Implementing Location Independent Invocation

ANDREW P. BLACK AND YESHAYAHU ARTSY, MEMBER, IEEE

Abstract—**Location independent invocation is a mechanism that allows operations to be invoked on application-level objects, even though those objects may move from node to node in a distributed system. It is independent of any particular application, operating system, or programming language, and is applicable to many applications that are constructed from collections of named entities, provided that each of these entities is itself contained within a single address space. Our implementation is layered on top of an existing RPC system, and is designed to operate in a network that links thousands or tens of thousands of nodes in the wide area.**

This paper sketches our work on building a highly distributed office application based on mobile objects, elaborates on the technique used to find the target of an invocation, and describes how our technique is implemented. We also discuss our experiences building a prototype system and initial application, and include some early performance data.

Index Terms—**Distributed applications, forwarding address, location independence, object finding in large-scale systems, object mobility, remote invocation, remote procedure call (RPC).**

I. INTRODUCTION

THE motivation behind remote procedure call (RPC) [20] is to make distributed applications easier to construct. Procedure call is a well-known and well-understood mechanism for the transfer of control and data within a single address space. Remote procedure call represents an extension of this mechanism to provide for transfers *between* address spaces. It is generally a goal of an RPC implementation to make the semantics of a remote call as close as possible to those of an equivalent local call [5]. This enables programmers to write distributed applications without having to be aware of network protocols or external data representations. Thus, anything that narrows the conceptual distance between local and remote calls is considered to be desirable.

Nevertheless, there remain some intrinsic differences between the semantics of remote and local calls. For example, since a remote procedure executes in a different address space from the caller, passing parameters by reference is difficult, and the caller and callee may fail independently.

Of these intrinsic differences, the most fundamental is that a remote call must somehow indicate to *which* other address space it is directed. The caller discharges this responsibility by *binding* to the appropriate address space before the first call can proceed. The action of binding results in a data structure (also called a binding) that identifies the callee and must be passed as an implicit or explicit argument to every remote procedure call.

Manuscript received April 21, 1989; revised August 25, 1989. A preliminary version of this paper was presented at the 9th International Conference on Distributed Computing Systems.

The authors are with the Distributed Systems Advanced Development Group of Digital Equipment Corporation, Littleton, MA 01460.

IEEE Log Number 8931944.

Choosing the correct server can be difficult. Various techniques have evolved to lift this burden from the programmer of the calling code—in other words, to automate to some extent the choice of the callee. Two commonly used methods are default binding and clerks.

- *Default binding* (sometimes called automatic binding) occurs when the RPC system chooses the server without an explicit request from the client. The choice may be made nondeterministically, or in a way that depends on factors that are normally hidden from the caller (such as network distance). Default binding may be appropriate if only a single server of the requested type is available, or if all of the available servers are semantically equivalent. However, these are necessary but not sufficient conditions: if, for example, two semantically equivalent servers have widely differing costs, the user is unlikely to be satisfied with a default selection.
- *Clerks* are application-dependent subroutine packages that use information about the call semantics to choose the callee. For example, in an environment in which a file name indirectly identifies the file server that stores a file, a *file system clerk* might be used to direct an *open* call to the appropriate server. The clerk interface is pleasant to use, and directly supports the abstraction in which the client is interested, namely, a file service. Unfortunately, it is specific to filing; a new clerk must be written for each new application.

Location independent invocation (LII) is a general way of eliminating the binding step; it completely removes RPC bindings from the view of the application programmer. This technique is applicable to a wide variety of applications; it requires only that the application operate on a set of named entities, and that the whole of each entity be located in a single address space.

LII should be thought of as a service at a level of abstraction above RPC. Our particular implementation is layered on RPC, but this is by no means essential. LII has previously been implemented at the operating system level (the Eden system [1], [6] incorporates a similar concept) and as part of a programming language (Emerald [15]). LII can also be implemented at the application level; the file system clerk mentioned above may be regarded as an example. The contribution of this paper is to present LII as a conceptual service that is independent of any particular application, operating system, or programming language, and to show how it can be implemented without language or system support in any environment that provides reliable interprocess communication. The particular implementation described here uses several techniques to efficiently and reliably find mobile objects in a widely-distributed system. As in Emerald, we

normally use temporal forwarding addresses that are updated cheaply as a side-effect of object migration and of invocation. However, our method is unique in its use of stable storage and a global name service to ensure that active objects are highly available despite crashes and without resorting to exhaustive search.

The remainder of this paper is organized as follows. In Section II we study the "indications" for object mobility and location independent invocation, i.e., the circumstances under which the abstraction that we are proposing is beneficial. Section III describes the context of our work: an application domain in which we believe LII will be useful and the core services that support it. In Section IV we describe our object-finding algorithm, and in Section V we present our implementation. Section VI deals with the relationship of LII to earlier work on object finding and location independence. Section VII summarizes our experience designing and implementing the system and a sample application; it includes performance measurements of our prototype system.

II. When is LII Useful?

Before examining the situations in which LII is useful, it is necessary to introduce some terminology. An address space that is willing to accept RPC calls is conventionally known as a server, and one that makes calls is known as a client. Note that these terms are meaningful only with respect to a particular call; a given address space may function as both a client and a server simultaneously. The set of calls that are acceptable to a server characterizes its interface, and is therefore known as the service type. A particular server that implements that type is known as a service instance. In conventional RPC systems such as Birrell and Nelson's [5], the service type is chosen when a program is written, whereas the server instance is chosen by the run-time bind operation.

If the RPC interface provides a computational service, e.g., fast Fourier transform, all the service instances may be semantically equivalent. In this case the necessity of choosing a service instance may be an opportunity rather than a problem: the binding process may provide a way of sharing load amongst the various servers, or of selecting a server that provides the required precision.

However, if the RPC interface provides access to application data, then the services available from different servers may be distinct, even though they share the same service type. Consider, for example, a service type that defines the interface to a directory of a company's employees. Different instances of this service might provide information about the employees of Digital, General Motors, or IBM: clearly, it is important to select the correct service instance. Having done so, one might then find that there are several *server* instances that implement the whole of the Digital employee directory; the choice amongst them may influence performance, but not correctness. In general, the application domain (directory services) will be concerned with some number of application dependent objects (specific directories of employees); there will be a mapping from application entity to service instance that enables a client to choose the correct instance.

In the case that this mapping is constant, or changes very occasionally, it is easy to arrange that every client of the interface has a copy of a function that computes the map. A trivial implementation of such a function would encode the location of the application object in its name.

Once the mapping changes more than very occasionally, it becomes difficult to implement the function that computes it. A distributed global name service such as the DNA Naming Service [11] can be used; such a service trades off consistency in favor of availability. Updates can take a significant amount of time to propagate through the name service, and during this time the function implementing the mapping is wrong. The severity of this problem increases with the frequency of updates.

When the mapping changes frequently in a large system, computing it becomes a distributed task, as well as a significant burden to the application programmer. Location independent invocation as described in this paper relieves the programmer of this task by automating it. LII is appropriate when the application entities move frequently enough to make the implementation of a static mapping function impractical.

Why should the application objects move? Although a comprehensive evaluation of the costs and benefits of object mobility is beyond the scope of this paper, we would like to characterize in the rest of this section the circumstances under which it is useful. A fuller discussion of the advantages of object mobility can be found in [15].

The advantages of mobile objects include those of process migration: the ability to share load between processors, reduced communication cost, increased availability, reconfigurability, and the possibility of utilizing the special capabilities of a particular machine. In addition, mobile objects provide a convenient way for application data to be moved to the point where it is needed without the complexity of making a copy and keeping it consistent with the original despite failures and possible network partitions.

This is not to say that mobile objects solve every problem. In particular, if a single resource is used concurrently in widely separated places, making it mobile will not help to improve locality, and may instead cause thrashing. Replication may offer an appropriate solution in this case; our restriction (mentioned in the introduction) that the whole of an invoked object be located in one address space means that a distributed replicated entity cannot be treated as a single object for the purposes of LII. It is possible to implement a replicated resource by using *multiple* objects and various consistency protocols, but the caller must then be aware of this extra level of indirection [21], [23].

Nevertheless, we believe that there is a class of applications for which mobile objects are a suitable technology. This class includes applications in which clients at various sites interact intensively with some objects, and in which sequential sharing between clients is much more common than concurrent sharing. These characteristics are typical of applications in which real-world objects move; in addition to office applications (itself a broad field) we mention medical data, which often moves with the patient, and manufacturing information, which is frequently recorded on a docket that moves with the lot of goods. (See [26] for a proposal to use mobile objects in

manufacturing.) In addition, our system provides support for applications that allow incremental growth and that scale to large numbers of clients and large geographic areas.

Moving an object is typically more expensive than accessing it remotely since the object migration mechanism must ensure that it is possible to find the object at its new home, and to recover it safely in the event of a failure. Nevertheless, the *quality* of a local interaction is often sufficiently better than that of a remote one to justify this cost. Consider a physician examining a digitized X-ray photograph; very high bandwidth communication between the object holding the image and the physician's display will be required, and co-locating these two entities may be the only feasible way of providing it. Moreover, if it is possible to predict that the physician will wish to examine the X ray, the appropriate object can be moved to his office before the first attempt is made to access it. (This might be achieved, for example, by scanning the physician's appointments' calendar a day in advance.) The time taken to perform a migration may then be almost immaterial.

III. The Hermes System

To investigate the use of mobile objects we chose to automate a corporation-wide, real-life office application. The application deals with expense voucher forms circulating within Digital Equipment Corporation. A form is filled-in by an employee and signed by different managers, who approve or reject the expense reimbursement represented by the form. If appropriate, a petty cash disbursement is made, and then the form is archived for tax and audit purposes. The people acting on the form may all be located in one building, spread across the country, or situated in different continents. As in the broad class of applications discussed earlier, the interval between update sessions may vary from minutes to days, but the interactive response is important to the person acting on the form. Other important functions of the system include translation of organizational titles to employee names, and authentication; these topics are beyond the scope of this paper. In this section we detail those characteristics of the system that are relevant to the discussion of finding objects and making invocations location independent.

Our application must be able to span thousands of machines: Digital's internal network is world-wide and currently interconnects over 36 000 nodes. Although the processing of most forms will be geographically localized, some may require the attention of two people on opposite sides of the globe. These requirements make it infeasible to store all the forms in a single centralized database, or even in a number of geographically dispersed databases. Instead, we represent the data and code of each form as an object that can move around the network as the application demands.

Objects are named entities that perform operations on each other by invoking well-defined interfaces. An object consists of some state and a set of operations that manipulate it; the operations are invoked by one or more threads of control. Using system services, an object can control its own persistence, recovery and placement, and can invoke operations on remote objects without concern for their location. Likewise,

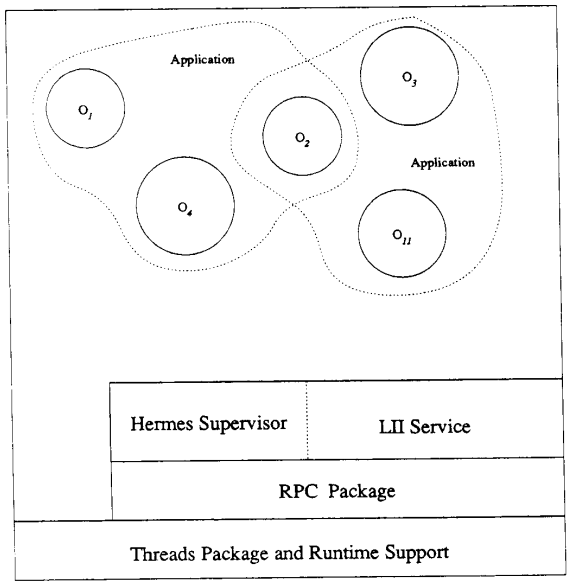


Fig. 1. A Hermes Nodes—Internal view.

the application programmer, although in control of the placement of objects, need not be concerned with the mechanisms used to move objects and to handle remote invocations.

Thus, the view of the world presented to application programmers is a distributed ocean in which application-dependent objects of their own design can be floated. The Hermes system provides the infrastructure necessary to implement such objects; it is composed of Hermes *nodes,* which hold active objects in virtual memory, and of stable backing storage that holds objects when they are not currently active or when a node crashes.

Each Hermes node consists of a number of components (see Fig. 1): a Hermes supervisor, a collection of application objects, and an inter-object communication mechanism. The supervisor provides services for object creation, finding, and relocation: the inter-object communication mechanism consists of the LII layer and the underlying RPC system. An application is implemented by a collection of cooperating objects. All of the components share a threads package that provides for concurrency and cooperation.

In current prototype, each Hermes node is equipped with all the code and structures necessary to support all the different types of objects defined by all the applications in the Hermes system. Any instance of an object of any type can migrate to, or be created at, any Hermes node. In a production system this is an unreasonable assumption, and it will be necessary either to provide for the dynamic loading of object code, or to restrict the location of objects to those nodes that already have the appropriate code.

The storage service that is used to make objects persistent is implemented by a distributed collection of *storesites.* We assume that each storesite has local access to stable storage, and that there are fewer storesites than Hermes nodes. An

TABLE I
OBJECT FINDING ALGORITHM

Invoking node

1. If the object (o) is local, invoke the requested operation (p) on it and **return** whatever it returns.
2. A *tad* (t) for o must exist.
   Perform RPC to *t.node* (*Remote node* below) to request p, passing along p's arguments and t.
3. If the RPC returns, it indicates the last *tad* (t') used to find o.
   Replace t by t', if the latter is newer, and
   3.1 If o was found, **return** to caller whatever result or exception the remote p has returned.
   3.2 If o could not be reached (because its node or storesite were not available), **return** *Unavailable*
   3.3 If the search for o involved too many nodes (*HopCountExceeded*), **goto** step 2 to start a new chain.
4. If the RPC fails, call *Get Tad From Storesite* (below), and
   4.1 If a new *tad* is obtained, **goto** step 2.
   4.2 Otherwise, **return** *Unavailable*.

Remote node

1. If the object (o) is local, invoke the requested operation (p) on it and **return** whatever it returns *and* my *tad*.
2. If caller's *tad* is newer than mine, call *Get Tad From Storesite* (below), else **goto** step 3.
   2.1 If a newer *tad* is obtained, replace my *tad* by it and **goto** step 3.
   2.2 Otherwise, **return** *Unavailable* and caller's *tad*.
3. If the call has been propagated already to too many nodes, **return** *HopCountExceeded* and my *tad*.
4. Perform RPC to the node indicated in my *tad*, passing my *tad* and p's arguments.
5. If the RPC returns, it will include a *tad*.
   If the *tad* is newer, use this *tad* to replace mine.
   In any case, **return** whatever results the RPC returned.
6. If the RPC fails, call *Get Tad From Storesite* (below), and
   6.1 If a newer *tad* (t') is obtained, replace my *tad* by t', and **goto** step 4.
   6.2 Otherwise, **return** *Unavailable* and my *tad*.

Get Tad From Storesite

1. Call the storesite (s) that I believe is currently supporting o.
2. If the call returns, then if it contains
   2.1 a *tad*: **return** it.
   2.2 a reply "Another storesite (s') is currently supporting o": **goto** step 1, calling s' instead.
   2.3 a reply "I know nothing about o": call the name service to learn o's current storesite and **goto** step 3.1.
3. If the call fails, call the name service to learn o's current storesite, and
   3.1 If the call returns, then
      3.1.1 If a newer storesite (s'') is returned, **goto** step 1 (calling s'' instead).
      3.1.2 otherwise, **return** failure.
   3.2 otherwise, **return** failure.

object can write its entire state to a storesite periodically (a checkpoint), and can log changes to its state between checkpoints; the storesite is then said to *support* that objects. After a crash, an object can be recovered up to the last logged change. To prevent the inadvertent "recovery" of objects that are still functioning but on inaccessible nodes, the storesite also records the current location of every object that it supports. It will not permit an object to be recovered until it is sure that the object in question has really crashed. The LII mechanism also uses the location information held in stable storage to find objects in certain circumstances; this is discussed in Section IV.

The Hermes system assumes the existence of two generic distributed services: a name service and an authentication service. The former is needed for a Hermes supervisor to find other supervisors and storesites, and to map the external names of certain well-known objects to their object identifiers. As will be seen later, the name service is also used occasionally to help find an object that has moved. The authentication service is needed for Hermes supervisors to authenticate each other, as well as for objects to authenticate each other.

Hermes nodes are implemented as ordinary processes distributed across the network. The components of a Hermes node all share the same address space, making local invocation simple and fast. Users interact with Hermes using a window-based interface; each user has his or her own interface process, which communicates with some convenient Hermes node using RPC. Hermes uses ordinary operating system services and the RPC system developed at Digital's Systems Research Center. We chose to implement Hermes in Modula-2+ [24] because it provides threads and because the RPC system is integrated with the multithreading facilities.

IV. Finding Objects in Hermes

This section commences with an overview of the finding algorithm; a more detailed description of the finding process appears in the following three subsections. Table I summa-

rizes the algorithms of the parties that participate in object finding.

Overview

Every object is assigned at creation and keeps forever a globally unique identifier (*guid*). Each object also has a current location (the Hermes node that contains the volatile copy of the object) and a current storesite (which contains the object's log and checkpoint records); both of these attributes may change. In addition, each object has an *age,* which is initially zero and is incremented whenever the object attempts to move to another node. If object o is at node n after its a^{th} move, then the pair $\langle n, a \rangle$ is a *temporal address descriptor* (*tad*) for o. A *tad* is a true statement about an object's location at some time in its history. Given two tads $\langle n_1, a_1 \rangle$ and $\langle n_2, a_2 \rangle$, it is easy to determine which is newer, and thus likely to be better: $\langle n_2, a_2 \rangle$ is newer if and only if $a_2 > a_1$. Inside a Hermes node, references to objects are essentially *guids.* Whenever a reference is passed from one Hermes node to another, or written to stable storage, the newest available *tad* is passed in addition to the *guid.* Each Hermes supervisor contains a stash of *tads;* there is a *tad* for every object local to the supervisor's node, a *tad* for every object reference contained in a local object, and, in addition, some collection of *tads* for other objects that are not currently referenced, but which have been referenced in the past and might be referenced in the future.

Our location scheme employs a succession of increasingly expensive but increasingly available techniques to find an object. Invocation and location are combined, both to save messages and to prevent an object from moving after it has been found but before it has been invoked. First, an attempt is made to invoke the operation on the object locally; this will fail if the object is not local. In this case the operation is attempted again, but this time remotely at the node named in the *tad.* If the *tad* is up to date, the invocation will succeed at this point; if it is not, the remote node will normally have a better *tad,* and will either return this *tad* to the caller, or forward the invocation to the node that it names. This process is repeated until is succeeds or there is no newer *tad.* In effect, we are using the information in the *tads* as forwarding addresses; this is inexpensive, since a newer *tad* is normally obtained as a side effect of a failed invocation. Moreover, when an invocation succeeds, it is cheap to pass the current *tad* back to the caller with the results (or exceptions), so that subsequent invocations can be directed to the correct place immediately.

If we run out of good forwarding addresses the algorithm resorts to finding the objects in stable storage, consulting the storesite that currently supports that object. The storesite keeps current information as to which Hermes node the object occupies. If necessary, the global name service is consulted to ascertain the appropriate storesite for the sought object. One might ask: why not use the name service to store the object's current location in the first place? This would certainly simplify the finding algorithm. However, the high availability of a name server is obtained at the cost of consistency: the information it provides is possibly out of date. Thus it would still be necessary to follow forwarding addresses, and to determine which of two addresses is the newer. Moreover, the information obtained from a *tad* when an invocation fails is essentially free, whereas a name server enquiry must cost at least a network round trip, and usually more. Furthermore, the cost of updates in the name service is presumed to be high since name services are usually engineered for a high ratio of enquiries to updates. We except to have to tune our finding algorithm depending on the actual costs of lookups and updates in the name server and at the storesite.

Using Temporal Address Descriptors

When an object o is created at a Hermes node n, the supervisor creates a descriptor that includes the object's fresh *guid* and the *tad* $\langle n, 0 \rangle$, as well as other essential information. This *tad* is obviously current. If o later migrates to m, its descriptor on n is updated to include the new *tad* $\langle m, 1 \rangle$. Assuming that objects do not migrate too often, this *tad* represents a forwarding address that is likely to remain correct for some time.

When another object q in n wants to invoke an operation on o, the LII mechanism uses the information in o's descriptor to find o and perform the operation. If o is still local, the invocation becomes a local procedure call. Otherwise, a binding to the appropriate interface in m is acquired, and the RPC system is used to make the invocation.

If o is no longer at m, then m will normally have a newer *tad* for o. What should m do? There are two alternatives: 1) return a failure indication along with its stashed *tad,* or 2) propagate the invocation to the location indicated in its *tad.* The advantage of the first approach is its simplicity and efficiency: an invocation directed to a node either succeeds or returns immediately, without tying-up a thread of control until the chain of forwarding addresses completes. Of course, in this approach the invoker has to try each of the forwarding addresses in turn. The second approach not only ties up a thread, but is more susceptible to disruption by crashes. However, it has the advantage that by delaying the return until the invocation has completed (successfully or otherwise), node m can obtain the new *tad* for o.

In Hermes we chose a hybrid of the two approaches. If the invoked node does not have the object, the invocation is propagated to the location indicated in the *tad;* this continues up to some small number of hops. This number is a tunable parameter; setting it to zero reduced the hybrid approach to the first one. Based on Fowler's work on object finding [13], [14], which analyzes the use of forwarding addresses under a wide variety of circumstances, we believe that chains of forwarding addresses will usually be short. Hence, the location algorithm will frequently succeed at this stage, with the side effect of updating all the *tads* for o along the forwarding chain. Otherwise, a failure with a newer *tad* is returned to the originator of the search, n in this example, which starts afresh with the new forwarding pointer.

A *tad* is also obtained when an object is used in an invocation. Whenever an object reference is passed as an argument to or as a result of a remote invocation, or even as a field buried in a structured argument or result, the LII mechanism piggybacks the newest *tad* that is locally available

onto the object's *guid*. This is done automatically by a customized RPC marshaling routine.

It is obvious that a *tad* received in either way might not be current when it arrives, even if it were current when it was sent: the object might migrate while a *tad* is traversing the network. If a Hermes supervisor that receives a *tad* for object *o* has no other *tad* for *o*, it might as well add that *tad* to its stash. Otherwise, it should keep the newer of the *tad* it holds and the *tad* just received.

Since a *tad* for an object becomes outdated only when that object moves, it is appropriate to measure the currency of a *tad* by its age field. It would be possible to measure *tad* currency with real time stamps, but this would require synchronized clocks. Notice that *tads* are not hints: every *tad* is a true statement about the location of the corresponding object at some time in the past or present. Consequently, a Hermes supervisor can safely replace an old *tad* by a newer one. The result of this method is that, ignoring crashes, a Hermes supervisor's location information "is getting better, all the time" [19].

Provided that no node is unavailable and that no information is lost in a crash, every object in the system can be found by following forwarding pointers. However, real systems are subject to both of these problems. One disadvantage of the forwarding address technique is that not only must the object itself be available, but so must all the nodes along the forwarding chain. Thus, although long chains reduce performance only linearly, they reduce availability exponentially. Although our experiments to date confirm that chains are usually short, when a forwarding pointer is not available we must have an alternative strategy.

Using Storesites to Find Objects

When the object-finding algorithm reaches a point where there are no more *tads* to chase (i.e., no *tad* was returned with a failed invocation, the *tad* returned is older than one that has already failed, or the node named in a *tad* cannot be contacted at all), it has to revert to another method. Using a broadcast or a multicast to locate the object or obtain a newer forwarding address, although justifiable in a small research network such as that used for Emerald [15], is inappropriate in a global network with a complex topology and tens of thousands of nodes. Moreover, when using an unreliable multicast, the absence of a response provides no information.

Instead we use the location information in the stable storage service to help find the object. As mentioned previously, when an object *o* is created it is assigned to a storesite. The location of *o* along with its initially checkpointed state is recorded in stable storage at that storesite. When *o* moves to another Hermes node, its new location is recorded at the storesite using a two-phase commit protocol. Thus, if one can find the storesite that supports a given object, then one can find the object in volatile memory.

Notice that although the *tad* at the storesite is current, by the time it reaches the requesting Hermes node the object may have moved, and it may therefore be necessary to follow a forwarding addresses. Nevertheless, this process will terminate so long as invocation is faster than migration.

Crashes impact the finding algorithm because they cause a Hermes node to lose its entire stash of location information. Naturally, the node will also lose all its local objects, which will need to be recovered from stable storage. They may be recovered onto another node, or onto the crashed node after it has rebooted. Recall that whenever an object checkpoints or writes a log record to stable storage, every object reference is written as a *guid* and a *tad*. Consequently, when the crashed objects are recovered, their new supervisor is provided with a *tad* for every object that they reference (which it will discard if it already has a newer *tad*).

Rather than letting a Hermes node's stash of *tads* be lost in a crash, the supervisor might write the *tads* to stable storage at intervals, or even after every update to its stash. A crash would then result in little or no loss of location information. However, writing frequent updates to stable storage is expensive, and may be of little benefit: if the node stays down for any length of time many of the *tads* will be of purely historic interest. In our current prototype, a Hermes node does not save its stash of *tads* on stable storage; after a crash it may obtain an old *tad* from a peer, and will update it only when the corresponding object is invoked. It would be interesting to evaluate the relative costs of saving the *tads* and of this approach.

If a node *n* does lose its *tad* in a crash, or even if it recovers historic *tads* from stable storage, after *n* has come back up the first invocations for an object *o* directed to *n* will fail without returning a newer *tad* to the invoker. Then the storesite supporting *o* will be found (see below), and queried, and will return *o*'s current location. However, it may be that *o* was itself at *n* at the time of the crash, and that the storesite has not learned that *n* has crashed; it will therefore still believe that *o* is located at *n*. In this case, *o* needs to be recovered and reactivated. The fact that *n* no longer hosts *o* is itself sufficient to distinguish this case from the case in which *o* migrated to *n* after *n* had recovered.

Notice that the existence of stable storage is not by itself sufficient to guarantee that invocations always succeed in the presence of failures. Consider the case where a machine in the chain of forwarding addresses does not respond to an RPC. If we can be sure that the invocation did not take place, then we can treat the lack of response as if it were a failure message, and try to get a newer *tad* from the storesite as described above. However, if it were possible that the invocation completed and that the node crashed just before sending the reply, then locating the object through stable storage and reactivating it might cause the invocation to be performed twice. To prevent this we need another mechanism, such as an invocation sequencer, and a higher-level protocol to decide whether to repeat the invocation once the object is found. Alternatively, the application may decide that it is always safe to repeat certain operations (e.g., because they are idempotent). This problem is a consequence of the fact that writing the log record that records the effect of an invocation and replying to the invocation is not an atomic action in our system.

Changing and Finding Storesites

If objects can move in volatile memory, why should they be fixed in stable storage? Indeed, if a form is sent for approval to

a manager on the opposite coast, and the object consequently migrates to a machine there, it makes sense that the object be supported by a nearby storesite. Hermes therefore lets objects change storesite. However, the former storesite of an object keeps a forwarding pointer (in stable storage) to its successor. Now it is possible to find the current storesite for an object by a process similar to the one already described for Hermes nodes. Since by definition stable storage does not lose information in a crash and storesites are highly available, the search is likely to succeed at this stage.

Provided that all storesites are always available and that they keep all the forwarding addresses forever, every object in the system can be found in this way. But these conditions are not always true—storesites will occasionally be unavailable. Moreover, a storesite does not need to keep the checkpoint and log records of an object that migrates to another storesite once the object has checkpointed at the new storesite, so it would be convenient if it were eventually possible to purge the forwarding pointer too. We also need to provide a mechanism whereby the initial storesite for an object can be found.

One possible solution is to register each object with the name service, with an attribute indicating its current storesite. Once the name service has propagated this information and can guarantee that an enquiry will no longer be answered with old data, a storesite is free to delete its forwarding pointer.

It seems likely that many objects, particularly short-lived ones, might remain at their original storesite for as long as they live; hence, registering them with the name service could be wasteful. We therefore include an object's initial storesite as a component of its *guid.* This makes it easy to find an object's initial storesite, and also means that there is no need to make a name server entry for an object until it has moved its storesite for the first time. Moreover, assuming that most objects choose a nearby storesite, structuring *guids* from storesite names gives them geographic locality, thereby making them suitable for efficient lookup and update in a global but geographically clustered name service [17].

Having a *guid* and no other information, there are thus two ways of starting the finding process: derive the name of the initial storesite from the *guid,* or look up the storesite in the name service. If either operation fails to return information about the object, it is appropriate to try the other. The first method will be most efficient if the object has never moved its storesite; the second will be most efficient if it moved long ago. Which situation is more likely depends on the application. One can also seek to minimize elapsed time by making both enquiries in parallel.

A possible extension of this algorithm is to append a *tad* to the information stored in the name service. Now, when a supervisor asks the name service which storesite currently supports a target object, a *tad* for the object is returned too. The supervisor can try this *tad* immediately, instead of calling the storesite. This modification will pay off if objects move in stable storage as frequently as they move between Hermes nodes, and if the name service propagates updates sufficiently rapidly, because under these conditions the *tad* returned from the name service will be as good as the one at the storesite. Otherwise, the *tad* will be stale and trying it first will lengthen the search for the target object. Since it appears that these conditions will not be satisfied in our environment, we decided not to adopt this extension.

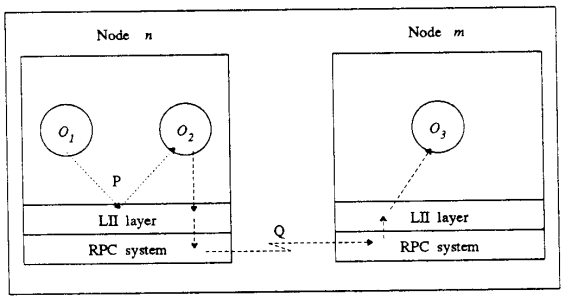


Fig. 2. Invocations through the LII layer.

V. Implementing the LII Mechanism

Conceptual View

Fig. 2 shows a conceptual view of two invocations (P and Q). The LII layer is the intermediary in both invocations, directing them to the correct node and object. In the first case (dotted line), object o_1 invokes operation P of o_2. The LII layer discovers that o_2 is local and calls $o_2.P$ locally. In the second case (dashed line), o_2 invokes operation Q on o_3. The LII layer issues an RPC to its peer on node m, which then calls $o_3.Q$ locally.

How is this redirection of invocations achieved? Imagine that for every object in the Hermes system there is a *proxy* object on every Hermes node. Both the real object and its proxies export the same interface, which consists of operations P_1, P_2, and P_3. However, whereas the real object's operations are bound to "real" procedures X_1, X_2, X_3 that perform the operations on the object, a proxy object's operations are bound to "stub" procedures X_1', X_2', X_3' that merely locate the real object and invoke the respective X_i procedure. Notice that because of our requirement for strong typing and because the operation name is not a parameter, we need a different stub for each operation.

When the invoker makes an invocation, it neither knows nor cares if it is operating on a real object or a proxy. In Fig. 2, when o_2 invokes $o_3.Q$, it actually calls the stub Q' in the local proxy for o_3. The proxy ascertains from the supervisor that o_3 is remote and calls $o_3.Q$ on the remote machine.

Implementation Details

We implement our proxies using machine generated procedures called LII stubs. Two stubs are generated for each procedure P in the interface of an object. The first stub (*P_prime*) is called by the application code, while the second (*P_prime_remote*) is located at the remote node and is called (via RPC) by *P_prime.* If *P_prime_remote* needs to forward the invocation a second time, it will make an RPC to an identical *P_prime_remote* on another node. An example stub *P_prime* is shown in Fig. 3(a); it has the same interface as the real procedure P except for the addition of the exception *Unavailable. P_prime_remote* is shown in Fig. 3(b). Notice

```
procedure P_prime (in target: ObjectId, <P's params> )
                  returns <P's return type>
                  raises { E1, ..., En, Unavailable};
var  myTad, remoteTad: TAD;
     binding: RPC.Binding;  re: RPC.CallErrors;
     exception: ExceptionDesc;
     result: <P's return type>;
begin
  if Object.TryToFix(target, myTad) then
     try return( P(target, <P's params>) )
     finally Object.UnFix(target)
     end try
  end if;
  /* object is remote: try RPC */
  remoteTad := myTad;
  loop
     try binding := LIISupport.GetBinding(myTad.node);
         if Client.P_prime_remote(binding,
                  remoteTad, 0, exception, result,
                  target, <P's params>) = Returned then
            LIISupport.UpdateTad(target, remoteTad);
            return(result);
         else
            LIISupport.UpdateTad(target, remoteTad);
            case exception.which of
            | e1: raise(E1, exception.arg)
            | unavailable:
                  raise(Unavailable, exception.why);
            | HopCountExceeded:
                  myTad := remoteTad;
            end case
         end if
     except
        | RPC.BindingFailed:
            if not Object.GotNewerTad(target) then
               raise(Unavailable, CannotBind);
            end if
        | RPC.CallFailed(re):
            if RPC.MightOccured(re) then
               raise(Unavailable, RPCmightOccured);
            elsif not Object.GotNewerTad(target) then
               raise(Unavailable, RPCnotOccured);
            end if
     end try
  end loop
end P_prime
```

(a) Stub called by object's clients

```
procedure P_prime_remote(in target: ObjectId; <P's params>;
                  in-out remoteTad: TAD; in-out hops: short;
                  out exception: ExceptionDesc;
                  out result: <P's return type> )
                  returns LIISupport.CallOutcome;
var myTad: TAD;          co: LIISupport.CallOutcome;
    binding: RPC.Binding;  re: RPC.CallErrors;
begin
  try if Object.TryToFix(target, myTad, remoteTad) then
        try try result := P(target, <P's params>);
                  return (Returned);
            except Ei (exception.arg):
                  exception.which := ei;
                  return (Raised);
            end try;
        finally Object.UnFix(target)
        end try
      end if
  except Unavailable (exception.why) :
            exception.which := unavailable;
            return (Aborted);
  end try;
  /* object is remote: try RPC */
  remoteTad := myTad;
  if hops >= LIISupport.MaxHops then
     exception.which := HopCountExceeded;
     return (Aborted);
  end if;
  loop
    try
       binding := LIISupport.GetBinding(myTad.node);
       co := Client.P_prime_remote(binding, remoteTad,
             hops+1, exception, result, target, params);
       LIISupport.UpdateTad(target, remoteTad);
       return (co);
    except
        | RPC.BindingFailed:
            if not Object.GotNewerTad(target) then
               exception.which := unavailable;
               exception.why := CannotBind;
               return(Aborted);
            end if
        | RPC.CallFailed(re):
            if RPC.MightOccured(re) then
               exception.which := unavailable;
               exception.why := RPCmightOccured;
               return(Aborted);
            elsif not Object.GotNewerTad(target) then
               exception.which := unavailable;
               exception.why := RPCnotOccured;
               return(Aborted);
            end if
    end try
  end loop
end P_prime_remote
```

(b) Stub called by other stubs remotely

Fig. 3. Pseudo-code for LII stubs.

that this stub has a different interface; it does not raise any exceptions, and returns as a result the outcome of the call as well as the result of *P* itself.[1]

The only state required by the stub is the location information stored in the supervisor's *tad* for the invoked object, so the same stub can be used for all the objects of a given type. Since stubs are almost identical, it is easy to generate them automatically. Notice that this is different from an RPC stub where almost everything in the body of the stub depends on the types of the parameters. Our stub generator scans the Modula-2+ interface definitions using the Unix™ tool *lex*, and is itself written in *awk*.

Both the local and remote stubs check whether the target object is local; if so, they call the local *P* and return its result to their caller. Otherwise, the remote stub at the node named in *myTad* is called. If the object is local, it is *fixed* for the duration of the call and is *unfixed* (by the **try**... **finally** construct) at the end of the call. The reason for fixing the object is to avoid migrating it while it contains an active thread, which would involve finding all location-dependent values on the thread's stack and in its registers and marshaling them between the two nodes. (The Emerald system actually does this, but it is possible only because the compiler distinguishes between location dependent and independent values [15].) In this respect moving a single object that is embedded in a shared address space is harder than migrating an entire address space [3].

The action of the stub that calls *P* depends on whether it is local or remote to the original client; the local stub can return directly to its caller even if *P* raises an exception, while the

[1] The existence of two stubs is largely a matter of convenience, influenced by the fact that in Modula-2+ a procedure that raises an exception cannot also return result parameters.

™ Unix is a trademark of AT&T Bell Laboratories.

remote stub has to catch whatever is returned or raised by *P*, package it, and return it together with the current *tad*. Conversely, after a stub completes an RPC, the remote stub can simply propagate the result returned from the call (and update its own *tad*), whereas the local stub has to unpackage the result and accordingly return to, or raise an exception in, the client application.

Notice that *TryToFix* takes an optional third argument, which is the best *tad* so far found during this invocation. In the remote stub [Fig. 3(b)], should *remoteTad* be newer than *myTad, TryToFix* obtains a newer *tad* from stable storage or from the name service, as discussed in Section IV. In attempting to do so, it may fail to communicate with them, in which case the exception *Unavailable* will be raised.

Alternative Implementations

We considered two alternative approaches for the implementation of location independent invocation.

1) In the *system-service approach,* the service both locates the object and performs the invocation. For example, suppose that this service is called *Invoke;* to perform the operation *P* of an object name by *o*, instead of calling *o.P(params)* or *P(o, params),* one calls *Invoke(o, P, params).* This is similar to the approach taken by Kalet and Jacky [16].

2) In the *integrated-mechanism approach,* an extended RPC system supports LII, that is, both finds objects and performs the invocations.

We ruled out the first alternative mainly because of its lack of strong typing: it requires that compile-time type checking be circumvented, whereas we wanted to take advantage of it. It also incurs the complexity of maintaining in one place a complete list of operations and their parameter types. With our *interface-layer* approach we maintain strong typing and avoid having such a list by generating a separate interface for each operation. The second alternative was rejected because our RPC system was developed and is maintained by others; it would require that we modify every new version of the RPC system to support LII. (This is not to say that RPC should not be extended to give better support to objects in general and mobile objects in particular, as discussed in Section VII). In addition, our choice of the interface-layer approach allows us to combine search and invocation and to make only minimal changes to the interface seen by an object's clients.

VI. Related Work

The notion of location independent invocation appears in the Eden distributed object-oriented operating system [1], [6]. Eden supports objects at the system level and it provides invocation as a system service. Eden uses *hints* to find objects, and is quite cavalier about timing out hints when they have been unused for a few minutes. If it is necessary to find an object for which there is no hint, or after a hint has failed, the object's capability is used as a hint to find a stable storage server, called a *checksite.* If this turns out not to be the current checksite for the object, the invoking system uses an Ethernet broadcast to interrogate *all* Eden kernels and *all* checksites in the Eden network. This is a reasonable approach for a research prototype of a dozen machines, but is inappropriate for a world-wide distributed system.

Eden deals with the problem of type-checking invocations in the same way as an RPC system: it generates a stub for each operation, which marshals the arguments into a uniform data structure. The system invocation interface takes this structure as its argument.

Emerald [7], [15] is a programming language that supports location-independent operations on mobile objects. The Emerald compiler packages invocation parameters directly, so there is no need for stubs. Emerald also uses forwarding addresses for location, but falls back on a broadcast enquiry (a *shout*) should the forwarding chain be broken. If there is no response to the *shout,* the invoking node performs an exhaustive search of *every* Emerald node, first by broadcasting a *search* message that requires a response from every node, and then by following up with reliable point-to-point messages to those nodes that have not responded.

Although Emerald predated Hermes, in several respects it is more advanced. Emerald objects can move at any time, even while invocations are executing inside them. New code can be injected into a running Emerald system, and will migrate around the system as necessary to support migrating objects. Emerald has an innovative object type system to ensure that invocations on newly-created objects can still be type-checked. These facilities are only feasible because Emerald is a programming language as well as a run-time system. While an attractive paradigm for the future, we judged that we were unlikely to successfully introduce a new programming language for commercial distributed applications. Moreover, Emerald currently has no provision for making objects persistent.

The problem of finding objects in Hermes resembles that of finding an interprocess communication channel in a system that supports process migration. There are various ways to redirect these channels after a process has migrated [2], depending on whether the IPC mechanism treats channels as hints (as in Demos/MP) [22]) or as absolutes. Our use of *tads* is similar to the unidirectional links used in Demos/MP. However, unlike *tads* in Hermes, forwarding addresses in Demos/MP are not updated when used, and they are discarded only if the process returns to the same machine.

Locus [9], MOS [4], and R* [18] use the "home" approach to record where an entity is located. In the distributed systems Locus and MOS the entities of interest are processes; in R* they are database objects. Each entity is assigned a home machine at birth; this machine is notified whenever the entity migrates. Sprite [12] extends this approach to direct location-dependent kernel calls to the home machine of a process, regardless of where the process is located. Hermes borrows from this approach by using an object's current storesite as a temporary "home."

The layering in Hermes and the support for LII resemble a method used to achieve location-independent resource access in the MOS system. The MOS kernel is a modified Unix kernel divided into three layers. The *lower-kernel* provides low-level, location-dependent kernel services, such as disk access. The *upper-kernel* provides user-level and location-

independent services, using the lower-kernel when necessary. Between these layers is the *linker,* which decides whether a call from the upper to the lower-kernel should be executed locally or remotely. The Hermes LII layer is analogous to the upper-kernel, the object operations to the lower-kernel, and the RPC system to the linker. Notice, however, that the targets of invocations in MOS (i.e., resources) do not migrate, so finding them is much simpler than in Hermes.

The system-service approach for implementing invocations without language support has been used in a large (but centralized) object-oriented application for radiation therapy [16]. This application was written in Pascal, with records representing objects. A "system" function *Invoke* is used to initiate all operations; it uses a case statement to select the right operation, and enqueues a "work request" to be performed by the system at some convenient time. To circumvent typing, arguments are passed as a record that has one variant per object interface and per operation. The operation name and target object are arguments too. This approach suffers from poor maintainability and readability: with every interface added to the system, the system administrator has to modify the Invoke function and the structures it uses. Invocation arguments must be correctly packaged into a record rather than being written as a list.

Another system built to support object mobility is that of Wolfson, Voorhees, and Flatley [26]. Unlike our work, they support migration between heterogeneous machines. Objects choose when and where to move based on their internal state. However, this system does not include any mechanism for locating an object from another node or communicating with it in a location independent fashion.

VII. Experience

The Hermes system has been designed and implemented over the course of about a year and a half, with most of the implementation completed by March 1989. At the time of writing (August 1989) the current prototype of Hermes has been running in the laboratory for a few weeks over a small number of machines on a LAN. Below we present some early performance measures and summarize our experience. The performance figures we offer here are necessarily very preliminary since no attempt has yet been made to tune the system's performance.

Performance

The measurements reported in Table II were taken on MicroVAX™ processors running on a dedicated Ethernet. A loop that issued a large number of calls was used to eliminate start-up effects. The time for a local null call represents the cost of a procedure call and return on this machine in our programming environment. "Local Null LII" refers to the cost of an location-independent call to an object that turns out to be local; the LII is roughly thirteen times as expensive. This factor is made up in part by real work, and in part by several procedure calls introduced by our extensive use of data abstraction, together with a compiler that fails to in-line trivial procedures. Note that a null LII does not correspond to a null procedure; a target object must be passed as an argument. The "real work" therefore includes passing this argument, checking whether the object that it names is in fact local, finding that object's representation, and using the dymanic type system to cast the representation to the type that the operation body requires. In addition, we must fix the object before the operation commences and unfix it on exit as discussed in Section V.

™ Ultrix and MicroVAX are trademarks of Digital Equipment Corporation.

TABLE II
PERFORMANCE MEASUREMENTS FOR INVOCATION

operation	time	Δ
Local Null Call	18 μs	
Local Null LII	240 μs	
Round trip UDP	10 ms	
Null RPC	22 ms	12 ms
Null LII	26 ms	4 ms
256 byte RPC	26 ms	
256 byte LII	30 ms	4 ms

Note that most of this work must be done regardless of the approach taken to implement LII, or indeed even if LII is not supported, provided that objects can move. With our current compiler, some of the overhead can be eliminated by breaking open the encapsulated data structures of the supervisor "by hand," but this approach is error prone. Only a minor fraction of the measured overhead of LII can be attributed to the fact that the stubs are machine generated and not hand tuned to each interface.

The second segment of the table shows the cost of a round trip from one Ultrix™ user process to another on a remote machine using UDP. The RPC mechanism that we used (the experimental RPC from Digital's Systems Research Center) requires one such round trip per call in the normal case; the time for RPC with no parameters is shown. (For a thorough analysis of the performance of this RPC on a *different* substrate, see [25].) The null LII call incurs an overhead of 4 ms relative to the null RPC. This timing is based on having an up-to-date *tad* for the invoked object at hand, and having the binding for the remote node already cached. Recall that a null LII must still send the identity of the called object; it also marshals result parameters that are used in the case of RPC failure and for the remote *tad* that is sent back when the object is found. Similarly, the timing for a 256 byte RPC is not strictly comparable to that for a 256 byte LII: the latter also involves sending an object reference, receiving exception and call-outcome descriptors, and both sending and receiving *tads* and hop counts. As a consequence, not only are the messages larger (by about 35 bytes), but extra procedures must be called to marshal these extra parameters.

To assess the cost of migration, we measured the time taken to interrogate the state of a small form object locally and remotely, and compared this to the time taken to migrate the object. Unlike the timings given above, these measurements were not made in an isolated laboratory environment, but represent a "real" use of the system, with other processes and the window manager running concurrently. They therefore include system overhead for context switching externally

TABLE III
PRELIMINARY TIMINGS FOR FORM INVOCATIONS

	User interface and Hermes Node are: co-located	remote
object is local	200 – 240 ms	80 – 140 ms
object is remote	290 ms	370 – 660ms
penalty for remote invocation	50 – 90 ms	300 – 500 ms

between processes as well as internally between threads, and the overhead of other network traffic (which was measured to be about 11 percent). Moreover, single calls were timed, rather than loops that issued large numbers of calls, so startup, paging, and swapping effects may be significant. For these reasons the timings are reported as ranges rather than as single numbers.

All the invocation measurements reported in Table III include issuing a request from the user interface (a separate Ultrix process in the current prototype) and receiving the results back at the interface. The RPC's between the interface and the Hermes Node used the "opaque reference" mechanism, and marshaled around 90 bytes of structured data. The left column reports the timings for the case where the user interface and the Hermes Node were sharing the same machine; in the right column they were on separate machines. The remote case is faster, indicating that the machines were loaded enough for the gain in parallelism to outweigh the cost of communication. The first row refers to the case where the invoked object was at the Hermes Node at which the interface initiated the invocation; in other words, the LII was local. In the second row the object was remote from the invoking Hermes Node, and the LII mechanism was thus forced to make an RPC. However, the timings in the bottom right-hand corner are unrealistic since only two nodes were used for the tests; the user interface and the invoked object's Hermes Nodes shared one machine, while the interface initiated its invocations at a Hermes Node on a second machine. (The 660 ms timing is particularly suspect, and may indicate that the first machine was swapping.)

Looking at the timings in the first column, the penalty for remote invocation agrees fairly well with the measurements reported in Table II, given that the invocation returned a complex data type with several array, record, and structured text fields that must be marshaled by calling procedures. The penalty derived in the second column is much larger and requires further analysis.

Migration involves two remote procedure calls: one that passes the object to the destination node, and a second to commit the move and update the object's location information at its storesite. For our experiments we simulated stable storage using the Ultrix file system. We measured the overhead of calling the storesite and committing the move to be 150–230 ms. A careful implementation of stable store, as used in a commercial database, might take 10 ms to write a commit record; even after adding the cost of the RPC, our simulation is five to eight times slower. The measured elapsed time for migrating forms of two different sizes are reported in Table IV.

The cost of moving a small object thus appears to be about four times the penalty for invoking it remotely. Thus, on the one hand, if a single invocation needs to be made on an object that already resides at a remote node, the object should be invoked remotely. On the other hand, if several invocations will be made, it will probably be beneficial to move the object. Migration also makes sense when the system can determine in advance where an object will be required, as it usually can in the expense form application; in this case the object can be moved in the background before an interactive session starts.

TABLE IV
PRELIMINARY TIMINGS FOR OBJECT MOVE

size of form moved	time
252 bytes	190 – 310 ms
3176 bytes	690 – 870 ms

For comparison, the cost of migrating a trivial object in Emerald is reported to be 2 ms above the cost of a remote invocation [15]. Of course, this does not include the cost of committing the object's new location to stable storage; neither does it include marshaling or de-marshaling, since the representations of Emerald objects are well-known. The cost of marshaling is significant. For example, process migration systems (supported at the operating system kernel) are able to move medium-sized address spaces (say, 32 kbyte) in times that are comparable to that taken for a small Hermes object (say 200–300 ms) largely because they avoid marshaling [3].

Reliability

One of our initial goals was to create a reliable environment for applications despite the inevitability of machine and network failure in very large distributed systems. Our system does not lose object data or location information when a node fails, although it will do so if "stable storage" turns out not to be stable. However, Hermes is not fault tolerant because an operation can be rejected if the node where the target object is located or the storesite supporting it are temporarily unreachable. Nevertheless, despite the ability to scale to very large number of nodes spread over a large geographic area, failures, are "soft"; that is, although it is possible for a particular object to be temporarily unavailable, the system as a whole will still function and operations on other objects can proceed.

The main tool that we used to increase reliability was the almost complete elimination of residual dependency. Although finding a remote object may require one to traverse several nodes, the failure of an intermediate node does not necessarily reduce the object's availability or the system's ability to locate the object. The failure can be circumvented by using the storesite, and in some circumstances even storesite failure can be circumvented by using the name service. In contrast, in other systems that use forwarding addresses (e.g., Demos/MP), the failure of any node containing a forwarding address to an object renders it unavailable.

Finally, our approach introduces some uncertainty concerning invocations. Using a reliable RPC system, one can normally detect and discard duplicate invocations. However, when the RPC fails, it is not always possible to ascertain whether the remote operation has taken place or not. Our ability to reactivate an object from stable storage does not allow us to recover from these failures transparently. In these cases we report failure and expect higher layers to decide

whether reissuing the invocation is necessary or safe. Making this recovery automatic would require adding a sequencer to invocation requests, and ensuring that the sequencers of completed invocations are written to stable storage when an object logs or checkpoints. Transaction-based systems recover from this sort of failure in just this way.

Using RPC

The complete separation of the LII layer from the RPC system is practical rather than conceptual. In fact, we believe that much of the support for LII might migrate into the RPC systems of the future. In particular, the RPC system should allow control over the granularity of the target of a binding, instead of dictating the grain to be an address space or module. The RPC system that we are using supports binding to separate modules, not merely to an address space. It does not facilitate combining the interfaces of several modules that share the same address space: every remotely-callable interface must be called via a separate binding. Used naively, this mechanism would necessitate constructing and keeping at hand a dozen or more bindings to every other supervisor. To eliminate these extra bindings we have designed an automatic translator that, given a list of object interfaces, produces a single "umbrella" interface by systematic renaming. Fortunately, the Modula-2+ language enables this merging of interfaces to be done without incurring the cost of extra procedure calls.

RPC systems usually take pains to ensure that a remote call behaves as if it is executed by the calling thread: alerts (inter-thread "interrupts") sent to the calling thread are propagated to the callee, and exceptions raised by the callee are propagated back to the caller. Although our RPC system enjoys the latter property, it does not help us, for when a called procedure raises an exception we need to propagate not only the exception but also the *tad*. The LII stubs therefore duplicate this functionality, and at considerable inconvenience, because exception codes are not manipulable values in the Modula-2+ programming language.

Our insistence on passing the result and *tad* all the way back to the original invoker is based on the assumption that the propagation chains will usually be short. If the chain becomes long, this incurs a significant overhead in marshaling and de-marshaling both the arguments and the results in the intermediate nodes. Our design might have changed had the RPC system provided a simple way to return directly from the end of the chain to the original invoker; this is essentially the same optimization that a compiler might perform in eliminating a tail-call from a procedure. Failing this, some way of avoiding the repeated de-marshaling and marshaling of arguments at the intermediate nodes would be beneficial, although it is not at all clear how to achieve this while preserving the RPC abstraction.

Language Extensions

Our experience writing a distributed object-oriented system with a language that is not geared towards object-oriented programming showed us that several language features are necessary to facilitate writing distributed applications and services.

- *Concurrency* is endemic in a distributed system; such a system includes multiple processors and (often) multiple users, and concurrent activity is the norm. The programming language should provide integrated support for multithreading, including synchronization primitives for the access of shared data.
- *Parameter lists* should be *first-class* data objects. This would greatly simplify the kind of procedure call redirection that is performed by the LII layer, while maintaining strong typing. Ideally, a procedure should be able to call another procedure by *Module.MyName(MyParams)* where *MyName* is dynamically bound to the caller's name for the procedure and *MyParams* is a parameter list data object. (*MyName* is similar to the *argv*[0] notion of Unix.)
- *Polymorphic procedures*, as in Russell [10], and implicit parameterization by types, would allow us to write a single stub that could be used for every operation.
- If *Exceptions* were manipulable as *data objects,* the task of writing our stubs would have been significantly simplified, as mentioned above. At the level of the language implementation exceptions are indeed data objects, and it is not clear why this is hidden from the user.

In addition, while Modula-2+ does have reasonable data typing facilities, they are quite inconvenient to use. For example, while combining interfaces it is necessary to automatically generate a type that can represent the value of one of a number of existing enumerations. This can be done as a variant record (which is very clumsy) or an ordinary record (which wastes space). A true discriminated union of types would have been much easier to use.

VIII. Summary

This paper presented some of the problems of finding objects in large and wide-area networks where objects may change their location in volatile memory as well as on stable storage. We discussed possible solutions and described those adopted in the Hermes system. We designed and developed a location-independent-invocation mechanism that combines finding with invocation, using temporal location information. The mechanism also updates the system's knowledge of an object's location as a side-effect of invocation and object migration. Based on our assumptions about object mobility, objects are likely to be found within a few propagations of an invocation. If they cannot be found in this way, stable-storage and name services are used to locate the object. The major contribution of this paper is to show how LII can be achieved in a large and dynamic environment in which objects are supported by neither the operating system nor the programming language.

Acknowledgment

D. Ting and R. Sudama helped in the basic design of the finding algorithm and the LII mechanism; K. Somlawar and K. Alagappan both participated in the development effort. P. Tzelnic helped keep us pointed in the right direction. We thank T. Wobber and A. Birrell of Digital SRC for making available Modula-2+ and the RPC system, and for assisting us with its use.

REFERENCES

[1] G. T. Almes, A. P. Black, E. D. Lazowska and J. D. Noe, "The Eden system: a technical review," *IEEE Trans. Software Eng.*, vol. SE-11, pp. 43–59, Jan. 1985.

[2] Y. Artsy and R. Finkel, "Simplicity, efficiency, and functionality in designing a process migration facility," in *Proc. 2nd Israel Conf. Comput. Syst. Software Eng.*, Tel-Aviv, Israel, May 1987, pp. 3.1.2: 1–12.

[3] ——, "Designing a process migration facility—The Charlotte experience," *IEEE Computer*, vol. 22, no. 9, pp. 47–56, Sept. 1989.

[4] A. B. Barak and A. Litman, "MOS: A multicomputer distributed operating system," *Software—Practice & Experience*, vol. 15, no. 8, pp. 725–737, Aug. 1985.

[5] A. D. Birrell and B. J. Nelson, "Implementing remote procedure calls," *ACM Trans. Computer Syst.*, vol. 2, no. 1, pp. 39–59, Feb. 1984.

[6] A. P. Black, "Supporting distributed applications: Experience with Eden," in *Proc. 10th ACM Symp. Operating Syst. Principles*, Dec. 1985, pp. 181–193.

[7] A. P. Black, N. Hutchinson, E. Jul, H. M. Levy and L. Carter, "Distribution and abstract types in Emerald," *IEEE Trans. Software Eng.*, vol. SE-13, pp. 65–76, Jan. 1987.

[8] A. P. Black and Y. Artsy, "Implementing location independent invocation," in *Proc. 9th Internat. Conf. Distributed Computing Syst.*, Newport Beach, CA, June 1989, pp. 550–559.

[9] D. A. Butterfield and G. J. Popek, "Network tasking in the Locus distributed UNIX system," in *Proc. Summer USENIX Conf.*, June 1984, pp. 62–71.

[10] A. Demers and J. Donahue, "Revised report on Russell," Tech. Rep. 79-389, Dep. Computer Sci., Cornell University, Ithaca, Sept. 1979.

[11] Digital Equipment Corporation, *DNA Naming Service Functional Specification, Version 1.0.1.* Digital Equipment Corporation, Maynard, MA, Nov. 1988, Order Number EK-DNANS-FS-001.

[12] F. Douglis and J. Ousterhout, "Process migration in the Sprite operating system," in *Proc. 7th Internat. Conf. on Distributed Computing Syst.*, Berlin, West Germany, Sept. 1987, pp. 18–25.

[13] R. J. Fowler, "Decentralized object finding using forwarding addresses," Ph.D. thesis, Dep. Computer Sci., Univ. Washington, Dec. 1985.

[14] ——, "The complexity of using forwarding addresses for decentralized object finding," in *Proc. 5th Annu. ACM Symp. Principles of Distributed Computing*, Aug. 1986.

[15] E. Jul., H. Levy, N. Hutchinson, and A. Black, "Fine-grained mobility in the Emerald system," *ACM Trans. Computer Syst.*, vol. 6, no. 1, pp. 109–133, Feb. 1988.

[16] I. J. Kalet and J. P. Jacky, "An object-oriented programming discipline for standard pascal," *Comm. ACM*, vol. 30, no. 9, pp. 772–776, Sept. 1987.

[17] B. W. Lampson, "Designing a global name service," in *Proc. 5th Annu. ACM Symp. Principles of Distributed Computing*, pp. 1–10, Aug. 1986.

[18] B. Lindsay, "Object naming and catalog management for a distributed database manager," in *Proc. 2nd Internat. Conf. Distributed Computing Syst.*, Paris, France, Apr. 1981, pp. 31–40.

[19] P. McCartney, "It's getting better," in *Sergent Pepper's Lonely Hearts Club Band*. BMI, 1967.

[20] B. J. Nelson, "Remote procedure call," Tech. Rep. CSL-81-9, Xerox PARC, May 1981.

[21] J. Noe, A. Proudfoot and C. Pu, "Replication in distributed systems: The Eden experience," in *Proc. Fall Joint Computer Conf.*, Nov. 1986, pp 1197–1209.

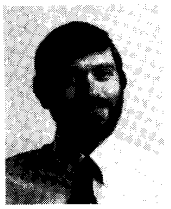
[22] M. L. Powell, and B. P. Miller, "Process migration in DEMOS/MP," in *Proc. 9th ACM Symp. Operating Syst. Principles*, Oct. 1983, pp. 110–119.

[23] C. Pu, J. D. Noe, and A. Proudfoot, "Regeneration of replicated objects: A technique and its Eden implementation," *IEEE Trans. Software Eng.*, vol. 14, pp. 936–945, July 1988; also appears in *Proc. Second Internat. Conf. Data Eng.*, Los Angeles, CA, Feb. 1986, pp. 175–187.

[24] P. Rovner, "Extending Modula-2 to build large, integrated systems," *IEEE Software*, vol. 3, no. 6, Nov. 1986, pp. 14–57.

[25] M. D. Schroeder and M. Burrows, "Performance of Firefly RPC," in *Proc. 12th ACM Symp. Operating Syst. Principles*, Litchfield Park, AZ, Dec. 1989.

[26] C. D. Wolfson, E. M. Voorhees, and M. M. Flatley, "Intelligent routers," in *Proc. 9th Internat. Conf. Distributed Computing Syst.*, Newport Beach, CA, June 1989, pp. 371–376.



Andrew P. Black was born in London, England. He received the B.Sc. (Hons.) degree from the University of East Anglia in computing studies, and the D.Phil degree from the Programming Research Group of the University of Oxford in programming languages and software engineering.

He was on the faculty of the Department of Computer Science at the University of Washington, Seattle, from 1981 to 1986, where he worked primarily on distributed systems and languages, and on the way that these areas interrelate. Since then he has been with Digital Equipment Corporation in Distributed Systems Advanced Development Group and the Cambridge Research Laboratory.

Yeshayahu Artsy (S'85–M'88) received the M.S. and Ph.D. degrees in computer sciences from the University of Wisconsin, Madison, in 1984 and 1987, respectively; the B.A. degree in economics and MBA degree in management information systems from Tel-Aviv University in 1979 and 1981, respectively; and the B.A. degree in political sciences and in statistics from the Hebrew University of Jerusalem in 1975.

He is now with the Distributed Systems Advanced Development Group at Digital Equipment Corporation since October 1987. Prior to joining DEC he managed the Tel-Aviv University Management School Computing Center (1976–1979), was in charge of system software support for Burroughs Medium Systems in Israel (1979–1982), and participated in the design and implementation of the Charlotte distributed system in the University of Wisconsin (1983–1986), responsible in particular for the IPC and process migration mechanisms. His interests include distributed operating systems and services, open systems, and object-oriented design.

Dr. Artsy is a member of ACM and the IEEE Computer and Communication Societies.