The Amadeus GRT – Generic Runtime Support for Distributed Persistent Programming

Vinny Cahill, Seán Baker, Chris Horn and Gradimir Starovic

Distributed Systems Group,
Dept. of Computer Science, Trinity College Dublin, Ireland

Distributed Systems Group
Department of Computer Science
University of Dublin
Trinity College, Dublin 2, Ireland.
Fax: +353-1-772204

Document Status	Published
Distribution	Public
Document #	TCD-CS-93-37
Publication	OOPSLA '93 Conference Proceedings, pages 144-161

The Amadeus GRT – Generic Runtime Support for Distributed Persistent Programming

Vinny Cahill, Seán Baker[*], Chris Horn and Gradimir Starovic

Distributed Systems Group,
Dept. of Computer Science, Trinity College Dublin, Ireland

Abstract

Many object-oriented programming language implementations have been extended to support persistence, distribution or atomicity by integrating the necessary additional support with the language's runtime library.

We argue that a better approach is to provide a *Generic Runtime* library (the GRT) which implements that part of the support which is independent of any language. The GRT should be designed to interface to a language's existing runtime in such a way that the language's local object reference format and invocation mechanism can be retained. Hence existing compilers do not necessarily have to be modified, and a range of different object-oriented languages can be supported simultaneously.

This approach has significant merits including: the ease with which a language can be extended; the sophistication of the underlying support immediately available to a language implementer; and the ability to support fine-grained language interworking.

[*]email: vinny.cahill@cs.tcd.ie, sean.baker@cs.tcd.ie

1 Introduction

Amadeus [11, 4] is a general purpose, object-oriented programming environment supporting distributed and persistent applications in multi-user, distributed systems. While previous work has typically concentrated on designing or adapting a single language either for distribution [21, 23, 6, 20] or persistence [1, 7, 2], Amadeus can support a variety of existing object-oriented programming languages (OOPLs). This is important since any large distributed system will be diverse not only in its hardware but also in its language requirements. Our goal was to support a number of languages on one distributed platform without unnecessary duplication of effort, while facilitating language interworking.

In Amadeus, the implementers of an OOPL can choose between a set of (inter-related) properties, including persistence, distribution and atomicity for its objects. A further goal of Amadeus was to allow a language to be extended without imposing entirely new constructs and models on the base language. That is not to say that a language's syntax should not be changed to reflect the provision of persistent, remotely accessible or atomic objects, but this choice, and the style of change, if any, should be made per-language rather than being imposed by Amadeus.

To date we have concentrated on allowing persistence, distribution and atomicity to be added to languages whose base versions do not support these features. Amadeus currently supports extended versions of both C++ [9] and Eiffel [16]. We are currently investigating what extensions may be necessary to Amadeus in order to support languages designed specifically to support distribution or persistence.

The goal up to this point has been to allow persistence, distribution and atomicity to be added to a language without necessarily requiring changes to its compiler. In particular, Amadeus does not require changes to a language's native object reference format or operation invocation mechanism.

Typically, a language to be supported will already have its own execution structures implemented by a compiler or preprocessor and a Language-Specific Runtime library (LSRT). Amadeus provides a *Generic Runtime* library (the GRT) on top of which these individual LSRTs may be supported. The GRT provides the support for persistence, distribution and atomicity which is required by a range of language implementations but which is independent of any particular language. Whenever language-specific information or actions are required, the GRT makes an *upcall* to code supplied for the particular language; in some cases this code will be specific to the class of object being manipulated. The GRT has been designed specifically to interface to a LSRT rather than to provide an API for application programmers to use directly. This was dictated by the goal of supporting the use of high level languages to program distributed and persistent applications.

To support a language above the GRT only requires the provision of the necessary upcalls and the production of additional code used to intercept accesses to certain objects. The upcalls can be generated on a per-class basis by a preprocessor – as in our C++ implementation – or implemented once in the LSRT if sufficient type-information is available – as in our Eiffel implementation. Naturally, some changes are also required to the LSRT to interface to the GRT.

The GRT provides the language implementer with a choice of mechanisms for supporting persistence, distribution and atomicity. For example, the GRT supports the use of different mechanisms to detect attempts to access objects which are not currently collocated with the object accessing them. As another example, the GRT will support eager, lazy or no swizzling of references - the choice of which to use in a particular language depending on the intended use of the language. Of course the GRT cannot support all possible means of providing some particular functionality. A language implementer may provide additional mechanisms within the LSRT of a language. Similarly, the GRT may not support the use of some language constructs. In this case the language implementer may either restrict the use of these constructs in the extended language [6] or provide additional support for them in the LSRT. In effect, there is a tradeoff between the amount of work to be done in the LSRT and the range of mechanisms available to a language implementer. The GRT is intended to provide a range of alternatives to cater for most requirements.

The GRT is supported by the Amadeus *kernel* [5]. While the GRT is a purely local component, linked into the address space(s) of each application, the kernel is a distributed component providing secondary storage management, location services, security mechanisms, transaction management, distributed processes, and load balancing. In some sense, the GRT may be seen as the interface between a LSRT and the underlying kernel. The kernel is currently implemented above UNIX[1] as a collection of trusted servers with an associated library which is linked with each application.

We believe that our approach has significant advantages over other approaches. First, adapting a language is easy. The major effort is in providing the upcalls and auxiliary code required by the GRT and in interfacing the LSRT and GRT.

Second, the GRT (together with the underlying Amadeus kernel) provides all the necessary support

[1]UNIX is a trademark of UNIX Systems Laboratories, Inc.

for persistence, distribution and atomicity. This support is immediately available to a language implementer via the GRT and does not have to be provided anew for each language.

Finally, given a common underlying support system, language interworking is facilitated, although not completely solved – additional mechanisms are still required at higher levels to, for example, support inter-language type checking.

1.1 Roadmap

The remainder of this paper is concerned with the GRT support for persistence and distribution. The support for atomicity is described elsewhere [17]. We proceed as follows. Section 2 discusses related work. Section 3 gives an overview of the functionality required to support remotely accessible and persistent objects, and suggests how this can be divided between the GRT and LSRTs. Section 4 describes the functionality provided by the GRT and section 5 describes its current implementation. Section 6 describes the interface between the GRT and a LSRT. In section 7 we provide a critique of the current GRT and discuss those aspects of it that will be extended to support a wider range of options for the language implementer. Finally, section 8 concludes the paper with a summary of our achievements and a description of on-going and future work.

2 Related Work

In contrast with other systems, our aim is to provide a generic layer which can be used to support persistence and distribution in whatever way a language requires.

Many systems support one only approach to persistence and/or distribution and implement this on top of the basic operating system interface. For example, although the Gemstone OODBMS [2] supports two languages, Opal and C++, it in fact only supports one approach to the implementation of persistence. Opal is the basic language and C++ is layered on top of it: a C++ class definition is translated into an Opal type definition.

The Object Management Group's (OMG) Common Object Request Broker Architecture (CORBA) [8] also aims to support the use of different languages in a distributed system. However, using CORBA requires that the application programmer writes an Interface Definition Language (IDL) description of each type of object to be used by distributed clients. Clients can then invoke objects of these types from each supported language, in a manner defined by CORBA for each language. Currently, C is the only language considered in [8]. Moreover, persistence and atomicity are not currently considered.

Our approach to the provision of a generic layer has been influenced by the Portable Common Runtime (PCR) [15] which provides interoperability between programming languages, to the extent that an application can use different languages within a single address space without having one of the languages play a dominant role. Those features required for interoperability are provided in common by PCR: space allocation and garbage collection, basic I/O, threads, and symbol and code binding. We share the goal of language interoperability, but to this we add the aims of adding persistence and distribution to existing and new languages with the minimum of work required above the generic layer. Points in common with PCR include the use of upcalls to the individual language levels in order to locate references.

Amadeus is the reference implementation of the Comandos architecture [3]. As such, the goals of Amadeus are shared by the other implementations of the Comandos architecture, notably IK [22] and COOL [13].

3 Requirements

This section addresses the separation of functionality between LSRTs and the GRT. There are important tradeoffs here: the GRT should provide as much of the support for persistence, distribution and atomicity as possible so that it is easy to add these facilities to an existing language, however, each language implementer must be able to choose how best to use this support.

This separation of functionality will be discussed under the following topics:

- the naming of objects within an address space, on a remote node or on disk, and the transmission of object names;
- the layout of objects in memory and on disk;
- the mapping of objects into an address space, and the management of object references when this occurs;
- the binding of code to an object when it is mapped into an address space;
- detection and handling of *object faults*: that is, attempts to access *absent objects* which are not located in the current address space;
- marshalling and dispatching of invocation requests to absent objects.

Object naming and layout: It is clear that unless the language implementer is prepared to substantially re-implement the compilation system and associated LSRT, it must be possible for the language to retain its own naming scheme for objects within an address space. Most languages use virtual memory addresses to reference objects; other formats, such as indices into a table, are also possible. For the same reason, the layout of objects must continue to be dictated by the compilation system.

Hence, the GRT should view objects simply as blocks of storage which can be uniquely identified by a language and on which local invocations can be made transparently to the GRT. However, if an object is to be persistent or remotely accessible then it must be known to the GRT and hence the GRT must be notified of the creation of such objects.

Since use of language-specific references to name remote or stored objects will not always be sufficient, the GRT must provide a location independent naming mechanism suitable for uniquely identifying every object in the system. Such references – known as *global references* – can be used (directly or indirectly) by objects which are on disk or are being transmitted over the network, and when references are passed as parameters in remote requests.

Mapping objects and reference management: The GRT must provide the underlying means of storing, mapping and unmapping objects. The fact that objects may use different reference formats when on disk and when in an address space, requires that references be *swizzled* from one from to the other when an object is mapped in and *unswizzled* when it is mapped out. However, since the LSRT is responsible for the format and manipulation of language level references, and also their locations within objects, the GRT cannot do this swizzling and unswizzling without the cooperation of the LSRT. At a minimum, the LSRT must provide information about the location of references. Additionally, given a global reference, it must be possible to translate that reference to an appropriate language-specific reference (whether or not the target object is mapped into the current address space). This translation involves cooperation between the GRT and the LSRT.

Swizzling can be performed in two manners (lazy and eager) for an object and the GRT must not dictate which is appropriate for a given language implementation. Lazy swizzling involves swizzling each individual reference as it is used for the first time after the object is mapped in; eager swizzling means swizzling all of the references when the object is mapped in or is first used.

In fact, when a language is implemented specifically to support persistence and/or distribution, it may use the same reference format when an object is on disk and when it is mapped in. The GRT will have to support the two ways in which this can be achieved. Firstly, global references can be used in both cases [18]. The disadvantage of this approach is that references must be looked up in a hash table on use; its advantage is that objects can be mapped and unmapped without swizzling and the target of an object reference can be unmapped from memory without affecting the object. The alternative to this mechanism is to use virtual memory addresses as global references within the system [12]. This raises difficulties for the global allocation of references, and also when supporting stores which are larger than the size of virtual memory.

Binding code to objects: Code must be associated with each object. Languages differ considerably in how they make this association: for example, many implementations of C++ store one or more virtual addresses in each object to point to the object's *virtual function table(s)*; some implementations of Eiffel use the object's type code to index into a code table. Therefore, the binding of code to a recently mapped object must be performed by the LSRT when the GRT maps in an object.

Object fault detection: There are several possible mechanisms to detect attempts to access absent objects and hence the GRT must support a range of these mechanisms which can then be used at the language level.

Example mechanisms include the use of *proxy* objects which are local representatives of absent objects. A proxy knows the identity of the object that it represents and can inform the GRT of attempts to invoke that object. The GRT may then use the services of the kernel to locate the object and forward the invocation request to it; or, the target object can be mapped into the caller's address space, replacing the proxy. This mechanism provides a high degree of transparency to the language level – it need only provide the code for the proxy and then all language level operation calls can be local. However, it does not allow local access to public member data within the absent object. An alternative mechanism which does allow this is to ensure that use of a language-specific reference for an absent object causes a virtual memory fault. This can be caught and the absent object mapped into the address space.

These GRT level mechanisms can be supplemented by other mechanisms implemented entirely at the language level, such as presence tests (the explicit testing of whether the target object is present before using a reference), and the use of GRT mechanisms to map in the object or forward the invocation to it if it is absent.

Object location and remote invocations: The GRT must also provide an interface to the kernel's facilities for locating objects and transferring remote access requests to the target objects. The formatting of a request must however be carried out at the language level. The GRT should provide routines to marshal object references and values of basic types (that is, to allow construction of a contiguous message from the parameters to a request). In the case of object references, the translation of the reference to the appropriate global form can be carried out by the GRT, but this requires cooperation with the language level. Similar comments apply to migration of objects. On the remote side, the language must be prepared to accept incoming requests from the GRT, to unpack the parameters, to dispatch the request in the language-specific manner and, once the request has been completed, to format the reply message. The dispatching of the request involves choosing the correct method to call. This must be carried out by the LSRT since only it understands the operation invocation mechanism to be used.

Finally, the GRT can also perform local garbage collection, but it must use the LSRT to determine the references each object holds.

In general, it can be seen that the GRT is responsible for providing the underlying mechanisms, for example the mapping and unmapping of objects, but it must cooperate with the LSRT to carry out some actions. This cooperation takes the form of

- *upcalls* from the GRT to the LSRT when the GRT requires the LSRT to perform some action or provide some information that is language implementation specific,
- *downcalls* from the LSRT to the GRT when the LSRT needs to use the GRT's mechanisms.

4 Functionality of the GRT

In summary, the GRT provides support for:

- object creation and naming;
- detection of object faults;
- mapping and unmapping of objects;
- marshalling, unmarshalling and dispatching of invocation requests;
- clustering of objects; and
- local garbage collection.

Each of these aspects is briefly described in the following sections. The GRT also provides support for the use of atomic objects and for object access control, as well as providing the interface to the facilities provided by the underlying kernel including processes,[2] atomic transactions, extents,[3] and containers.[4] These aspects are not, however, considered in this paper.

It should be noted that the fundamental unit of distribution and storage supported by the underlying kernel is a cluster of (related) objects rather than a single object. Clusters are the units of mapping and unmapping into and out of an address space and every object is associated with exactly one cluster. Objects may, of course, migrate between clusters during their lifetime.

4.1 Objects

Fundamentally, the GRT is concerned with the management of objects that are remotely accessible (known as *global objects*) and/or persistent. From the GRT's point of view, such an object is an opaque element of storage which can be uniquely identified and to which code implementing the interface to the object can be bound dynamically. The GRT does not know anything *a priori* about the internal structure of a particular object nor about the semantics implemented by that object. Such information can be acquired by the GRT by making upcalls to the language level.

A global or persistent language object – such as, for example, an individual C++ object – must be mapped, in a way specific to its programming language, onto a GRT object. The most natural mapping is a single GRT object for each heap allocated language object. Other mappings are possible. For example, in C++, a language object may be embedded within another language object which is, in turn, mapped onto a single GRT object. In the remainder of this paper the term *object* is used as an abbreviation for GRT object, and programming language objects are always qualified as *language objects*.

New global or persistent objects are created by explicitly calling the GRT. The GRT allocates space for the object (including space for a GRT header for the object) in the heap of the current process and returns the address of the object to the language level.

When initially created by the GRT, an object is *immature*. Immature objects exist and are known only within the address space in which they were

[2] *activities* which are distributed threads of control and *jobs* which are collections of activities.

[3] collections of protected objects.

[4] secondary storage partitions

created. Objects may become known outside of the address space in which they were created (either because a reference to the object has been passed out of the address space, or because the object itself has migrated out of the address space). If this happens, the object is *promoted* to be a *mature* object and is given a global reference - its *pid* - which is sufficient to identify the object anywhere in the system.

An object's GRT header stores information used by the GRT to manage the object and to link the object to the code implementing the language-specific upcall operations required by the GRT. In normal operation an object's GRT header is transparent to the language level.

4.2 Object References and Object Fault Detection

On disk, a persistent object is stored with its GRT header followed immediately by the object's data (possibly containing references to other objects) which is followed by a set of *indicators* for the objects that it references.

An *indicator* is an object reference which holds enough information not only to identify and locate the referenced object (i.e. its *pid*) but also to create a *proxy* for the object if required.

The layout of an object's data is controlled by the language level and, in general, the space allocated by the language for a language-specific reference – an *lsref* – is too small to store a full *indicator*. Hence, when on disk each reference within an object's data is stored as an offset either to the referenced object, if it is stored in the same cluster, or, in the case of an inter-cluster reference, to an *indicator* for the referenced object. This allows the layout of the object's data to be the same as it would be for a "normal" version of the object. Such an offset is known as a *pptr*.

When two objects are collocated in the same address space, the format of references (*lsrefs*) between them is determined by their programming language. For example, two collocated C++ language objects can use virtual addresses between them for the duration of their coresidence. The GRT, with the aid of language-specific support, is responsible for swizzling *pptrs* to *lsrefs* and vice versa as necessary.

When an object is fetched into an address space, each of its *indicators* is located, by upcalling to the language level, and examined in turn. If an *indicator* refers to an object which is located in the same address space, the corresponding *pptr* will be replaced by a *lsref* for that object.[5] Subsequent dereferencing of (and invocation via) that object reference can then use the native language mechanism. For example, two collocated C++ objects will use the usual C++ invocation mechanism, and under these conditions will not use the services of the GRT.

If, on the contrary, an *indicator* refers to an object which is not (currently) located in the same address space, some mechanism must exist to trap attempts to access the absent object. Two mechanisms for this are currently supported by the GRT: O *proxies* and C *proxies*. Other mechanisms can be implemented at the language level.

An *O proxy* for an absent object is essentially a GRT object which contains no data but is the same size as the object it represents. The code bound to the *proxy*, which must be provided by the language level, must implement the same interface as the absent object and, when invoked, is responsible for calling the GRT to resolve the object fault having first marshaled the parameters to the invocation. O *proxies* are dynamically created by the GRT as required. Such *proxies* need never be stored, and they are invisible to application programmers.

[5] Note that the GRT employs lazy swizzling in so far as the references within an object are not swizzled when the object's cluster is mapped, rather this is delayed until such time as the object is actually accessed. However, eager swizzling is used within an object, because all of its references are swizzled at that time.

Further, an invocation via that reference will proceed as indicated above using the language mechanism. The *proxy* is responsible for reacting to the attempted invocation by calling the GRT to handle the object fault. If the object represented by a *proxy* is subsequently mapped into the same address space, the *proxy* can be *overlaid* by the GRT with the real object – in this way, any *lsrefs* to the object or to its internal parts from elsewhere in the address space remain valid. The GRT then forwards the invocation to the real object.

A *C proxy* represents an absent cluster, a reference to one of whose objects has been swizzled. A C *proxy* is a protected area of virtual memory large enough to hold the absent cluster should it be mapped into the address space. A swizzled reference to an absent object belonging to the cluster points at the location in memory where the object will be loaded when the cluster is mapped. An attempt to access the object will be caught by the GRT as a protection violation and it will attempt to map the cluster. When the cluster is mapped, the access can then be allowed to proceed as normal.

O *proxies* may be used to trap accesses to an absent object if the object can only be accessed through its operations. If client objects are allowed direct access to the instance data of an object, then additional mechanisms must be provided at the language level, an example being a locality test before accessing the object. In addition, this mechanism depends on the language level to provide the *proxy* code responsible for reporting the attempted access to the GRT. C *proxies* may be used to trap accesses to objects whether or not they are accessed directly or only through their operations. The main disadvantage is that having detected the object fault it is not usually possible for the GRT, in a language independent manner, to determine the cause of the object fault nor to marshall the parameters to the invocation (if any). Therefore such object faults are normally resolved by mapping the required cluster and restarting the access locally.

For these reasons, the GRT currently creates O *proxies* whenever swizzling a reference to an absent global (persistent or non-persistent) object, and a C *proxy* for the target object's cluster whenever swizzling a reference to an absent (non-global) persistent object.[6]

4.3 Dispatching and Marshalling

Dispatching of invocations is performed in a language-specific way. In the case of two collocated objects compiled by the same compiler, invocations need not use the services of the GRT and so can proceed directly as explained previously. For remote invocations, each (global) object must be prepared to receive an invocation request in a canonical (GRT-defined) format and dispatch it in the appropriate language-specific way to the requested operation. Likewise, the outcome, including abnormal or exceptional conditions (if any), must be returned to the GRT in a canonical format.

Marshalling of invocation frames is a responsibility of both O *proxies* (on the sending side) and of the dispatching mechanism (on the receiving side). The GRT provides a suite of marshalling routines which can be used by the *proxy* code to marshal the parameters and results of an invocation into standard message formats expected by the kernel including encoding and decoding of individual marshalled values into the canonical format used for transmission between heterogeneous nodes.

A side effect of marshalling a reference to an immature object is to cause that object to be promoted to being globally visible. This is handled within the appropriate marshalling routine and is transparent to the O *proxy*.

4.4 Cluster Management

In practice, many applications do not choose to explicitly manage clusters, but rather employ the

[6] In the remainder of this paper the term *persistent object* is used to mean a persistent but non-global object.

default mechanisms of the GRT. By default, the GRT creates new clusters using a simple heuristic, and places each new object into the most recently created cluster. As well as the default mechanism, the GRT provides primitives to explicitly create a new cluster (into which subsequent new objects will be placed).

4.5 Garbage Collection and Cluster Unmapping

In Amadeus, garbage collection can be tackled at two levels. Mature objects may be known throughout the system, and thus distributed and secondary storage garbage collection is the responsibility of the kernel. Immature objects are known only within a single address space, and can therefore be garbage collected locally by the GRT.

The GRT currently includes a simple non-incremental mark and sweep garbage collector for immature objects. The roots for the collection are all the mature objects, and the stacks and machine registers of all the threads executing within the address space. All threads present in the address space are temporarily suspended during garbage collection.

A cluster is naturally unmapped from an address space when that address space is no longer required. Moreover, it is clear that in general, a cluster cannot be unmapped prior to the termination of the address space, unless there are no *lsrefs* referencing any of the objects within that cluster. To determine if there are *lsrefs* for an object, the GRT must scan the objects in the cluster, and the stacks of all activities executing in the address space. This is expensive, and so is coupled with the garbage collection of immature objects.

However, the GRT also provides a primitive to explicitly unmap a cluster from an address space. It is the responsibility of the caller to ensure that this primitive is used with care, since there may be outstanding invocation frames on the stacks of various threads in the address space which contain references to the cluster.

5 GRT Internals

This section gives an overview of the internals of the GRT and describes, in detail, the steps involved in mapping, unmapping and dispatching an invocation to a clustered object.

5.1 Address Space Layout

Each address space contains the following regions:

- A text area which contains a set of classes which have been statically linked with the mainline of some application and the GRT.
- A heap consisting of two distinct parts:
  - An area containing mapped clusters and C *proxies*. Clusters (and hence C *proxies*) always consist of an integral number of pages. The pages representing C *proxies* are protected.
  - An area containing (i) newly created objects, (ii) O *proxies* and (iii) objects which have been mapped as part of a cluster but have been copied out of the cluster to overlay an O *proxy*.
- A set of stacks for threads running in the address space.

5.2 Object and Cluster Layout

Each GRT object consists of a GRT header which is managed by the GRT, a data area which may be used to contain language objects, and a set of *indicators* for objects referenced from the data area.

The GRT header contains the following fields:

- *uc*: a pointer to the upcall function table for the object;
- *magic*: a magic number – a value to search for when looking for an object header;

- *indicator*: an *indicator* for the object itself containing the following fields:
  - *state*: encodes the state of the object;
  - *size*: of the object's data area;
  - *loffset*: used in *indicators* referencing embedded objects and in the header of a global object which has overlaid its O *proxy*;
  - *pid*: the unique identifier of the object;
  - *cluid*: the cluster in which the object is contained;
  - *cloffset*: the offset to the start of the object in its cluster;
  - *clsize*: the size of the target object's cluster;
  - *lid*: identifier of the language in which the object is programmed;
  - *iid*: identifier of the interface (i.e. abstract type) implemented by the object;
- *cid*: the identifier for the object's class;
- *vid*: the position of the object's entry in the GRT's Object Table;
- *link*: a pointer to the next object in the same cluster.

The *uc* field points to the upcall function table for the object and will differ depending on whether the header belongs to a real object or an O *proxy*.

The *state* field specifies the kind of object (i.e. global or persistent) and its state (one of *dormant*, *reg*, *inactive* or *active*, as described in section 5.4). This field is also used to distinguish real objects from O *proxies*.

An *indicator* for an embedded object is the same as that for its enclosing object except that the *loffset* field is used to give the offset of the embedded object within its enclosing object. In the case of a global object that has overlaid its *proxy*, the *loffset* field is used to store the original virtual memory address of the object in its cluster.

Only mature objects have *pids*. *pids* are allocated by the kernel and used (together with the object's *cluid*) to locate the object when required.

Note that the GRT distinguishes between the type of an object as defined by its *iid* and its class as defined by its *cid*. Any type can be implemented by several different classes. In particular, the *iid* for an object and for an O *proxy* for the object are the same, while the *cids* are different.

Within each address space, every object (whether immature or mature) has an entry in the local Object Table (see section 5.3) managed by the GRT. An object's *vid* gives the position of the object's entry in the OT.

When stored in the storage system the *state*, *vid* and *uc* fields of an object's header are not used and *lsrefs* in its data area are converted to *pptrs*.

The layout of objects and clusters in virtual memory and on disk is shown in Fig. 1. Note that some objects may be copied out of their cluster to overlay their O *proxies* in the same address space.

5.3 GRT Data Structures

Each address space also contains a number of tables that are used by the GRT to keep track of the clusters and objects which are mapped in the address space:

- The Object Table (OT) is used to keep track of the objects that are mapped into the address space. Each OT entry contains a pointer to the start of the corresponding object's GRT header.
- The Cluster Register (ClR) which lists the clusters which are mapped into the address space.
- The Cluster Map (ClM) which lists the clusters for which C *proxies* exist in the address space. Given a pointer to any address within a C *proxy*, the ClM is used to identify the corresponding cluster.

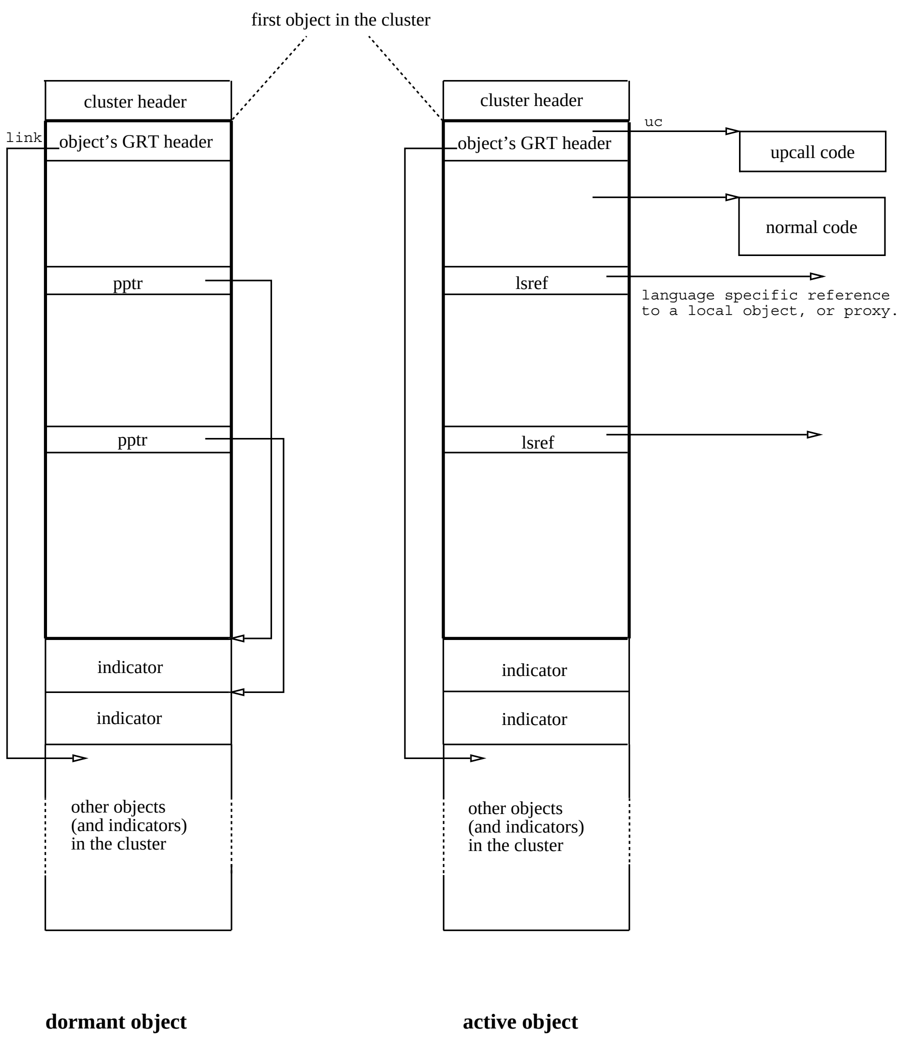

Figure 1: The structure of a cluster

5.4 Mapping an Object

When stored in the storage system, every object is in the *dormant* state.

A cluster is normally mapped into an address space when an attempt is made to access one of its objects. On being mapped into an address space, all global objects in the cluster will be registered in the OT as being present in the address space. Some of these objects may already be represented by O *proxies* in the address space and others not. Any objects for which an O *proxy* does not already exist in the address space will enter the *reg* state. In this state, no code is linked with the object.

If a *pptr* that references a global object in the *reg* state is swizzled, then that object is converted to the *inactive* state. To do this, code which is capable of responding to invocations on the object must be associated with the object; for example, in the case of a C++ object, any virtual function table pointer(s) must be set up. Note that this code need not be the real code for the object's class. When the object is first invoked it must be converted to the *active* state – with the associated binding of the real code for its class to the object and the swizzling of its *pptrs* to *lsrefs*.

The *inactive* state was introduced so that a global object can be made *active* without making the global objects that it references *active*. So doing would require more and more objects to be made *active* when a large and complex structure was accessed. One way to avoid this would be to test whether the target object is *active* or not when dereferencing a *lsref*. Requiring this would not allow the operation invocation mechanisms of conventional languages to be retained. Hence, the *inactive* state was introduced.

The code associated with an *inactive* global object must be able to trap the first invocation and make the object *active*. This can be done either by associating special code with the object in the *inactive* state, and changing this to the *active* code when the object is made *active*. Alternatively, code shared by both states could have a test at the start of each operation, using the *state* field in the object's header to determine whether it is already *active* or not. Obviously the latter approach, not only incurs an overhead on each invocation, but also implies that the language's invocation mechanism is changed.

When an object is made *active*, its references are swizzled. Some of its *pptrs* may reference (through their *indicators*) objects not currently mapped into the address space. In this case, either an O or C *proxy* for each such object is created, unless one already exists. In the case of an O *proxy*, code must be bound to the *proxy* to respond to attempts to invoke the real object. The information required to construct the O or C *proxy* can be found from the *indicator* referenced by the *pptr* being swizzled. The swizzled *pptr* becomes an *lsref* to the *proxy*.

When an O *proxy* is invoked for the first time, its code passes the invocation to the GRT (the *pid*, etc, of the required object can be extracted from the *proxy*'s header). The GRT can decide either to make a remote invocation or to map the invoked object's cluster into the caller's address space. In the former case, the GRT makes the invocation and returns the result to the *proxy*. In the latter case, the GRT will map the cluster in, overlay the *proxy*, make the object *active* and pass the invocation to it. The *proxy* is overlaid so that the *lsrefs* already referencing it will remain valid. Each of the other global objects in the cluster either overlay their *proxies* and are made *inactive*, or if no *proxy* exists, are converted to the *reg* state. Note that a global object cannot be mapped into an address space and have a *proxy* in it.

The first access to a C *proxy* results in a protection violation which is trapped by the GRT. The GRT can then obtain the identifier of the target cluster by looking up the faulting address in the ClM. In this case the GRT will request the kernel to map the required cluster into the space allocated to it in the current address space, thus overlaying the C *proxy*. Each persistent object in the cluster then enters the *active* state immediately. The protection on the pages making up the cluster is then changed and the attempted access restarted.[7]

5.5 Unmapping Objects

To unmap a cluster, all of its objects must first be transformed into the *dormant* state. A number of steps may be required to convert an object from the *reg*, *proxy*, *inactive* or *active* states to the *dormant* state.

An *active* global object that has no outstanding invocations, and is not referenced by any object that has, can be converted to the *inactive* state. Outstanding invocations can be detected by looking at the stacks and machine registers of all of the threads in the address space. If the stacks and machine registers are viewed as being roots, then an *active* object that is not directly referenced from these roots can be made *inactive*. Further, any objects not directly or indirectly referenced by swizzled pointers from these roots can be converted to the *reg* state. If all objects in a cluster are in the *reg* state then they can be converted to the *dormant*

[7]Note that it is not necessary for all objects within the cluster to be made *active* but only those objects sharing the same page (or pages) as the object being accessed.

state, and the cluster unmapped (asynchronously). To increase efficiency, the GRT maintains a per-cluster count of the number of objects in the *active* or *inactive* states, as well as the total number of objects in each cluster.

In principle, a cluster containing only persistent objects can be unmapped at any time by changing the protection on the pages occupied by the cluster so that further accesses to the cluster will be trapped, that is, by effectively replacing the cluster with a C *proxy*. Each object in the cluster could then be transformed to the *dormant* state by the GRT in the background. In practice, such objects will not be unmapped until there are no swizzled references to the cluster anywhere in the address space. Since this in general requires a scan of the entire address space, it is combined with local garbage collection.

On transforming an object from the *active* state to one of the *inactive*, *reg* or *dormant* states, the references in the object must be unswizzled. This is done by using language-specific code to locate the *lsrefs* it holds. Where *lsrefs* are not virtual addresses, the address of the object (or *proxy*) referenced by each *lsref* must be determined. Each *lsref* needs to be unswizzled to a *pptr* with the GRT filling in the *indicator* allocated to the reference. The *indicator*'s information can be found by locating the target object's (or *proxy*'s) GRT header, but because of the idiosyncrasies of different languages, the address found by translating an *lsref* may not point to a fixed offset from its target object's GRT header. Therefore, the GRT searches backwards for the *magic* pattern, locates the object's *vid*, and checks if this is valid using the OT. If the magic pattern appears by coincidence in the data of an object, then the OT entry corresponding to the value taken to be the *vid* will show that the GRT header has not yet been found (because the current search location will not equal the contents of the OT entry), and the backward search will continue.

The *loffset* field of the *indicator* is set to the difference between the address corresponding to the *lsref* and the address of the GRT header. When the reference is swizzled later, the *loffset* is added to the address of the GRT header to give the address from which the language constructs its *lsref*.

5.6 Code Management by the GRT

Each global object has one of three different normal code blocks associated with it at any time:

- the real or *active* code for the object,
- the *inactive* code, or
- the *proxy* code,

and three different *upcall code blocks* (managed and called by the GRT):

- the *active* upcall code,
- the *inactive* upcall code, or
- the *proxy* upcall code.

Each global object in the *active*, *inactive* and *proxy* states has the corresponding normal and upcall code bound to it. The normal code is bound in a language-specific manner; the upcall code is bound by the GRT – using a pointer in an object's GRT header (much like a C++ virtual function table pointer).[8]

Only the *active* code performs the object's proper actions. The *proxy* code reports attempts to invoke the object to the GRT, and the *inactive* code merely waits for the first invocation on the object so that it can cause the object to be made *active*.

Three normal and three upcalls code blocks are not normally needed. In fact, it is possible to have only one normal code and one upcall block per-implementation, since a single version of these could use the object's state to determine their actions. However, the normal *active* code will usually be separate, so that operations can run without

[8]In fact, the GRT is implemented in C++ and it views object headers as C++ objects with virtual functions which implement the upcalls.

testing the object's state (see section 5.4). Similarly, the *active* upcall code will be separate. The normal *inactive* and *proxy* code will usually be the same code block. The *inactive* and *proxy* upcall code will normally be the same also. For persistent objects, only the *active* code and *active* upcall code are required.

The combination of a normal code block and its associated upcall code block defines a class. Each class is identified by its *cid* and is described by a GRT data structure known as a *class descriptor*. The set of all class descriptors for all known classes is maintained in the *Class Register* in the storage system. The GRT provides an interface to allow a class descriptor for each newly defined class to be registered by the LSRT.

Each class descriptor contains the (string) name of the class, its *cid*, and the size of instances of the class, as well as a reference to the upcall function table for the class.

The GRT maintains a cache of class descriptors in each address space. In memory, each class descriptor is stored in the data of a GRT object which is referenced by the Class Register Table – a hash table indexed on *cid*. If a class descriptor is not present in the Class Register Table then the GRT must search the Class Register using the *lid* and *cid*.

The primary use of the class descriptor is to locate the upcall code for an O *proxy* or object. For example, when creating a new object, the class descriptor of the normal class is used to locate the upcalls for the object; likewise when creating an O *proxy*, the *proxy* class descriptor is used.

6 The GRT Interface

The downcalls provided by the GRT fall into three main categories:

- downcalls for the management of objects, including object creation, trapping of object faults and marshalling of parameter frames;
- downcalls concerned with cluster management;
- and downcalls related to the abstractions implemented by the underlying kernel which are accessed via the GRT.

In the following, only the downcalls concerned with management of persistent and global objects are described:

create is used by a language to create new persistent and/or global objects. This call requires as parameters the size of the object and its *lid*, *iid* and *cid*. The *cid* can be used to find the upcall code for the object and hence to bind the normal code to the object in a language-specific manner.

resolve is typically called by the *proxy* code to check if the associated object is in the *proxy* or *inactive* states and, in the latter case, to cause the object to be made *active*;

tblock is called to open a buffer into which the parameters to a remote invocation can be marshalled. This call takes as parameters the size of the parameter frame and the number of references in the frame. This routine is typically used by a *proxy* routine.

push is called to marshall a parameter into a buffer opened with *tblock* or a result into a buffer received in a remote request. In fact, *push* is an overloaded operation – different versions exist for basic types and for references. Pushing a reference will result in an *indicator* being produced from the reference and being marshalled and may result in the corresponding object being promoted. Note however that only a reference to a global object can be marshalled in this way. All versions of *push* take care of formatting the value for transmission to heterogeneous nodes.

pop (also an overloaded operation) is used to retrieve a parameter or result from a received parameter frame. Popping an object reference may result in an O *proxy* for the object being created if necessary.

rpc calls the GRT with a parameter frame describing an attempted invocation and asks the GRT to handle the implied object fault. This may result in a remote invocation or in the mapping of the target object. *rpc* is typically called from a *proxy* routine.

For each object, an upcall may be class-specific or it may be generic if there is enough runtime type information available for the class in the LSRT. The upcalls, and comments on how they may be implemented, are as follows:

onuse requests the object to perform language-specific binding of its code. The upcall code has to hold the identity of the code to be bound; this must be built into it at compile time or link time. In our C++ implementation, *onuse* sets up the virtual function table pointers. The Eiffel runtime binds code to a class by using a class identifier (which is known as the Dynamic Type of the class and is stored in the Eiffel-level object header) to index into a table of pointers to per-class function tables. In our Eiffel implementation, *onuse* changes the dynamic type of the object (e.g., from that for the *inactive*/*proxy* code to that for the *active* code).

activate is called when an object is being transformed from the *inactive* or *proxy* state to the *active* state. It can, and will normally, change the *cid* field in the object's header to indicate that the object in the *active* state should run different code. This is normally its only action; however, a language may allow user written code to be executed as well.

deactivate is called when an object is being transformed from the *active* to the *inactive* state. It can, and will normally, change the *cid* field in the object's header to indicate that the object in the *inactive* state should run different code. This is normally its only action; however, a language may allow user written code to be executed as well.

nextptr is called so that the GRT can find the locations of references within the object, so that they can be swizzled (going from *inactive* or *proxy* to *active*), or unswizzled (going from *active* to *inactive*). It is also called by the GRT garbage collector. In our C++ implementation, *nextptr* is generated from the class definitions. In our Eiffel implementation, *nextptr* is not class-specific: the implementation accesses the type information maintained by the Eiffel runtime.

remote_dispatch is called when the GRT receives a remote invocation for one of the object's operations. The GRT cannot directly call the operation because it does not understand the calling mechanism, so it calls *remote_dispatch*, which calls the operation. *remote_dispatch* must understand the format of the parameters passed from the *proxy* sending it the invocation. In our C++ and Eiffel implementations, class-specific *remote_dispatch* routines are generated; each uses a case statement with an operation identifier as the selector. Each arm of the case statement makes a language-level operation call.

The binding of upcall code is managed by the GRT, and is invisible at the language level. The binding of normal code to the object is language dependent; the GRT is not concerned about how this is done, but it informs the LSRT, via the *onuse* upcall, when to do the binding.

7 Critique

In this section we review some possible improvements to the current GRT design and implementation. We begin by discussing improvements aimed at better support for conventional languages and then consider improvements aimed at supporting a wider range of languages - particularly those already implemented for persistence or distribution.

7.1 Conventional Languages

The current version of the GRT was designed primarily to aid the introduction of persistence, distribution and atomicity into languages whose base versions do not support these facilities. While we believe that our goals in this respect have been largely achieved, as testified by the addition of persistence, distribution and atomicity to C++ and Eiffel, we are aware of a number of possible improvements to the current design.

Firstly, *indicators* are large – thus increasing the size of objects and therefore the cost of mapping and unmapping them. Their size arises in part because *indicators* simultaneously support two different approaches to object fault detection: use of O and C *proxies*. Both mechanisms require the *pid* and *cluid* fields in *indicators*. O *proxies* also require the *iid* field because O *proxies* are type-specific; the *size* field because they can be overlaid by real objects; and the *lid* field because *iids* are unique only within a language. C *proxies* require the *clsize* and *cloffset* fields. Naturally, the easiest improvement is to overlay mutually exclusive fields. Other improvements in this area will be discussed in section 7.2 below.

Secondly, the OT will have to be extended to allow for languages whose *lsrefs* are not memory addresses or GRT *pids*, for example languages which use indices into an object table as *lsrefs*. This is straightforward because the GRT only needs to store these *lsrefs* – it does not need to use them.

Thirdly, the *nextptr* upcall needs to be changed to more efficiently handle sequences of references.

The C *proxy* mechanism for object faulting can be used to allow direct access by client objects to the data of an object, however a resulting object fault is handled by trying to map the cluster into the client's address space or by raising an exception if this cannot be done immediately. In future versions, the lock manager will be used to delay conflicting (legal) requests. A further step would be to allow a cluster to be mapped into multiple address spaces using DSM techniques [14] to maintain consistency of the distinct copies.

Finally, the interface between the LSRT and GRT may be extended so that the LSRT provides more type information to the GRT for each object, either through a standard interface or by extending class descriptors to describe the layout of instances of each class. Some languages, Eiffel for example, already maintain this information. Some of the class-specific upcalls can then be replaced with generic code which interprets an object's type-information. The *nextptr* upcall is the obvious example. This will make it easier to port a language to the GRT. Since more work will be done by the GRT, the language implementer will only have to provide class-specific type information which, as noted, is already available for several languages. There are also other advantages in having such information available, for example, to allow low-level language-independent class browsers to be written.

7.2 Persistent or Distributed Languages

The initial aim of supporting the addition of persistence and distribution to languages whose native versions do not support these properties has had a number of effects on the design of the GRT, not least that it supports eager swizzling within an object, which ensures that pointers are swizzled into *lsrefs* before the *active* code of the object is run. The GRT currently cannot support lazy swizzling or no swizzling. In fact, the changes required to support these are not major; for example, when being activated the *nextptr* upcall could indicate that an object has no references, and then the LSRT could choose whether or not to swizzle its references as it uses them. Nevertheless, this would require a number of small changes to the GRT: for example, distinguishing between the different occasions when *nextptr* is called so that an object which does lazy swizzling can still make its references visible to the garbage collector.

There is also a need for increased flexibility in the forms of references allowed in non-*active* objects. Languages that use O *proxies* and lazy swizzling will not need the *size* field in *indicators*. Also, the use of offsets (*pptrs*) to *indicators* will not always be appropriate; hence in-line references will be supported for objects on disk.

In addition, for applications that access large volumes of data, it is desirable to allow clusters to be unmapped without requiring an address space scan. This is possible if a language's operation invocation mechanism tests for the presence of an object before invoking on it, and if active invocations on an object are easily detected. For example, if no swizzling is performed and if calls to *pin* and *unpin* routines [19] surround each invocation (or sequence of invocations) on the objects in a cluster, then that cluster can be unmapped when its *pin-count* is zero. We refer to such clusters as *non-anchored clusters*. The GRT will be extended to allow a cluster to be marked as being *non-anchored* as it is mapped into an address space.

The object references in some persistent and distributed languages [12, 6] are always virtual memory addresses, those avoiding the need to do any swizzling. The GRT does not fully support this mechanism. One component of this is already in place, that is the allocation of virtual memory space to absent objects and the handling of resulting protection faults. However, this will need to be extended with global allocation of memory addresses as object identifiers, and issues such security and heterogeneity support will have to be examined again.

8 Conclusions

In this paper we have described the motivations and requirements for providing a *Generic Runtime* library which facilitates the simultaneous extension of multiple conventional OOPLs with support for persistence, distribution and atomicity without duplication of effort. We showed that such a GRT has been implemented which allows an OOPL's existing (local) object reference format and invocation mechanism to be retained so that its existing compiler need not be modified. The GRT does not impose any language syntax changes but leaves the need for, and the style of, any such changes as matters for the language designer. We have shown that, using this GRT, extending a language for use in a multi-user, heterogeneous distributed system only requires the provision of certain well-defined pieces of code. We claim that the use of a common GRT also facilitates language interworking.

The GRT has been used to add persistence, distribution and atomicity to C++ and Eiffel – to give C** and Eiffel**. The formats of *lsrefs* and the operation invocation mechanisms remain unchanged in both languages. C** introduces new keywords to allow programmers to control which classes should have persistent, global or atomic instances; in Eiffel** no syntax changes were introduced. These languages, together with the underlying support system, are being used to provide sophisticated distributed applications including an implementation of the DASSY CAD tool interface [10] supporting a collection of interworking CAD tools and programmed in C** and an interoperator for heterogeneous database systems.

Our current work is concerned with designing and implementing the improvements described in section 7. We are working on supporting a version of the E persistent programming language – a variant of C++ – and extending it for distribution. In doing so we will retain E's invocation mechanism which has been changed from that of C++.

Future work includes support for inter-language working (via *iproxies* which accept operation invocations in one language and forward them to another) and the provision of debugging tools.

References

[1] Agrawal, R. and Gehani, N.H. Rationale for the design of persistence and query processing facili-

ties in the database language O++. In *Proceedings of the Second International Workshop on Database Programming Languages*, pages 25–40, Gleneden Beach, Ore., June 1989. Morgan Kaufman.

[2] Butterworth, P., Otis, A. and Stein, J. The Gemstone object database management system. *Communications of the ACM*, 34(10):64–77, October 1991.

[3] Cahill, V., Balter, R., Rousset de Pina, X. and Harris N. (Eds.). *The COMANDOS Distributed Application Platform.* ESPRIT Research Reports Series. Springer-Verlag, To appear 1993.

[4] Cahill, V., Horn, C. and Starovic, G. Towards Generic Support for Distributed Information Systems. In *Proceedings of the International Workshop on Object-Orientation in Operating Systems*, pages 104–107, Palo Alto, CA, USA, 1991. IEEE.

[5] Cahill, V., Taylor, P., Starovic, G., Tangney, B. and O'Grady, D. Supporting the Amadeus Platform on UNIX. Technical Report TCD-CS-92-25, Dept. of Computer Science, Trinity College Dublin, July 1992.

[6] Chase, J.A., Amador, F.G., Lazowska, E.D., Levy, H.M. and Littlefield, R.J. The Amber System: Parallel Programming on a Network of Multiprocessors. In *Proceedings of the* 12th *Symposium on Operating Systems Principles*, pages 147–158. ACM, December 1989.

[7] Deux, O. et al. The O_2 system. *Communications of the ACM*, 34(10):35–48, October 1991.

[8] Digital Equipment Corporation, Hewlett-Packard Company,HyperDesk Corporation, NCR Corporation, Object Design, Inc., and SunSoft, Inc. *The Common Object Request Broker: Architecture and Specification. Revision 1.1.* Object Management Group and X/Open, 1991.

[9] Distributed Systems Group. C** Programmer's Guide (Amadeus v2.0). Technical Report TCD-CS-92-03, Dept. of Computer Science, Trinity College Dublin, February 1992.

[10] GMD. The DASSY Prototype. Technical Report Preliminary Version 0.5, GMD, June 1992.

[11] Horn, C. and Cahill, V. Supporting Distributed Applications in the Amadeus Environment. *Computer Communications*, 14(6):358–365, July/August 1991.

[12] Lamb, C., Landis, G., Orenstein, J., and Weinreb, D. The ObjectStore database system. *Communications of the ACM*, 34(10):50–63, October 1991.

[13] Lea, R. and Jacquemot, C. The COOL Architecture and Abstractions for Object-oriented Distributed Operating Systems. In *Proceedings of the* 5th*SIGOPS European Workshop*, Le Mont-Saint-Michel, France, 1992. ACM.

[14] Li K. and Hudak P. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[15] Weiser, M. et al. The portable common runtime approach to interoperability. In 12th *Symposium on Operating Systems Principles*, pages 114–122. ACM, December 1989.

[16] McHugh, C. and Cahill, V. Eiffel**: An Implementation of Eiffel on Amadeus, a Persistent, Distributed Object-Oriented Applications Support Environment. In *Proceedings of TOOLS Europe '93*, Paris, France, March 1993. To appear.

[17] Mock, M., Kroeger, R. and Cahill, V. Implementing Atomic Objects with the RelaX Transaction Facility. *Computing Systems*, 5(3), 1992.

[18] Richardson, J.E., Carey, M.J., DeWitt, D.J. and Schuh, D.T. Persistence in EXODUS. In *Proceedings of the Workshop on Persistent Object Systems: their design, implementation and use*, pages 96–113, Appin, Scotland, August 1987.

[19] Richardson, J.E. and Carey, M.J. Persistence in the E Language: Issues and Implementation. *Software - Practice and Experience*, 19(12):1115–1150, December 1989.

[20] Riveill, M. An Overview of the Guide Language. In *Proceedings of the* 2nd *Workshop on Objects in Large Distributed Applications*, Vancouver, Canada, 1992.

[21] Shapiro, M. et al. Persistence and migration for C++ objects. In *Proceedings of the European Conference on Object-Oriented Programming*, Nottingham, July 1989.

[22] Sousa, P., Zúquete, A., Sequeira, M., Guedes, P. and Alves Marques, J. IK-2 Implementation Report. Technical Report COMANDOS-INESC-TR-0040, INESC, 1993. To appear.

[23] Strom R.E. Hermes: An Integrated Language and System for Distributed Programming. In *Proceedings of the Workshop on Experimental Distributed Systems*, Huntsville, AL, October 1990.