenced its design. Specifically, our course lab is comprised of a local area network of 46 high-performance diskless workstations (HP 720s) evenly divided across 2 high-powered centralized network file servers (HP 750s). Ultimately, however, we envision MIT SchMUSE spreading across the Internet as an extremely distributed global network of potentially collaborating simulation sites. For example, now that MIT provides network access in the campus dormitories, we would not be surprised if our students choose to leave their otherwise idle PCs active as SchMUSE servers, running atop publicly available DOS MIT Scheme [DOS Scheme].

The classroom mode of operation for students SchMUSE-ing is envisioned to be as follows: a student logs into an unused workstation and brings up a local SchMUSE client session. During this session, the student can interact (via TCP/IP sockets) with objects on either of two highly available centralized file servers, likely creating a few new instances of standardized classes on the servers, subject to a modest space quota.

More interestingly, the student can also extend the simulation system with new classes and objects on their client machine. These new creations can then be made available to other students in the lab by allowing other clients in the lab to establish server connections to their client. In this way, we blur the distinction between server and client since we allow clients to themselves act as servers to other clients. This permits very interactive collaboration throughout the network. A student on one workstation can build on the efforts of students on several other workstations, and vice versa, to build elaborate team experiments.

When a student logs out of a workstation, unfortunately their client/server sessions must be terminated (to make room for the next student), but their state can be dumped to persistent storage. Specifically, their code, as well as the state of their local object instance database, can be dumped as files to a personal floppy disk or `ftp`'d to a remote file server.

When the student returns to the lab (or to their dorm room), all their SchMUSE state can be restored from their last state dump and their session can continue. Of course, some of the other student servers they were interacting with may have also moved to a different workstation (*e.g.,* so the student can sit closer to someone else they are collaborating with or because their reservation on the machine they were using had expired) or they may be altogether unavailable (*e.g.,* the student who was running the server session may be temporarily in transit to a new location or dormant during sleeping and eating periods).

1.2 Outline

The following sections deal with the difficulties of mitigating this sort of capricious network environment of itinerant virtual servers, where servers can become unavailable and where servers can move among workstations, both with disturbing regularity. First, however, we establish our language context by describing the SchMUSE object system in section 2. In section 3, we proceed to describe how remote delegation is supported in our object system. In section 4, we show how imposing class structure on our delegation system enables various performance enhancements for remote delegation. Section 5 introduces *message deferral* for improved reliability in a system where servers are itinerant. Section 6

[1] Of course, delegation-based dialects of these languages would be required. The exemplar-based dialect of Smalltalk [LaLonde 86] could be used. Delegation-based extensions can be made fairly painlessly to CLOS and Dylan by careful use of the Meta-Object Protocol (MOP) [Kiczales *et al.* 91], as demonstrated in [Blair & Maessen 93]. Finally, the feasibility of delegation in C++ was demonstrated in [Johnson & Zweig 91].

relates our language design to other work in the field. Section 7 emphasizes the unique contributions of this work. Finally, in section 8, we conclude by outlining future work in the project.

2 The SchMUSE Object System

The sole means of object interaction in MIT SchMUSE is through message passing [Agha 86]. Methods are invoked by passing messages to objects, and instance slots are accessed via messages. (It will become clear below why we choose to distinguish object slot accesses from other, more compound methods).

Unlike prototype-based delegation systems [Lieberman 86], our system employs class-based delegation. Specifically, our objects are created as instances of classes, where each class declares the local variables and methods defined on instances of the class as well as declaring any parent's classes.[2] When an instance of a class is generated, a chain of partial objects is made which corresponds to the inheritance chain prescribed by the object's class structure. An example is sketched in Figure 1. Each partial object (box), hereafter called a *node*, contains slots for the local variables as well as pointers to node instances of the node's parent classes. Notice that a node taken with the transitive closure of its parent nodes constitutes an object instance. A node is said to *delegate to* its parent objects. In this way, an object is represented as a directed acyclic graph (DAG) of nodes that directly reflects the inheritance structure of its class definition.

Note from Figure 1 that slot names, in addition to method names, can shadow one another. For example, persons and students both have a nickname slot. In addition, as our figure suggests, an object can be the parent of more than one child node. In our figure, the person is a common parent of both a student node and an employee node.

This very general framework for sharing is what makes delegation-based inheritance most compelling for distributed systems. Specifically, in a distributed setting, each object node could reside on a separate workstation. We call this distribution of delegate nodes across several sites *remote delegation.* Note that traditional class-based inheritance systems such as Smalltalk [XLRG 81], CLOS [Bobrow *et al.* 90], and C++ [Stroustrup 86] provide no such distributed sharing. Instead, they flatten objects into one long array of slots, one per each unique slot name. Consequently, slot shadowing is not supported in these systems. Also, slot sharing among multiple children is not directly supported.[3]

Delegation-derived sharing allows for complex patterns of centralized sharing with privacy control. In our example, for instance, centralized sharing is illustrated by the person being shared by both the student and employee nodes. By centralizing the sharing, privacy can be enhanced; for instance, the employee child may be granted access to the person's social security number while the student child may be denied this information. Moreover, by permitting multiple independent children we in effect provide multiple independent views of the same essential object. Specifically, an application that knows about the student view of our person may be unaware of our employee view. Thus, a student's advisor may be unaware that s/he is consulting on the side. This too is an important form of privacy.

3 Remote Delegation via Globally Unique Tickets and Realm Resolution

All objects in the SchMUSE are passed to methods via object reference. In order to support remote delegation, these object references must embody *globally unique identifiers* [Leach *et al.* 82].

3.1 Globally Unique Tickets

In MIT SchMUSE this need for globally unique IDs for object references is accomplished by implementing every object reference as an *object ticket.* Similarly, class references are implemented as *class tickets.* Collectively, object and class tickets comprise what we call *globally unique tickets.* The grammar for these GUTs of the SchMUSE object system is shown in Figure 2.

The first field of each ticket is not strictly necessary: they are introduced primarily for debugging support, although we will later exploit the `ClassTicket` within an `ObjectTicket`. A `btime` is the encoding of a millisecond real-time clock reading of when the entity was born, and `bmachloc` is the encoding of the "birth machine location", *i.e.,* the network address of the machine on which the entity was created. Together, these two data provide a globally unique identifier for every object in the network.[4] They do not, however, provide location information if we allow SchMUSE sessions to migrate about the network. This is where the `RealmTicket` comes into play.

Each server/client session maintains its own namespace of classes and instances in the underlying Scheme session. A class table and object table are maintained to map class tickets to classes and object tickets to object instances. Since SchMUSE sessions are itinerant, these namespaces are consequently mobile. We conceptually divide each namespace into *realms* of course-grained collections of objects. For example, a student may be developing an adventure game simulation with a Blade Runner Realm and a Time Bandits Realm. Each separate realm can be dumped to disk and transported to another workstation separately.

When an object class or instance is created, it is always created *in some realm.* This realm information is encoded by the `RealmTicket` inside the entity's ticket.

[2] We permit multiple inheritance. For simplicity, we follow the mechanism of left-to-right topological linearization of multiple parents described in [Snyder 86] and used as the default in CLOS [Bobrow *et al.* 90] and Dylan [Apple 92].

[3] Some contortions involving indirection through shared cell data is possible. This is what we resorted to in [Blair & Maessen 93].

[4] Note also that they obviate the need for distributed clock synchronization: so long as each workstation assures that its own clock progresses strictly forward, no two distinct entity tickets can collide in SchMUSE space-time.

```
.--------------.    .----------------.
| MobileObject |    | AnimateObject  |
|--------------|    |----------------|
|  slot: place |    |  slot: asleep? |
|method: move  |    |method: hear    |
`--------------'    `----------------'
           /|\    /|\
            |      |
      .------------------.
      |     Person       |
      |------------------|
      |slots: soc-sec-no |
      |       possessions|
      |       nickname   |
      |methods: move     |
      |         say      |
      `------------------'
           /|\    /|\
            |      |
.------------------.  .----------------.
|     Student      |  |   Employee     |
|------------------|  |----------------|
|   slot: nickname |  |  slot: nickname|
|methods: move     |  |method: work    |
|         say      |  |        say     |
`------------------'  `----------------'
```

Figure 1: An example delegation-based instance.

$$
\begin{array}{rcl}
\texttt{ObjectTicket} & ::= & \texttt{ClassTicket} \times \texttt{btime} \times \texttt{bmachloc} \times \texttt{RealmTicket} \\
\texttt{ClassTicket} & ::= & \texttt{classname} \times \texttt{btime} \times \texttt{bmachloc} \times \texttt{RealmTicket} \\
\texttt{RealmTicket} & ::= & \texttt{realmname} \times \texttt{btime} \times \texttt{bmachloc}
\end{array}
$$

Figure 2: Globally Unique Tickets

When remote sessions access a local realm and receive copies of its local object tickets, this realm information is later used to locate the object again. That is, when a message is sent to an object ticket, if the object ticket's realm matches the realm we are in when processing the message, then the object must be local. The local object table is then used to map the object ticket to the desired object and the message can be processed on that local object. If, on the other hand, the object ticket's realm is not that of a local realm, then the message is forwarded to the machine where the remote object's realm resides.

3.2 Realm Resolution

Notice, however, that `RealmTickets` pointedly do *not* contain location information other than the birthplace of a realm. Thus, there is nothing in the ticket to declare where a particular realm actually resides in the network. For this we resort to a "Brazilian" [5] distributed hierarchical realm resolution/location mechanism which we have dubbed `Central Services`. Central Services is not a single centralized entity but, rather, a distributed hierarchical network. Each SchMUSE session which initiates server access does so by contacting some nearby authority and requesting access to Central Services. The authority designates some server within the network of active SchMUSE servers to serve as realm resolution authority for the new session. In this way, a variety of load-distributed resolution hierarchies can be initiated by the central authority, such as N-ary trees and H-trees [Leighton 92].

The server then informs the authority and its designated Central Services contact point of the realms which it is making available for server access. This information is then lazily distributed throughout the SchMUSE network on demand in a fashion similar to Internet namespace service. Should this contact point become unavailable, a new one can be requested from the nearby authority. The authority network, being the backbone of the SchMUSE, is fixed and distributed, much like Internet namespace service.

Each SchMUSE session, then, maintains a cache of its realm lookups to map `RealmTicket`s to real machine addresses. Of course, these address mappings will become obsolete when a remote server session terminates, so a time-out mechanism is implemented whereby an unsuccessful connection can resort to querying Central Services to see if the desired realm has moved. When a session is terminated, it is customary to therefore notify Central Services so that future inquires regarding the realm's location can be rejected without resorting to repeated time-out mechanisms.

[5] That is, inspired by the movie *Brazil*.

The details of this distributed hierarchical realm resolution mechanism of the SchMUSE are still very much in the experimental phase so little can be said of its performance. Nonetheless, we are confident that this style of lazy location with eager cancellation will suit our laboratory and campus environment well. We have also found that performance profiling and system debugging were greatly facilitated by separating an entity's birth machine location from its present location hint associated the object tickets. Specifically, the `btime` and `bmachloc` remain constant despite realm motion, providing a time-invariant UID by which to identify objects in the network.

4 Using Class Information for Efficient Remote Delegation

We turn now to the performance issue of making delegation-based inheritance efficient in the face of remote delegation. In short, we use static information from the class definitions to accelerate method dispatch and slot accesses. Most importantly, the static class information allows us to delegate directly to the node where a slot is located rather than traversing the full delegation chain of the object being manipulated. In the case of highly distributed delegation, this can have major performance improvements since superfluous delegation chain traversal amounts to superfluous network traffic and unnecessary indirection through numerous workstations, tying up valuable resources and heightening the liability of access failure. In the case where the complete structure information about a node is not present on the SchMUSE session that is processing a message, we must be a bit more clever.

In brief, the performance issues surrounding remote delegation can be divided along two lines of concern: first, the amount of structure information available concerning the object being manipulated; and second, the kind of message being processed by the object. These concerns naturally subdivide, in turn, as follows: regarding structure information, we must consider 1) if the *full* class information is co-resident with the instance, or 2) if we allow opaque remote classes, *i.e.,* if we can have a local instance whose local class information may be known but whose inherited class structure is remote and not known. As we shall see shortly, these two class policies have different implications depending on whether the message involved is A) a slot access message, or B) a compound method message.[6] The following subsections address each of the four resulting combinations.

4.1 Case 1A: Full class info; Slot access

Returning to our example delegation in Figure 1, imagine we are on the machine where the student node resides and we wish to access the student's `place` slot. Imagine that each ancestral delegate node resides on a separate machine. In such a configuration, if we were naive in accessing the `place` slot of the student, we would have to traverse the object's delegation DAG through the `Person` node to get to the `MobileObject` node where the `place` slot resides. If, however, we locally knew the object tickets in the full object class ancestry of the student and we further knew that the `place` slot was accessed as slot 0 of the student's parent's primary parent (encoded as delegate `<0,0>`), then we could directly send to the `MobileObject` node a request for the value of slot 0. This would entirely bypass the `Person` machine and would lighten the burden on the `MobileObject` machine by not engaging it to decode the `place` message into a slot access on slot 0.

We implemented such a strategy as follows: at class creation time, we "compile" the slot access methods to operate in terms of "lexical addresses" into the delegation chain. Specifically, at method installation time, we statically compute, for each slot accessor method, both the offset into the delegation chain for the delegate which possesses the target slot and the index of that slot within the target delegate's array of slots. We further arrange that each local node, upon creation, cache away a DAG representing its full delegation ancestry. In this way, whenever a slot access method is called on a local node, the encoded indirection into the delegation chain can be used to indirect into the cached delegation ancestry DAG. The encoded slot access can then be sent directly to the object ticket representing the target delegate. In our prototype implementation, we witnessed an order of magnitude in performance improvement when this enhancement was installed: a test case that took 20 minutes to complete without it took only 2–3 minutes with it.[7]

4.2 Case 1B: Full class info; Compound method

Compound methods can likewise be decomposed into more primitive operations. Specifically, every method ultimately decomposes into nested sequences of Scheme language primitives and slot accesses. Thus, if we can know the implementation of the `AnimateObject`'s `hear` method locally on the machine where the `Student` node inherits it, then `hear` messages sent to the `Student` can be decomposed *on the local `Student` node's machine* into Scheme primitives and slot accesses. This means that the bulk of the work involved in processing a `hear` message can be assumed by the machine on which the message was initiated. This dramatically reduces the load on the remote machine where the `AnimateObject` node resides, enhancing its availability to other remote child nodes. In our prototype implementation, we witnessed, on average, a 4–5 fold decrease in load on the remote node when this feature was enabled.

4.3 Cases 2A & 2B: Remote class info; Slot or Method messages

The natural question to follow from the preceding is, of course, what to do if the full class information (namely, delegate storage and inherited methods) is *not* available to some remote child. For example, imagine that the

[6] That is, a method whose body is a mixture of slot references and Scheme procedure calls.

[7] This test case involved an obscenely complex remote delegation DAG being pounded on during peek network usage hours.

local slot structure and local methods for a `Student` are known at the student's node but that the class information for a `Person` is not known. This may be a reasonable situation if, say, the `Person` class implementor has decided not to export the implementation of persons.[8] In this case, any message that we can decode locally using only information known about local students can still be done with some efficiency, yet any inherited message would be an unknown.

In such a case, we re-package the message and forward it to the machine where the remote class resides, using the `RealmTicket` in the `ClassTicket` of the object's parent to determine its residence.[9] In this re-packaged message we are also careful to include the object ticket of the `Student` object for which the message was originally intended. It is then the responsibility of the remote class to ultimately discover the method for this message. This may, in turn, involve further re-packaging to other remote classes. Nonetheless, assuming an applicable method is discovered, the invocation of this method on the intended object ticket will naturally result in the appropriate decomposition of the message into SCHEME primitives and slot accesses. Thus, the price for keeping a class implementation private is the cost of performing all non-exported methods at the class site. Notice that this suggests that some methods may be exported and some not, depending on their depth within the inheritance structure. Notice, further, that in the extreme case where no classes are exported, the remote class repackaging strategy will ultimately result in a full traversal of the delegation structure. This ultimately degrades into the naive delegation strategy outlined at the beginning of case 1A.

If the unknown re-packaged message is found to correspond to a simple slot access method, it is then desirable to request the intended object ticket to reveal its full ancestry so that the slot access optimization can proceed by using this ancestry DAG to avoid traversing the delegation chain, analogous to case 1A above.[10] This is very helpful in the case where some intermediate ancestor is only partially opaque (like `Person`) but some deeper ancestor atop which it is built is exported (like `MobileObject`).

If, on the other hand, the message is discovered to be a compound method, then, similar to case 1B, the method can be decomposed into more primitive operations and processing proceeds as sketched above. Notice that it is not until we tackle the message at the remote class's host that we discover whether the message is a slot access or a compound method.

4.4 Summary of Efficient Remote Delegation

In closing on this issue of efficient remote delegation in the face of full/remote classes a few points should be emphasized.

First of all, it should be noted that we intend to require every local object to at least have local class information on the machine where it resides. Specifically, we consider it undesirable to have a local `Student` object instance, say, on a machine that does not know about the `Student` class. Such a local object would be useless in that every message to it would ultimately have to be forwarded somewhere else only to have the most primitive slot accesses actually performed on the object locally.

Second, we do not have a problem with several independent implementations of the "same" class (*e.g.,* two distinct kinds of `Student`) since each class is stamped by when/where its implementation was born. Specifically, when a class implementation is exported, the `ClassTicket` for the class at the exporting site is embedded within the exported class code. Thus, distinct re-implementations of a class (*i.e.,* version updates) will have different `ClassTicket`s since their timestamps (at least) will differ. Similarly, exporting a class to a machine that then extends or otherwise modifies the class will again be reflected in a new distinct ticket for the resulting new class. To our knowledge, no other system has adopted such a clean approach. The use of class version numbers comes close, but requires global synchronization on the issuance of these version numbers. Our technique requires no such synchronization. We find the globally unique ticket mechanism to be quite elegant in this regard.

Finally, it is worth re-iterating that at any point in the class hierarchy one might choose to export or not export an ancestral class implementation. Thus, hierarchical privacy and opaque encapsulation are supported in the SCHMUSE. This mechanism, in effect, supports abstract types in a distributed object-oriented framework. The burden, of course, is that the site of the implementation is charged with processing the messages to such objects. We have not explored the ramifications of this policy in detail, but we consider it to be intriguing that to keep an implementation aspect private or proprietary one must pay the overhead of servicing requests for these secretive aspects oneself or arrange to have them serviced only by trusted authorized or licensed servers.

5 Message Deferral for Reliable Remote Delegation

We now move from the efficiency issues surrounding remote delegation to the reliability issues. Specifically, a distributed system is reliable if its performance and avail-

[8] For example, this class may be still under construction or one of its methods may use proprietary code or code under federal export control.

[9] This, of course, is complicated by multiple inheritance. In such a situation, each remote parent class would have to be attempted, in turn, until one of them is able to process the message, after which the deferral target can be cached to avoid repeated exhaustive searches.

[10] The reason this is only "analogous" to case 1A is that a trick is involved. The trick is to notice that in re-packaging to some delegate class, the offset into the delegation chain that that class's methods will embody will not include the delegation through the ancestry of the intended object up to that remote class. This is easily handled by the remote class by first walking through the delegation ancestry of the intended object to get to the delegate ancestor corresponding to the remote class. Once there, it can proceed by applying the slot access method to that object ticket.

ability are not affected by machines crashing. We have already seen how delegating directly to nodes that contain slots allows us to jump over intermediate delegate nodes. This certainly enhances reliability in the case where the jumped node may not have been up.

Otherwise, if some delegate is unavailable in our system due, for example, to the server that supports that delegate not being currently "plugged into the SCHMUSE" then we can just avoid any slot accesses to slots local to that delegate and hence avoid thrashing the Central Services in search of a realm that is not present and wasting time with time-outs on messages that cannot succeed.

5.1 The Problem

Beyond this, however, is a desire for more graceful degradation in behavior. That is, failing to process a message because of the inaccessibility of the node which holds some object slot (or some method, in the remote class case) seems severe. In many cases, it may be that the method on some node is merely a fanciful specialization of some inherited method. For example, the `say` method on `Student` may just print something silly (like "Yo") then proceed to invoke the parent `Person say` method. In such a situation, it might be desirable to permit the student behavior to gracefully degrade into normal person behavior when the student method is unavailable. Thus, were we to build a `GradStudent` class that delegates to `Student` but choose not to export the implementation of the `Student say` method to the grad student,[11] then attempts to make a grad student node say something would be fraught with peril if the parent `Student` node were crashed. Even when a grad student forgets for a moment how to behave like a student, it would be nice if they could at least still temporarily act like a normal person in the meantime rather than going totally catatonic.

Similarly, the node at which some slot resides may become unavailable. If that slot shadows some other slot with the same name but later in the delegation chain, it may sometimes be desirable to permit slot reads of the unavailable slot to be deferred as slot reads of the shadowed slot. Returning to Figure 1, for example, if the `Student nickname` slot becomes unavailable, it may be acceptable to read the shadowed `Person nickname` slot instead.

We have attempted to deal with this desirability of graceful inheritance degradation by a mechanism we call *message deferral*. For example, we would say that when the student class is not available, it may be safe to *defer* the `say` message to the shadowed person class. Some methods may be safe to defer while others may not be. For example, when a `MobileObject` moves, it merely changes its `place` slot and tells the place object to update its inventory to include the new object (and tell the old place to release the object). For `Person`s, however, we must not only move the person object but also move each of the objects being carried in the person's `possessions` slot. Thus, were the person class not available to some student, it would not be safe to defer `move` messages to the underlying `MobileObject` class upon which the person class was built. To do so would mean to lose your wallet along with all your other possessions!

Similarly, we may have an `Employee say` method that is specialized to say "Sir" then go on to call the `Person say` method. We might then build a `Manager` class that delegates to `Employee`. Were we to then ask a `say` message of a manager whose employee node is unavailable, it may be undesirable, due to corporate protocol, to defer to the less formal person `say` method. Similarly, it might be embarrassing to defer to the pedestrian `Person nickname` slot when an attempt is made to access the `Employee nickname` slot. These illustrate a subtle complication in the deferral mechanism: the safety of a deferral is not just a property of the message or slot read being deferred, it is also a property of the path by which the deferral takes place. In our `GradStudent/Student say` the deferral to `Person` was acceptable, but not so for the `Manager/Employee` deferral. Thus, when attempting to defer a message, some record of the deferral path must be passed along with the deferral attempt. Notice also that more than one node along the delegation chain may be unavailable at one time. Thus, an arbitrary depth of deferral may become necessary.

Finally, this deferral mechanism is further complicated by multiple inheritance. Consider, for example, some message having methods along both the primary parent delegation chain and the secondary parent delegation chain. If one chain is unavailable, perhaps the second should be attempted. In such a situation, the safety and propriety of deferral may involve shadowing of methods not via child shadowing but via neighboring parent shadowing. Viewing the child/parent paths as vertically directed and multiple parents of a node as horizontally splayed, we recognize that the child/parent shadowing is a matter of *vertical shadowing* while multiple parents that handle the same method could be said to engage in *horizontal shadowing*. This distinction will become necessary in understanding how we implement safe deferral. In a scenario where an employee is a student intern (and thus inherits from both a normal employee node and a student node), this issue can be critical.

5.2 The Solution

How then *do* we implement this selective deferral mechanism? Presently, each time a new class is created that delegates to a parent class, the parent class is notified of the names of all the local messages that the child class handles. In this way, if the parent class has never before been delegated to by a child of the new child class, the parent class can detect which of its methods are shadowed by the child. This same list of the child class's local messages is then forwarded to the parent class's parents' classes, and so on down the class delegation chain, so that deeply shadowed messages can be annotated. Note that at each point where a shadowing is detected, we store in the shadowed class an association between the shadowed message and the child classes which shadow it. If we choose to declare a particular message to be safe to

[11] For example, like when `GradStudent` behavioral norms are regulated by a different set of guidelines than typical `Student` norms, dude.

defer to some specific shadowed parent, we can specify that in the class definition by stating precisely, for each local method, which parent class(es) that method's message can safely be deferred to. In such cases, the child's message shadowing of that message is not recorded at the parent class.

Note also that this shadow information is propagated aggressively at the earliest possible moment, namely at class creation time. This is because if we tried to implement a lazy on-demand strategy, by the time we actually need to attempt a message deferral it may well be too late to attempt to propagate the shadowing information. Since this shadowing information is our means for judging if a particular deferral is safe, failing to propagate this vital information could be our undoing.

To see how this class shadowing information is used to detect deferral safety, consider the following scenario. An object receives a message and decomposes it into a slot access on some ancestor node. If this ancestor is unavailable, it may be desirable to attempt to defer to some other ancestor with a slot of the same name. We therefore consult the method dispatch table for an ancestor whose class indicates that it can handle the message which failed, then forward the message on to the ancestor, telling it that this is a deferral and telling it which ancestor(s) we have deferred across. If the deferred ancestor is likewise unavailable, we proceed through the ancestry with deeper and deeper deferral attempts. Similarly, if the attempt to discover if an ancestor's class can handle this message involves querying a remote class which is likewise unavailable then we treat that ancestor as unavailable. If we run out of deferral candidates, then we finally give up and return a failure signal. On the other hand, if we do find some ancestor available, then, at the point where we succeed, the receiving ancestor processes the message by first consulting its class's association of shadowed messages and the child classes which shadow them. If the present message is discovered to be shadowed by an object class of one of the ancestors we have just deferred across, then the deferral is unsafe and we send an error to that effect. Otherwise, we proceed with the message as usual.

This strategy correctly detects *vertical shadowing*, as defined above. Unfortunately, it does not handle *horizontal shadowing*. For that, we need additional class information. Specifically, each class, upon creation, queries each of its parent classes to discover the full structure of messages that each ancestral class handles.[12] Now, when a class is created that delegates with multiple inheritance, we merely arrange that the list of child messages that the parents are notified of includes those messages handled by neighboring parents. For example, in our `Person` class, the `AnimateObject` class would be notified of the local messages of `Person` as well as all messages of `MobileObject` (since we have left-to-right precedence in our multiple inheritance). When we then attempt a deferral that forces us to try the secondary parent `AnimateObject` delegation chain, we include in our trace of the deferrals we have already attempted all the ancestors tried along the primary parent delegation chain. In this way, the horizontal shadowing information is given to the delegates that may be asked to process a deferred message, and the path which the deferral followed includes the horizontally neighboring ancestors of the target deferral ancestor.

In closing, note that we have expressed this bookkeeping of vertical and horizontal shadowings in terms of messages sent among classes. We did so because, in general, communication with remote classes may require it. In the case where a local class must communicate information to another local class, the message send is somewhat fanciful. Indeed, with some compiler extensions, block compiling a selection of classes could effect the same result directly in the internal class representations. We have not explored such compiler extensions since we wished to make our SchMUSE implementation based solely on portable MIT Scheme language features. Compiler extensions generally do not port quite so easily.

6 Related Work

As mentioned at the outset, our delegation style object system was modelled after [Adams & Rees 88]. Our advocacy of this style's sharing properties and support of multiple shared views comes from Clovers [Stein & Zdonik 89], although our introduction of classes into the delegation style is not traditional in the sense that it is not prototype-based delegation [Lieberman 86].

Of the distributed object systems we have examined, nearly all have opted to extend a traditional class-based inheritance system to a distributed object system only to afterward comment that a delegation-based object system would have been more desirable, as was anticipated by [Otten & Hagen 90]. Most of them cite the extra flexibility provided by delegation systems as the motivating factor. [Bennett 90] even goes to the extent of identifying remote classes as the primary point of tension, arguing that delegation systems, by normally storing methods along with the local slots inside objects, naturally satisfy this key flexibility concern. Their careful examination of the design options surrounding remote classes was very helpful in justifying to ourselves our constraint that local object nodes must have local class information about the node's class. Unlike them, however, we opted to promote classes to the status of objects themselves, complete with their own kind of class reference (namely, `ClassTickets`). We find our solution to this problem of remote classes, therefore, to be fairly elegant without requiring delegation-based object instances to store their methods directly in their representations. Rather, we use class information to establish a method dispatch table separate from the object slots array.

Our use of *globally unique tickets* was inspired by [Leach *et al.* 82]. It differs from the apparently common use of object "proxies" for remote object references [Decouchant 86]. Proxies are objects that accept messages and forward them to a remote object. We instead make our message send procedure detect if the object

[12]Even with opaque remote classes, the ancestral messages are revealed since this does *not* expose the implementation. It only exposes some small structural property of the implementation.

being queried is local or remote (which is easily determined by inspecting the object ticket's realm ticket) and remotely forward the message if it is remote. It seems to us that the use of proxies is an apology for having an awkward object reference mechanism or for having a system which demands every target of a message be an "object" in some built-in language specific sense. In fact, [McCullough 87] even goes so far as to then provide global UIDs for their proxies, along with a table to map from proxies to UIDs to test object identity. This seems like a further apology for having used proxies at all. Since we were not retro-fitting our remote object references to an existing local object language system, we were not compelled to make such apologies.

Some distributed object systems [Nascimento & Dollimore 92] [Feeley & Levy 92] have adopted version numbers for dealing with object and class references. A similar common approach is to adopt a centralized authority to coordinate the issuance of sequential object ID numbers. Any such strategies must globally synchronize their numbering to some extent to avoid collision. Our globally unique identifiers require no such synchronization. This makes our system scalable. To wit, our strategy could support a SchMUSE server at every single Internet address. Moreover, although we shall permit "pointer snapping" of `RealmTicket`s within object/class tickets when those entities migrate among realms, the remainder of the globally unique ID remains stable, as an immutable birthtime/birthplace of each object. This has greatly enhanced debugging and profiling of our system.

Our *realm resolution* is currently loosely modelled after Internet namespace service. We intend to investigate alternative distributed resolution strategies, such as the "clearinghouse" approach of [Oppen & Dalal 83]. Admittedly, given the small size of our prototype lab network (48 workstations; 2 file servers), we have not been motivated to explore fanciful variations. Now that MIT has completed its installation of Internet drops within campus dorm rooms, the motivation to explore creative options is expected to grow and we anticipate a veritable army of participants avid to explore those options.

Finally, our blurring of the distinction between client and server in the SchMUSE is similar to that found in Flamingo [Anderson 86]. Their system, however, is a cooperative mailer system; ours is an interactive simulation environment, so the types of client/server interactions are characteristically different.

7 Contributions and Conclusions

Our novel use of *globally unique tickets* as object references has proven to be elegant, versatile, low-cost, and scalable without requiring network-wide clock synchronization. Further, our separation of object identification from object location, by means of embedding `RealmTicket`s within object and class tickets, has proven quite useful. It allows course-grained object migration (by realm) without requiring forwarding addressing to disrupt the object database. This was achieved by using a separate realm table to map the realm tickets to actual machine locations. This mapping has been prototyped as a "Brazilian" distributed hierarchical lazy mapping strategy modelled after Internet namespace servers. Again, this will scale at least as well as the Internet has scaled.

We believe that our most significant contribution is our incorporation of class information into a delegation-based inheritance system, yielding what we call a *class-based delegation system*. We know of no other distributed object system that has wed these traditionally rivaled approaches of delegation and class inheritance.

Moreover, we found that class-based delegation facilitates several novel issues arising from remote delegation which have heretofore been unaddressed by the object-oriented community. For example, we are unaware of previous work in accelerating delegation through compile-time class analysis in a setting where classes themselves may be remote (and hence, compile-time opaque). In addition, we have also introduced a novel notion of *message deferral* which permits graceful degradation of object behavior and system availability as remote sites go down while paying heed to the safety and propriety of certain deferral.

8 Future Work

There are many directions for further development of MIT SchMUSE. Roughly, these include object migration, enhanced concurrency, access control, and communication security.

Foremost on our agenda is investigating mechanisms for allowing objects to migrate from one realm to another. We will likely follow the fine work of the Emerald Project [Jul *et al.* 88] [Jul 88], including their forwarding address strategy [Fowler 86] [Fowler 85] (for "snapping" `RealmTicket`s in stale `ObjectTicket`s to point to the new realm where the object has moved) as well as their distributed garbage collection algorithm [Vestal 87] (to GC stale forwarding addresses). Of course, distributed GC is valuable for automatic storage reclamation in general anyway so that space quotas with manual object de-allocation do not have to be imposed on users. Although the above citations give the impression that each of these extensions is a solved problem, actually implementing them should prove a respectable engineering task. More recent advances in distributed GC should prove helpful [Detlefs 90] [VAT 92] [Maheshwari 93].

Also, we would like to support a higher degree of concurrency in the SchMUSE network by providing a true transaction-based mechanism for our message passing. (Currently, our messages are not even atomic in that partially completed messages which fail do not back out). We will likely pursue a design based on nested transactions [Moss 81], possibly with some degree of optimistic concurrency among concurrent transactions. This should prove a fairly challenging task.

Next, the present prototype of the SchMUSE provides a fairly exotic locking strategy for object access control. This is documented in [Cohen 93]. At present, however, it provides only fine- and coarse-grained locking (namely, instance locking per message and instance locking per instance creator). We would like to extend it to provide medium-grained locking as well (that is, locking per instance). This is a straightforward extension.

Our initial implementation uses [Adams & Rees 88] style method delegation. We intend to look into a more DYLAN-like generic function method specialization mechanism for overriding default class methods.

Finally, it would also be interesting to support network privacy by encrypting messages and their results as they are shipped between SCHMUSE sites. This should prove a fairly painless extension.

Credit and Acknowledgments

Prof. Hal Abelson initially proposed exploring the idea of a SCHEME-based MOO-like language modelled after Xerox PARC Lambda-MOO but with distributed delegation. Michael Blair and Natalya Cohen drafted an initial design built atop Brian Zuzga's TCP/IP-based message-passing communication substrate. David LaMacchia later joined the design team and assisted in the realm resolution design. All members of the SCHMUSE team assisted in our on-going prototype implementation.

Several members of our Summer '92 MIT SCHEME Team assisted in alpha-testing the design and implementation and, consequently, provided very valued feedback. They include Joseph Boerges, Greg MacLarin, and Prof. Eric Grimson.

We would like to thank Abelson and Grimson for their encouragement and support in pursuing this project and in writing this report. Their guidance (and funding) has proved invaluable.

And finally, we thank the conference referees for their helpful comments and constructive feedback.

References

[Abelson & Sussman 85]
Harold Abelson and Gerald Jay Sussman with Julie Sussman
Structure and Interpretation of Computer Programs
The MIT Press, Cambridge, MA, 1985

[Adams & Rees 88]
Norman Adams and Jonathan Rees
Object-oriented Programming in SCHEME
In Proceedings ACM LISP and Functional Programming, Jul 88, pp.277–288.

[Agha 86]
Gul Agha
ACTORS*: a model of concurrent computation in distributed systems*
MIT Press, Cambridge, MA, 1986

[Anderson 86]
David B. Anderson
Experience with FLAMINGO*: A Distributed, Object-Oriented User Interface System*
In Proceedings ACM OOPSLA '86, pp.177–185.

[Apple 92]
Apple Computer, Eastern Research and Technology
DYLAN*: An Object-Oriented Dynamic Language*
Apple Computer, Inc, 1992

[Bennett 90]
John K. Bennett
Experience With Distributed SMALLTALK
Software–Practice and Experience, Vol.20 No.2, Feb 1990, pp.157–180.

[Blair & Maessen 93]
Michael R. Blair and Jan-Willem Maessen
CLOS CLOVERS*: Delegation-based Inheritance in* COMMON LISP *via the* CLOS *Metaobject Protocol*
Unpublished project. Implementation available at:
`ftp://swissnet.ai.mit.edu/pub/clovers`

[Bobrow *et al.* 90]
Daniel G. Bobrow, Linda G. DeMichel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon
COMMON LISP *Object System*
In [Steele 90], Chapter 28, pp.770–864.

[Clinger & Rees 91]
William Clinger and Jonathan Rees (Editors)
Revised[4] Report on the Algorithmic Language SCHEME
MIT/AI/Memo 848b, November 1991, and/or
In ACM LISP Pointers, Vol. IV, No.3, July-Sep 1991, pp.1-55.

[Cohen 93]
Natalya Cohen
SCHMUSE *001: An Introductory Guide to* SCHMUSE *001*
MIT Artificial Intelligence Laboratory, Project MAC Technical Memo.
Available as `schmuse-manual.{ps,dvi}` at
`ftp://swissnet.ai.mit.edu/pub/schmuse/`

[Decouchant 86]
Dominique Decouchant
Design of a Distributed Object Manager for the SMALLTALK-80 *System*
In Proceedings ACM OOPSLA '86, pp.444–452.

[Detlefs 90]
David L. Detlefs
Concurrent, atomic garbage collection
PhD dissertation. Carnegie-Mellon University, Technical Report CMU-CS-90-177, Oct. 1990

[DOS SCHEME]
MIT SCHEME Team
DOS SCHEME *Implementation*
Implementation available via e-mail request to
`info-cscheme-dos-request@altdorf.ai.mit.edu`.

[Feeley & Levy 92]
Michael J. Feeley and Henry M. Levy
Distributed Shared Memory with Versioned Objects
In Proceedings ACM OOPSLA '92, pp.247–262.

[Fowler 85]
Robert J. Fowler
Decentralized Object Finding Using Forwarding Addresses
PhD dissertation. University of Washington, Department of Computer Science, Technical Report TR 85-12-1, Dec 85

[Fowler 86]
Robert Joseph Fowler
The Complexity of Using Forwarding Addresses for Decentralized Object Finding
In Proceedings ACM Principles in Distributed Computing '86

[Hanson 91]
Chris Hanson
MIT SCHEME *Reference Manual*
MIT Artificial Intelligence Laboratory,
MIT AI TR-1281, Nov 1991

[Hutchinson 87]
Norman C. Hutchinson
EMERALD*: An Object-Based Language for Distributed Programming*
PhD dissertation. University of Washington, Department of Computer Science, Technical Report TR 87-01-01, Jan 87

[IEEE 91]
IEEE SCHEME Standardization Committee
IEEE Standard for the SCHEME *Programming Language*
IEEE Std 1178-1990, May 1991

[Johnson & Zweig 91]
Ralph E. Johnson and Jonathan M. Zweig
Delegation in C++
Journal of Object-Oriented Programming, Nov/Dec 1991, pp.31–34.

[Jul 88]
Eric Jul
Object Mobility in a Distributed Object-Oriented System
PhD dissertation. University of Washington, Department of Computer Science, Technical Report TR 88-12-06, Dec 88

[Jul *et al.* 88]
Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black
Fine-Grained Mobility in the EMERALD *System*
ACM Transactions on Computer Systems, Vol.6, No.1, Feb 1988, pp.109–133.

[Kiczales *et al.* 91]
Gregor Kiczales, Jim des Rivieres, and Daniel G. Bobrow
The Art of the Metaobject Protocol
MIT Press, Cambridge, MA, 1991

[LaLonde 86]
Wilf R. LaLonde, Dave A. Thomas and John R. Pugh
An Exemplar Based SMALLTALK
In Proceedings ACM OOPSLA '86, pp.322–330.

[Leach *et al.* 82]
Paul J. Leach, Bernard L. Stumpf, James A. Hamilton, and Paul H. Levine
UIDs as Internal Names in a Distributed File System
In Proceedings ACM Symposium on Principles in Distributed Computing, Aug 1982, pp.34–41.

[Lieberman 86]
Henry Lieberman
Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems
In Proceedings ACM OOPSLA '86, pp.214–223.

[Leighton 92]
F. Thomson Leighton
Introduction to parallel algorithms and architectures
Morgan Kaufmann, San Mateo, CA, 1992

[Maheshwari 93]
Umesh Maheshwari
Distributed garbage collection in a client-server, transactional, persistent object system
M.S. dissertation, MIT Lab. for Computer Science, MIT/LCS/TR-574, Oct. 1993

[McCullough 87]
Paul L. McCullough
Transparent Forwarding: First Steps
In Proceedings ACM OOPSLA '87, pp.331–341.

[Moss 81]
J. Eliot B. Moss
Nested Transactions: An Approach to Reliable Distributed Computing
PhD dissertation. MIT Lab. for Computer Science, MIT/LCS/TR-260, Apr 81

[Nascimento & Dollimore 92]
Claudio Nascimento and Jean Dollimore
Behavior maintenance of migrating objects in a distributed object-oriented environment
Journal of Object-Oriented Programming, Sept 1992, pp.25–32.

[Oppen & Dalal 83]
Derek C. Oppen and Yogen K. Dalal
The CLEARINGHOUSE*: A Decentralized Agent for Locating Named Objects in a Distributed Environment*
ACM Transactions on Office Information Systems, Vol.1, No.3, July 1983, pp.230–253.

[Otten & Hagen 90]
D. B. M. Otten and P. J. W. ten Hagen
On the Role of Delegation and Inheritance in Object-Oriented Database Systems
Technical Report CS-R9032
Centre for Mathematics and Computer Science, P.O.Box 4079,
1009 AB Amsterdam, The Netherlands

[Snyder 86]
Alan Snyder
Encapsulation and Inheritance in Object-Oriented Programming Languages
In Proceedings ACM OOPSLA '86, pp.38–45.

[Steele 90]
Guy L. Steele Jr.
COMMON LISP*: The Language* (Second Edition)
Digital Press, 1990

[Stein 87]
Lynn Andrea Stein
Delegation Is Inheritance
In Proceedings ACM OOPSLA '87, pp.138–146.

[Stein & Zdonik 89]
Lynn Andrea Stein and Stanley B. Zdonik
CLOVERS*: The Dynamic Behavior of Types and Instances*
Brown University Technical Report No. CS-89-42,

[Stroustrup 86]
Bjarne Stroustrup
The C++ Programming Language
Addison-Wesley, 1986

[VAT 92]
Nalini Venkatasubramanian, Gul Agha and Carolyn Talcott
Hierarchical garbage collection in scalable distributed systems
University of Illinois, Urbana-Champaign.
ILLINOIS UIUCDCS-R-92-1740, Apr. 1992

[Vestal 87]
Stephen C. Vestal
Garbage Collection: An Exercise in Distributed, Fault-Tolerant Programming
PhD dissertation. University of Washington, Department of Computer Science, Technical Report TR 87-01-03, Jan 87

[XLRG 81]
The Xerox Learning Research Group
The SMALLTALK-80 *System*
BYTE, Vol.6 No.8, Aug 1981, pp.36–48.