Implementing Location Independent Invocation

Andrew P. Black and Yeshayahu Artsy

Distributed Systems Advanced Development,
Digital Equipment Corporation

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. 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 techniques used to find the target of an invocation, and describes how our technique is implemented.

I. Introduction

The motivation behind RPC [18] 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 [4]. 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.

Authors' address: 550 King Street, LKG1-2/A19, Littleton, Massachusetts 01460. Electronic mail: black@dsmail.dec.com, artsy@dsmail.dec.com.

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*, where the RPC system chooses the callee, either non-deterministically, or in a way that depends on factors that are normally hidden from the caller, such as network distance; and
- *using a clerk*, in which an application dependent subroutine package uses its knowledge of 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.

Location Independent Invocation (LII) is another 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 [1,5] incorporates a similar concept) and as part of a programming langauge (Emerald [13]). 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 inter-process communication.

The remainder of this paper is organized as follows. In Section II we study the "indications" for location independent invocation, i.e., under what circumstances the abstraction 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 the current state of our work, and reports some early conclusions.

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 legal calls that are acceptable to a given 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.

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 [9] 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? 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 ability to utilize the special capabilities of a particular machine. In addition, object mobility provides a convenient way for application data to be moved to the point where it is needed without the necessity of making a copy and keeping it consistent with the original [13].

Any mechanism for moving objects that allows them to be found again afterwards, and to be recovered in the event of a failure, is bound to impose some overhead. However, the benefit of mobility can be increased responsiveness at other times. Imagine an application that spans a network of machines installed in offices, manufacturing plants, or hospitals and which is dispersed over a large geographic area. In such an application forms, production plans, or patients' files can be implemented as objects, each with a substantial data content. These objects might be intensively updated or accessed in real time in one place, and then at some future time updated again in another place. If the time interval between interactive sessions is relatively long (say, minutes or tens of minutes) and the interactive response for the update sessions is important, it may make sense to move the objects in between sessions almost regardless of the cost of migration. In contrast, if the major requirement is, for example, to minimize network traffic, then a simpler design in which objects are held at a central server and updated remotely might be more appropriate.

III. The Hermes System

To investigate the use of mobile objects we are trying to automate a corporation-wide, real-life office application.

The application we have chosen deals with expense vouchers 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. 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 30 000 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 forms as objects 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, 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.

The Hermes system provides infrastructure necessary to support such objects. It is composed of Hermes *nodes*, each consisting of a number of components (see Figure 1): a Hermes Supervisor, a collection of application objects, and an inter-object communication layer. The supervisor provides services for object creation, finding, and relocation; the inter-object communication layer consists of an LII package built upon an 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.

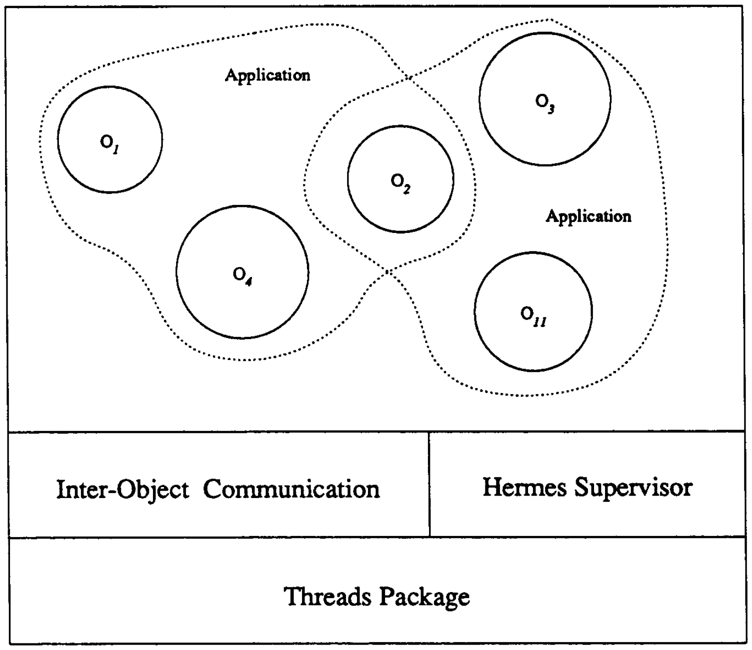

Figure 1: A Hermes Node – Internal View

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.

Hermes also provides a storage service that is used to make objects persistent; it 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 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 object. 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. 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+ [20] because of its provision of threads and because of the integration of the multithreading facilities with the RPC system.

IV. Finding objects in Hermes

This section commences by introducing some terminology, and then offers an overview of the finding algorithm. A more detailed description of the finding process appears in the following three sub-sections.

Every object is assigned at creation and keeps forever a globally unique identifier (*gUId*). Each object also has a location (the name of the Hermes node that contains the volatile copy of the object) and an *age*, which is initially zero and is incremented whenever the object moves to another location. If object o is at node n after its a^{th} move, then the pair $<n,a>$ is a *temporal address descriptor* (*tad*) for o. A *tad* $<n_1, a_1>$ is newer than a *tad* $<n_2, a_2>$ for the same object if and only if $a_1 > a_2$. References to objects are essentially *gUId*s, but 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 *tad*s; 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 *tad*s 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 reliable techniques to locate 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*. This process is repeated until it succeeds or there is no newer *tad*. In effect, we are using the information in the *tad*s as forwarding addresses; this is inexpensive, since a newer *tad* is normally obtained as a side effect of a failed invocation.

If we run out of good forwarding addresses the algorithm resorts to finding the object 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 some 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 pointers 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 probably high, since name services are usually engineered for a high ratio of enquiries to updates. We expect 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* $<n, 0>$, 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* <*m*, 1>. 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 (discussed in Section 5) 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*. 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 reduces the hybrid approach to the first one. Based on Fowler's work on object finding [11, 12], 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 *tad*s 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 *tad*s 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" [17].

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. Fowler's work encourages us to believe that chains will usually be short, and one of the objectives of our implementation is to verify his assumptions experimentally. Nevertheless, when a forwarding pointer is not available, we must have an alternative strategy.

Using Storesites and the Name Service to Find Objects

When the object-finding algorithm reaches a point where there are no more *tad*s to chase (i.e., no *tad* was returned with a failed invocation or the *tad* returned is older than one that has already failed), 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 has been used for Emerald [13], 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 stored with 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; in either case they take with them *tad*s for all of their object references. Whether these *tad*s are useful depends on how long the object has been "idle" at the storesite.

Rather than letting a Hermes node's stash of *tad*s be lost in a crash, the supervisor might write the *tad*s 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, this is expensive, and may be of little benefit: if the node stays down for any length of time many of the *tad*s will be of purely historic interest. We have not yet chosen a policy for committing *tad*s to stable storage; some compromise seems inevitable.

If a node *n* does lose its *tad*s in a crash, or even if it recovers historic *tad*s from stable storage, after *n* has come back up some 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*, and that the storesite has not learned that *n* has crashed, along with all the objects that it contained. The location of *o* will therefore still be recorded as *n*. In this case, *o* needs to be recovered and reactivated. To distinguish this case from that in which *o* migrated to *n* after it had recovered, it is necessary for the storesite to know the time (by *n*'s real-time clock) at which *n* last rebooted, and also the time at which *o* moved to *n*.

Notice that stable storage alone does not always help in 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. However, if it were possible that the invocation completed and that the node crashed just before sending the reply, then trying to locate 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).

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 nameserver entry for an object until it has moved its storesite for the first time. Moreover, assuming that most objects choose a nearby storesite, structuring *gUId*s from storesite names gives them geographic locality, thereby making them suitable for efficient lookup and update in a global but geographically clustered name service [15].

Starting with 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 lookup the *gUId*'s storesite in the name service. 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. We plan to experiment with the trade-offs in this area and to defer our choice until the behavior of objects is better understood.

V. Implementing the LII mechanism

Strategy

We considered three approaches for the implementation of location independent invocation.

(1) The *system-service approach*: the service both locates the object and performs the invocation. For example, suppose that this service is implemented by the procedure *Invoke*; to perform the operation *P* of an object named by *o*, instead of calling *o.P(params)* one calls *Invoke(o, P, params)*. This is similar to the approach taken by Kalet and Jacky [14].

(2) The *integrated-mechanism approach*: extend the RPC system to support LII, that is, to find objects and

perform the invocations.

(3) The *interface-layer approach*: provide a separate mechanism above RPC. Every invocation must then pass through this layer.

We used the following criteria to evaluate these strategies.

- The mechanism should preserve type information and be modular, in the sense that a change in the interface of one object should not affect the entire system. These requirements clash with the system-service approach, unless the service interface is polymorphic.
- It should not require changes to the underlying tools and language, which should be generic and supported by others. This criterion effectively rules out the integrated-mechanism approach.
- Search and invocation should be combined, as motivated in Section IV.
- The existence of LII should not change the interface seen by the programer, who should be able to write location-indpendent invocations as simply as local procedure calls (modulo possible inaccessibility problems). Extra information needed for LII should be provided automatically.
- It should be easy to change the finding algorithm as well as the way in which location information is managed, without changing application code.
- The overhead imposed by the mechanism should be minimal, beyond that inherent in locating an object .

The first two criteria effectively rule out all but the interface-layer approach. Except for systematically renaming procedures to include class names, our implementation preserves the object interface seen by the user. Below we first describe the implementation conceptually and then give some details.

Conceptual view

Figure 2 illustrates our approach. The LII layer is the intermediary in every invocation, directing it 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 $X_1, \cdots, X_3$ that perform the operations on the object, a proxy object's operations are bound to "stub" procedures $X'_1, \cdots, X'_3$ 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 one 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 Figure 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, one for each procedure in the interface of an object's type. The only state required by the proxy is the location information stored in the supervisor's *tad* for the invoked object, so proxies are

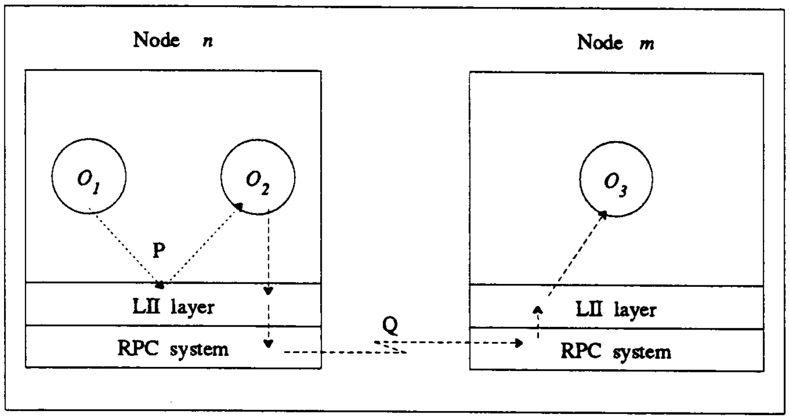

Figure 2: Invocations through the LII layer

inexpensive in terms of data space.

An LII stub is a three-way switch.

(1) If the object is local, the stub calls the object's operation directly, and returns whatever result or exception that operation returns. While the call is in progress the stub ensures that the object is *fixed* at the local node.

(2) If the descriptor contains a "better" *tad* for the object, the stub issues a remote procedure call to an identical LII stub on the node named by the *tad*. If the call is successful, the stub returns to the caller whatever result or exception is returned from the RPC together with the current *tad*. Otherwise, this step can be repeated; at each iteration the Hermes supervisor's *tad* for this object is updated with the newer information.

(3) If a good forwarding address is not available, the stub executes the "full" finding algorithm using storesites and the name service. This involves repeating steps (1) and (2) as appropriate.

A time-space trade-off is involved in deciding how much of the stub is inline code and how much is implemented by calling procedures. Only the call to the real object and the RPC to the remote LII stub are different from stub to stub; the other sections of the stub can be replaced by calls to standard procedures.

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*.

VI. Related Work

The notion of location independent invocation appears in the Eden distributed object-oriented operating system [1,5]. 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.

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

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 [6,13] 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.

Emerald is more advanced than Hermes in several respects. 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 also 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 [19]) or as absolutes. Our use of *tad*s is similar to the uni-directional links used in Demos/MP. However, unlike *tad*s 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 [7], MOS [3] and R* [16] 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 [10] 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 to implement invocations without language support has been used in a large (but centralized) object-oriented application for radiation therapy [14]. 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.

VII. Conclusion

This paper presented some of the problems of finding an object in large and dynamic networks, where objects may change their location in volatile as well as stable storage. We have discussed possible solutions and shown those adopted in the Hermes system. We designed and developed a location-independent-invocation mechanism that combines finding with invocation, using some temporal location information. The mechanism also updates this information as a side effect of invocations. Based on our assumptions of object mobility and opportunities to update such information, 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, where objects are supported by neither the operating system nor the programming language.

Our 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 to 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 to be done without incurring the cost of extra procedure calls.

Our experience with writing a distributed object-oriented system with a language that is not geared towards object-oriented programming taught 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 possibly 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 *argc[0]* notion of Unix.)
- *Polymorphic procedures*, as in Russell [8], and implicit parameterization by types would allow us to write a single stub that could be used for every operation.

Major parts of the Hermes system have been designed and implemented. The LII mechanism has been tested, and we expect a prototype system to be complete within the next few months.

Acknowledgements

Thanks go to other members of our group: Dennis Ting and Ram Sudama helped in the basic design of the finding algorithm and the LII mechanism; Kiran Somlawar and Kannan Alagappan are both participating in the development effort. We thank Ted Wobber and Andrew Birrell of Digital SRC for making available Modula-2+ and the RPC system, and for assisting us with its use.

References

[1] Almes, G. T., Black, A. P., Lazowska, E. D. and Noe, J. D. "The Eden System: A Technical Review". *IEEE Trans. on Software Eng.* **SE-11**, *1* (January 1985), pp.43-59.

[2] Artsy, Y. and Finkel, R. "Simplicity, Efficiency, and Functionality in Designing a Process Migration Facility". *Proc. 2nd Israel Conf. Computer Systems and Software Engineering*, Tel-Aviv, Israel, May 1987, pp.3.1.2: 1-12.

[3] Barak, A. B. and Litman, A. "MOS: A Multicomputer Distributed Operating System". *Software—Practice & Experience* **15**, *8* (August 1985), pp.725-737.

[4] Birrell, A. D. and Nelson, B. J. "Implementing Remote Procedure Calls". *Trans. Computer Systems* **2**, *1* (February 1984), pp.39-59.

[5] Black, A. P. "Supporting Distributed Applications: Experience with Eden". *Proc. 10th ACM Symp. on Operating Systems Prin.*, December 1985, pp.181-193.

[6] Black, A. P., Hutchinson, N., Jul, E., Levy, H. M. and Carter, L. "Distribution and Abstract Types in Emerald". *IEEE Trans. on Software Eng.* **SE-13**, *1* (January 1987), pp.65-76.

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

[8] Demers, A. and Donahue, J. "Revised Report on Russell". Tech. Rep. 79-389, Department of Computer Science, Cornell University, Ithaca, September 1979.

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

[10] Douglis, F. and Ousterhout, J. "Process migration in the Sprite Operating System". *Proc. 7th Int. Conf. on Distributed Computing Systems*, Berlin, West Germany, September 1987, pp.18-25.

[11] Fowler, R. J. *Decentralized Object Finding Using Forwarding Addresses.* Ph.D. Thesis, University of Washington, Dept of Computer Science, December 1985.

[12] Fowler, R. J. "The Complexity of Using Forwarding Addresses for Decentralized Object Finding". *Proc. 5th ACM Symp. on Prin. Distributed Computing*, August 1986.

[13] Jul, E., Levy, H., Hutchinson, N. and Black, A. "Fine-Grained Mobility in the Emerald System". *Trans. Computer Systems* **6**, *1* (February 1988), pp.109-133.

[14] Kalet, I. J. and Jacky, J. P. "An Object-Oriented Programming Discipline for Standard Pascal". *Comm. of the ACM* **30**, *9* (September 1987), pp.772-776.

[15] Lampson, B. W. "Designing a Global Name Service". *Proc. 5th ACM Symp. on Prin. Distributed Computing*, August 1986, pp.1-10.

[16] Lindsay, B. "Object Naming and Catalog Management for a Distributed Database Manager". *Proc. 2nd Int. Conf. on Distributed Computing Systems*, Paris, France, April 1981, pp.31-40.

[17] McCartney, P. "It's Getting Better", in *Sargent Pepper's Lonely Hearts Club Band.* BMI, 1967.

[18] Nelson, B. J. "Remote Procedure Call". Tech. Rep. CSL-81-9, Xerox PARC, May 1981.

[19] Powell, M. L. and Miller, B. P. "Process Migration in DEMOS/MP". *Proc. 9th ACM Symp. on Operating Systems Prin.*, October 1983, pp.110-119.

[20] Rovner, P. "Extending Modula-2 to build large, integrated systems". *IEEE Software* **3**, *6* (November 1986), pp.14-57.