Department of Computer Science
Trinity College
University of Dublin
Dublin, Ireland
{*Vinny.Cahill, Paddy.Nixon*}@*cs.tcd.ie*

Object Models for Distributed or Persistent Programming

V.J. Cahill, P.A. Nixon, F.A. Rabhi

October 9, 1998

Abstract

As use of object orientation for application development has increased, many researchers have investigated the design of object-based programming languages for the distributed and persistent programming. This paper concentrates on reviewing a number of object-oriented languages for distributed or persistent programming. In each case, the focus is on the object model supported and the mechanisms and policies employed in the implementation of distributed or persistent objects. In particular, each language reviewed has been chosen to illustrate a particular object model or implementation strategy.

Key words: Distributed Systems Implementation, Persistence, Distributed Programming, Object Orientation.

Introduction

In recent years, considerable interest in the design of parallel and distributed applications has emerged from various computer disciplines such as database systems, network systems, operating systems, and computer architecture as well as the traditional high performance community. A distributed application can be defined as "a system of several independent software components, cooperating towards a common purpose, where all the components will not all be co-located at a single site". Applications of this type represent a significant proportion of computer software today, including the following classes (identified by Bal et al. [11]):

- parallel and high performance applications;
- fault-tolerant applications and real-time systems;
- applications using functional specialisation such as a distributed operating system;

- inherently distributed applications such as email and World Wide Web applications.

These applications are intended for several different possible configurations, such as parallel architectures, embedded systems, networks of workstations communicating through a Local Area Network or simply a set of independent computers linked through the Internet. Experience of developing these applications has evolved from their respective disciplines and within each domain a set of design principles have been developed. However, differences in the development process between domain are now disappearing, mainly for the following reasons:

- the development process is increasingly being pushed at a higher abstraction level, which makes architectural considerations more and more irrelevant;
- there are many applications (e.g. meta-computations) which are suitable for both parallel and distributed platforms;
- advances in processor and communication technology have made the performance of distributed systems compete with those of parallel systems;
- there are now techniques and tools which can help common problems such as configuration management and load balancing.

The question is now whether it is possible to use more rigorous software engineering approaches with a set of methodologies and associated tools or languages in the development process of such applications. The main benefits of moving in this direction are:

- The programmer can focus on the structural and dynamic characteristics problem to be solved without consideration for the underlying hardware (except when this is part of the problem);
- The separation between problem and target architecture ensures portability of the system across a variety of platforms;
- There is a possibility for automatic code generation, that satisfies the stated requirements (e.g. real-time constraints, performance, fault-tolerance);
- Support for monitoring the run-time execution of the program, which is needed for assessing performance, debugging etc., can be provided within an integrated environment.

The other papers in this special issue focus on new and novel techniques for specifying and testing distributed applications. To complement these insights, this paper concentrates on reviewing a number of object-oriented languages for implementing distributed applications. The paper considers both the issues of distribution and persistence in parallel. In each case, the focus is on the object model supported and the mechanisms and policies employed in the implementation of distributed or persistent objects. In particular, each language reviewed has been chosen to illustrate a particular object model or implementation strategy. The review concentrates on the languages and how they are implemented since its main objective is to identify object models and implementation strategies that should be considered by an implementor of a distributed system. Section 1 looks at persistent programming languages while section 2 looks at languages for distributed programming.

1 Languages for Persistent Programming

The motivations for persistent programming languages – those that support the transparent use of data that outlives the execution of the program that created it – are by now well known and accepted. In the absence of a persistent programming language, it is usually necessary for any program data that is to survive program execution to be explicitly stored in a file system or database. Since the type, or perhaps more correctly, the data model supported by the storage system is usually different from that of the programming language, this requires the program to convert such data into the format expected by the storage system on storage and to reconvert it into the format normally used by the program on retrieval. Not only is the development of such code for every application likely to be tedious and error prone but the execution of this code is also likely to be computationally expensive. Moreover, it may be difficult to maintain pointer semantics in the face of such conversions. In addition, the type safety afforded by many programming languages may be lost when data is converted to some "external" format for storage. Persistent programming languages address these issues by making the storage and retrieval of data transparent to programs and by preserving both the relationships between values and their types [5].

A number of questions arise in considering the design of a persistent programming language. One such question concerns how persistent data is accessed and, in particular, whether access to persistent data is any different from access to non-persistent data. To be worthy of the label "persistent programming language", a language should allow access to persistent data to be the same as access to non-persistent data, I.E. the language should provide *access transparency*. This implies that the language should take care of automatically loading and storing persistent data as required. It does not, of course, preclude the provision of primitives to allow programmers to explicitly load and store data if required.

Another question concerns which types can have persistent values. Two approaches are commonly used. Persistence may be treated as an orthogonal property of all types so that values of any type may persist, or persistence may be regarded as a property of specific types and only values of those types may persist. The former approach is usually referred to as *orthogonal persistence* [5]. In the latter approach, transparency demands that the same type constructors be available for both persistent and non-persistent types. Of the languages reviewed later in this section, PS-algol, Texas/C++, and persistent Smalltalk provide orthogonal persistence while E and persistent Modula-3 distinguish between persistent and non-persistent types.

A further question concerns how values[1] that are to persist are identified (along with the related question of how persistent values that are no longer required are discarded). A common approach is to use *reachability* from some well-known persistent root as the criterion for determining whether a particular value persists. In this case, values of persistent types are said to exhibit the property of *potential persistence* in that only those that are transitively reachable via references from the root actually persist while values that are not reachable from the root are considered garbage and can be discarded. Other approaches are also possible. In fact, this is one feature where persistent programming languages show considerable variation. For example, variables whose values are to persist could be explicitly labelled. Alternatively, all values that are allocated in some designated heap(s) could be retained. In each of these cases, values would need to be explicitly deleted when no longer required (leaving open the possibility of *dangling references* referring to values that have been deleted). Hybrid approaches are also possible. For example, it might be specified that only values allocated in some designated heap are candidates to persist and that only those reachable from some well-known root will actually persist. Of the languages reviewed below, PS-algol, persistent Smalltalk, and persistent Modula-3 retain heap allocated values that are reachable from well-known roots, Texas/C++ retains values of persistent types that are allocated in designated heaps, and E allows variables of persistent types whose values are to persist to be labelled and also retains values that are dynamically allocated in designated heaps.

Finally, most persistent programming languages provide some form of persistent name service that allows symbolic names to be associated with persistent values and therefore provides a way of naming values that is independent of the programs in which they are used. Most also provide some form of transaction mechanism supporting atomic update to persistent data in the presence of concurrent accesses and failures.

The remainder of this section describes a number of influential object-oriented persistent programming languages. In each case, the position of the language in the spectrum of persistent programming languages is identified and the key features of its implementation outlined. Many more persistent programming languages exist and new ones are being developed currently, for example the PJava project at the University of Glasgow

[1] Of persistent types.

1.1 PS-algol – Orthogonal Persistence, Object Faulting, and Swizzling

The PS-algol language [6, 5] was developed in a joint project between the Universities of Edinburgh and St. Andrews. PS-algol is a persistent extension of S-algol, a member of the Algol family of languages, which was developed at St. Andrews. Thus, although not strictly an object-oriented language, PS-algol is included here because it was highly influential having been among the first programming languages to offer orthogonal persistence. In PS-algol, values of any type may persist, with persistence of heap-allocated values being determined by their reachability from a set of persistent roots that are under program control. The unit of persistence is an S-algol *structure*[2], which is analogous to an object in an object-oriented language. The term "object" is used in the remainder of this section to refer to structures as indeed it was by the designers of PS-algol.

The language design achieved its designers' goal of requiring minimal extensions to the host language. The basic model provided to a PS-algol programmer is of a large persistent heap that is partitioned into a number of named *databases*. A program may open one or more databases and obtain pointers to their root objects through which it can then access further objects. A database may be open for reading or writing. The first time that a database is opened by a program, a transaction is started and routines are provided to commit or abort all the changes made by a program. Closing a database without committing aborts all uncommitted changes to objects belonging to the database. Databases also serve as the unit of concurrency control: a single database can be open for writing by one program or reading by multiple programs. A basic name service is also provided to allow symbolic names to be associated with objects. In fact, the root of each database is a name service directory. Thus, all named objects act as persistent roots.

The best known implementation of PS-algol is the Persistent Object Management System (POMS) described in [23], which is itself implemented in PS-algol. POMS uses tagged 32-bit references to refer to objects. Such a reference may contain either a *persistent identifier* (*pid*) or a *local object name* (*lon*). A reference containing a *pid* is distinguished from one containing a *lon* by having its tag bit (I.E. its most significant bit) set. When stored on disk an object is referred to by its *pid*, which provides enough information to locate the object on disk. When resident in memory, an object may be referred to by its *lon*, which is its memory address. An attempt by a program to dereference a *pid* is trapped by the PS-algol interpreter. The *pid* is looked up in a table known as the Persistent IDentifier to Local Address Map (PIDLAM), which has an entry for every resident object containing the mapping between the object's *pid* and its address, and vice versa. If an entry is found, the *pid* it overwritten with the corresponding address and the dereference proceeds. If the target object is not

[2] Or an array or string.

resident in memory, it is *mapped* (I.E. loaded from disk and a PIDLAM entry for it made) before proceeding as above. Newly mapped objects will typically contain the *pids* of other objects, which may later be dereferenced and converted to *lons*.

Handling attempts to access non-resident objects as they occur is known as *object faulting*. POMS *detects* object faults using the PIDLAM and *resolves* them by mapping the required objects. Converting references in resident objects from the form used to refer to stored objects to an alternative form optimised to refer to resident objects is known as *swizzling*. POMS uses *lazy swizzling* since a reference is not swizzled until the first time that an attempt to dereference it is made. One effect of lazy swizzling in POMS is that both *lons* and *pids* may be used to refer to resident objects with the latter being swizzled as soon as they are dereferenced.

Once references are swizzled they may be used to access objects directly. Thus, in general, if an object is to be *unmapped* (I.E. removed) from memory, any swizzled references referring to that object must be *unswizzled* so that future faults on the object can be detected. In order to avoid the expense of the address space scan that would be required to find such references in the absence of additional information, objects are only unmapped at program termination [24]. During unmapping the PIDLAM is used to unswizzle any swizzled references contained in objects being unmapped.

A few other features of POMS are worthy of mention. POMS clusters objects on disk according to their type[3] and loads a set of objects in every disk operation. The unit of storage in POMS is known as a *logical block* and a single logical block may contain objects of only one type. The transaction mechanism uses a shadow paging strategy to update the persistent store resulting in logical blocks being stored at different physical disk locations over time.

[15] describes an implementation of POMS for UNIX systems. This implementation uses the same object faulting strategy and approach to swizzling. However, in order to reduce the frequency of checking whether an object reference contains a *pid* or not, certain optimisations are used. For example, to ensure that the PS-algol interpreter will never encounter a *pid* in the stack frame of an executing procedure or block (and hence that interpreter instructions that operate on the local stack frame need never check for *pids*), all references that are placed on the stack are swizzled. At commit, only modified objects and the newly created objects to which they refer are written back to the persistent store. New objects are first *promoted* to being persistent by having a *pid* allocated to them. In general, new objects are placed in the same database as one of the objects that refers to them.

[24] describes a different implementation of PS-algol that uses *eager swizzling*, I.E. the conversion of *pids* to *lons* is done as soon as the containing object is mapped. In this implementation, *lons* are not addresses but the indices of

[3] Or size in the case of arrays and strings.

the corresponding objects' entries in the PIDLAM. Thus, accessing an object always involves an indirection via the PIDLAM. Moreover, the PIDLAM must not only contain entries for resident objects but also for non-resident objects to which resident objects contain (swizzled) references. The PIDLAM entries of non-resident objects are marked in order to allow object faults to be detected. Object unmapping is easier because references are indirect. When an object is unmapped it is only necessary to update its PIDLAM entry to allow future faults on the object to be detected.

1.2 E – Persistent Types, Object Fault Avoidance, and No Swizzling

The E persistent programming language [47] was developed as one part of the EXODUS extensible database management system project [19] at the University of Wisconsin at Madison. E, a persistent extension of C++, was originally conceived as a language in which database system code could be written but was developed as a general-purpose persistent programming language supported by the EXODUS storage manager [20]. E provides a number of other extensions to C++ apart from persistence; these include generic classes and iterators.

E does not support orthogonal persistence as in PS-algol. Instead, E introduces a new type hierarchy, which parallels that of C++, together with a new storage class (`persistent`) and a predefined generic type (`collection`). In E, database (db) types are types whose instances may optionally persist beyond the execution of the program that created them. Both volatile and persistent instances of any db type may however exist. E provides a collection of fundamental db types, such as `dbint` and `dbchar`, and db type constructors, such as `dbclass` and `dbstruct`, as well as pointers to db types and arrays of db types. Any type that can be defined in C++ can be analogously defined as a db type [47]. If a variable of a db type is declared to have the storage class `persistent`, the object bound to that variable will persist. Thus, the names of such variables act as handles for objects in the persistent store. An E program may also create persistent objects dynamically using the predefined generic db class `collection`. An object created within a persistent collection will exist for as long as the collection does unless it is explicitly deleted. Thus, `collection` objects act as persistent heaps in which objects may be allocated (and deallocated). Note that E does not support garbage collection and hence objects must be explicitly deleted when required.

The distinction between C++ types and db types was motivated by the desire to optimise access to objects that are known not to be persistent. In E, access to C++ objects incurs no overheads beyond those of normal C++ while access to an object of a db type (which may not necessarily be persistent) incurs additional overheads.

E uses *object fault avoidance* rather than object faulting, I.E. it ensures that persistent objects are always resident before any attempt to access them is made

[46]. In E, persistent objects must be *pinned* in the storage manager's buffers before they are accessed by a program. Pinning an object causes it to be mapped (if necessary) and ensures that the object will not be unmapped until explicitly *unpinned*. The compiler generates the code necessary to pin persistent objects before they are accessed and to unpin them when the access is complete. In fact, the EXODUS storage manager allows subsets of an object to be pinned so that it is not necessary to map all of a large object. Thus, the E compiler only pins those parts of an object that are actually needed.

References to db type objects are called DBREFs and contain the identifier of the target object (allocated by the storage manager) and an offset within the object[4]. The identifier of a persistent object includes its physical disk address. The offset allows embedded objects and arrays of persistent objects to be supported. Given a DBREF, the storage manager can be asked to pin the target object in its buffer pool and return an (indirect) pointer to the corresponding offset in the object. Since E does not employ swizzling, once an object is unpinned the storage manager can write the object to disk in order to free up buffer space. Thus, E may be said to trade off fast access to resident objects in favour of fast storage and retrieval of objects. References to resident persistent objects are always DBREFs and accessing such an object always requires that the object be pinned.

Early implementations of E used various compile-time optimisations to reduce the number of pin/unpin calls executed by an application as described in [45]. Later implementations of E are based on an interpreter, the E Persistent Virtual Machine (EPVM), that provides the interface between the compiler and the storage manager [48]. Accesses to persistent objects are interpreted by the EPVM, which maintains a cache of recently-accessed objects by pinning (and unpinning) objects in the storage manager's buffers. By having the EPVM keep objects pinned between accesses, the intention is to reduce the number of storage manager calls required. The interface to the EPVM provides the ability to read and write values of basic types (including DBREFs) from and to persistent objects as well as some reference manipulation operations. The EPVM pins the required objects in the buffer pool and unpins objects when the buffer pool becomes full using a clock-based least-recently-used approximation to choose the objects to be unpinned.

The EPVM also supports a limited form of reference swizzling that allows local variables, which contain references to instances of db types, to be swizzled to indirect pointers to objects that are pinned in the buffer pool. Since the EPVM still reserves the right to unpin any object at any time, this scheme requires that the EPVM can locate swizzled references on the stack and unswizzle them when the object to which they refer is unpinned. This is accomplished by allocating such local variables on a separate reference stack that can be scanned by the

[4]Not all objects of db types are persistent; the DBREFs for such objects contain the memory addresses of the objects.

EPVM to locate references to be unswizzled. Although a swizzled reference is still an indirect pointer to the object, this scheme represents a saving over the use of a full DBREF. Using this approach also allows the compiler to generate code to access an object in the buffer pool without using the EPVM but at the cost of a runtime check to see if the local reference variable to be used has been swizzled or not.

1.3 Texas/C++ – Object Faulting and Eager Swizzling

The Texas project at the University of Texas at Austin implemented a persistent extension of C++ supported by the Texas persistent store [50]. Texas/C++ provides orthogonal persistence, supports both transient and persistent objects, and provides a high degree of transparency for C++ programmers by imposing few restrictions on the use of normal C++ constructs with persistent objects. In Texas/C++, a persistent object is one that is allocated in a persistent heap as opposed to the normal volatile heap, the stack, or the static area. The Texas/C++ language interface overloads the C++ `new` operator to allow persistent objects to be allocated in a heap corresponding to a specified persistent store [44].

Transparency is achieved by using virtual memory protection fault trapping to detect object faults together with eager swizzling. In particular, the implementation uses a technique referred to as "pointer swizzling at page-fault time" to detect and resolve object faults [54].

In Texas, a persistent store is structured as a collection of *persistent pages*. References contained in resident objects are virtual addresses. References contained in objects stored in the persistent store are so-called *persistent addresses*, which are longer than virtual addresses. Pointer swizzling at page-fault time then operates as follows. When an object is mapped, any persistent addresses that it contains must be swizzled to the corresponding virtual addresses. In order to do this, a virtual memory page is reserved for the persistent page to which each persistent address refers and the page protected. The persistent address is then swizzled to a virtual address referring to the correct offset in that page. Any attempt to dereference the swizzled reference will now result in a protection fault. The fault handler will retrieve the corresponding persistent page from the persistent store and make it available in virtual memory. To do so requires that any further persistent addresses contained in objects stored on that page be swizzled to refer to the corresponding virtual addresses. This may result in the allocation of further protected virtual memory pages for any non-resident persistent pages to which such persistent addresses refer. Thus, application programs never see persistent addresses. Any references accessible to a program will have been swizzled to virtual addresses when they were retrieved from the persistent store. If the objects to which they refer are not resident, then these references will refer to protected pages.

Pointer swizzling at page-fault time requires that a persistent page fit into a virtual memory page once swizzled. It must also be possible to determine the offset that an object will occupy in a virtual memory page from its offset in a persistent page. In Texas, persistent and virtual memory pages are the same size and persistent addresses are twice the size of virtual addresses. When persistent addresses are swizzled to virtual addresses they only occupy half of the available field. Thus, objects occur at the same offsets in persistent and virtual pages. In addition, it must be possible to accurately identify the locations of references within objects. For this purpose, Texas uses type descriptors that describe the layout of each class and every object has a header that identifies its class. Handling of objects that span page boundaries requires some special care since the required pages must be adjacent in virtual memory. Thus, the language must support some means of identifying the boundaries of such objects – Texas/C++ uses a special table per persistent heap to keep track of such objects. Another potential problem is exhaustion of the available virtual memory space, perhaps due to a large number of protected virtual memory pages being allocated but never accessed. [54] describes some techniques for handling this problem although these were apparently not implemented since it was felt that this problem was not likely to be encountered by realistic applications.

Each persistent store is stored as a single file and persistent addresses are 64-bit file offsets. Translation of persistent to virtual addresses is accomplished by replacing the page number in the persistent address with the appropriate virtual memory page number using a translation table with one entry per page. Programs can use multiple persistent stores each with its own persistent heap. Multiple persistent heaps are interleaved in virtual memory avoiding static partitioning of the address space. Per-persistent-heap free lists are also maintained persistently. The heap manager component of Texas splits each page into equal-sized chunks (of different sizes depending on the page) and new objects are allocated in an appropriately-sized chunk. Since objects are "chunk-aligned" within pages, locating the objects stored on each page (E.G. during swizzling) is easy once the chunk size for the page is known. This can be determined by examining the first object on the page. As mentioned above, objects that are larger than a page, I.E. those that span page boundaries, are handled specially.

Finally, Texas also supports checkpointing of the contents of a persistent heap using a write-ahead logging scheme. A design based on a log-structured storage system has also been proposed – see [50] for details.

1.4 Mneme – Persistent Smalltalk and Modula-3

The Mneme project [41] at the University of Massachusetts had as its primary goal the investigation of techniques for integrating programming languages with database features in order to support cooperative data-intensive applications. In particular, the project was concerned with the implementation of efficient object-oriented persistent programming languages based on the Mneme object store [41]. A particular feature of the project was that it investigated a wide range of techniques for supporting persistent programming languages (including different swizzling strategies [42], different object fault detection strategies and the effects of various clustering strategies [31], and compiler optimisations for persistence [30]) in the context of adding persistence to two existing languages: Smalltalk [26] and Modula-3 [18]. These particular languages were chosen because they represented different ends of the compiled versus interpreted, and statically- versus dynamically-typed language spectra; Smalltalk is a dynamically-typed interpreted language while Modula-3 is a statically-typed compiled language.

The remainder of this section gives a brief overview of the main features of the Mneme object store and reviews the various implementations of persistent Smalltalk and persistent Modula-3 developed within the Mneme project.

The Mneme Object Store The Mneme object store – referred to simply as "Mneme" in the remainder of this section – was intended to support a variety of existing persistent and database programming languages and therefore to support a variety of type, inheritance, and invocation mechanisms. Thus, Mneme provides only a very primitive object model. Essentially, an object consists of a byte-vector and a number of 32-bit slots. Bytes are uninterpreted by Mneme while slots are essentially tagged object references, which may contain either a 28-bit Mneme object identifier or a 31-bit integer value. Mneme stores objects contiguously and preserves the relationships between them.

All objects reside in Mneme *files* and the object identifiers stored in an object always refer to objects in the same file. References to objects in other files are made via forwarder objects in the current file. In order to support large collections of objects, while keeping object identifiers short, Mneme distinguishes between client-visible and on-disk forms of object identifier. The former, known as *client identifiers*, are used by clients of Mneme to refer to objects during a *session* of interaction with Mneme, while the latter, known as *persistent identifiers*, are stored in objects on disk. Client identifiers are guaranteed to name an object uniquely only for the duration of a single session and Mneme provides routines to convert between client and persistent identifiers, and vice versa, when necessary.

Two different interfaces are provided to allow clients to access objects. First, a client can create a *handle* for an object causing that object to be loaded into Mneme's buffer pool (if necessary). Given a handle, an object can then be manipulated via a set of routines provided by Mneme until such time as the handle is subsequently destroyed. Thus, handles are explicitly created and destroyed by clients to delimit periods when an object is actively being accessed.

Alternatively, a client can pin an object in the buffer pool obtaining a pointer to it that allows the object to be manipulated directly. Pinning an object will cause the object to be loaded if it is not already resident. When the object is unpinned Mneme must be explicitly informed if the object has been modified (so that changes to the object can be saved). Objects in the buffer pool are stored in their on-disk format so that any persistent identifiers stored in such objects are visible to clients when the object is accessed directly.

Since objects are expected to be too small a unit of transfer between disk and memory, Mneme groups objects within a file into a number of *logical segments*. A number of logical segments are in turn grouped into a *physical segment* and physical segments act as the unit of transfer between Mneme storage servers and their clients.

To support extensible policies for clustering, prefetching, caching, buffering, concurrency control, recovery, and versioning, Mneme provides *pools*. Every physical segment (and hence object) is under the control of a pool and every pool has a management strategy associated with it that identifies a set of routines to be used to manage (I.E. make policy decisions for) that pool when required.

The Mneme design also provides support for transactions and recovery that is beyond the scope of this review; see [41] for further details.

Persistent Smalltalk A number of different implementations of persistent Smalltalk are described in the various Mneme papers – primarily [32] and [31]. All provide transparent orthogonal persistence to applications and rely on object faulting. They differ in the swizzling strategies and/or object fault detection mechanisms used. The implementations were (mostly) carried out by modifying the Smalltalk *virtual machine* (encompassing the Smalltalk interpreter and runtime library) to provide the illusion of a large persistent heap. When the modified virtual machine detects an attempt to access a non-resident object it raises an object fault that causes the required object to be mapped.

The designers of Mneme distinguish between two different general strategies for object fault detection: *edge marking* and *node marking*. Viewing the heap as a directed graph in which nodes represent objects and edges represent object references, edge marking marks all references to non-resident objects while node marking marks all nodes of the object graph that represent non-resident objects.

Early implementations of persistent Smalltalk using both edge and node marking are described in [32]. Since Smalltalk already used tagged pointers, the edge-marking implementation used an unassigned tag value to mark references to non-resident objects and stored the 28-bit Mneme client identifier for the target object in the remainder of the reference. The virtual machine performs a check on each reference before dereferencing it. When a marked reference is found, the address of the object is obtained from the Mneme object store (resulting in the object being mapped if it was non-resident). Note that, in this scheme, a marked reference can be swizzled to an appropriately tagged pointer once the object is mapped. Thus, this scheme can use lazy swizzling. However, references are usually located on the stack when presented to the virtual machine and hence the source of the reference cannot be updated (although other marked references to the object on the stack can).

The node marking implementation uses *descriptors* (called *fault blocks*) as representatives for non-resident objects. References from resident to non-resident objects are implemented as pointers to the corresponding descriptors. Descriptors are made to resemble object headers with a single bit being used to distinguish them from real headers. The client identifier of the object that is represents is stored in each descriptor and replaced with the object's address once mapped resulting in a so-called *indirect block*. This indirection can subsequently be bypassed by updating any pointers to the indirect block with pointers to the target object, E.G. during garbage collection.

Various optimisations are used within the virtual machine to reduce the frequency of residency checks, E.G. always mapping an object's class and superclasses whenever the object itself is mapped.

When objects are retrieved from the object store a number of conversions are necessary to convert the object from the Mneme on-disk format to that expected by persistent Smalltalk. Most importantly, if node marking is used, references contained in newly-mapped objects are eagerly swizzled to point to the referenced objects, if resident, or to descriptors for non-resident objects.

During unmapping, newly created objects that are transitively reachable from persistent objects that are being unmapped are promoted to being persistent and their newly-allocated object identifiers stored in the objects that refer to them. In addition, all pointers to objects that are being unmapped must be found and updated to point to corresponding descriptors (in the case of node marking) or marked to indicate that they refer to non-resident objects (in the case of edge marking). Rather than requiring an address space scan to locate such pointers, persistent Smalltalk maintains a *remembered set* [53] associated with each Mneme buffer. Each remembered set records the memory locations that might contain pointers into the corresponding buffer.

A number of alternative implementations of persistent Smalltalk are presented in [31]. The first uses a modified edge-marking scheme that exploits the Smalltalk method lookup algorithm to avoid explicit residency checks. References to non-resident objects are represented as tagged immediate object identifiers that are associated with a special class whose only method responds to all invocations by faulting the target object and forwarding the invocation to it. The Smalltalk method cache is loaded with this method the first time that an invocation is made on an object identifier so that subsequent invocations can proceed without an explicit residency check.

A similar approach was also used for node marking. Rather than using descriptors to represent non-resident objects, this implementation uses *proxies* to represent such objects. A proxy can be considered to a representative for a non-resident object to which code implementing the interface to the object is bound and which can therefore be invoked as if it were the object that it represents. These proxies respond to all invocations by faulting the corresponding object and forwarding the invocation to it once it has been mapped.

Finally, a third object fault detection scheme, which relies on trapping virtual memory protection faults, was also implemented. This scheme ensures that the

Smalltalk virtual machine only sees resident objects. In this scheme, descriptors are protected so that any attempt to access a descriptor causes a protection fault. The protection fault handler then maps the corresponding object and restarts the interrupted access with a pointer to the real object. The descriptor is replaced by an indirect block pointing to the target object and protected. The implementation keeps a remembered set for each page of descriptors so that objects that contain pointers to descriptors can have their pointers updated to point to the corresponding objects once they have been mapped thereby avoiding unnecessary (and expensive) protection traps.

Persistent Modula-3 The main goal of the persistent Modula-3 implementation was to explore compile-time optimisation of persistent programs in the context of a statically-typed language.

Unlike persistent Smalltalk, persistence in persistent Modula-3 is not orthogonal to type. In a way similar to the E extensions to C++, Modula-3 was extended with a separate hierarchy of persistent types by adding *persistent references* and *persistent top-level variables* to the language and having persistence be inherited from supertypes.

The implementation uses descriptors to represent non-resident objects and has the compiler generate the necessary code to check residency when dereferencing a persistent reference. As in the node-marking implementations of persistent Smalltalk, references stored in objects are eagerly swizzled when the object is mapped. Likewise, descriptors are replaced with indirect blocks when the object to which they refer is mapped. Pointers to indirect blocks can be replaced with direct pointers to the corresponding objects when the indirect block is created or subsequently, E.G. by the garbage collector. In addition, descriptors may be used as proxies by binding code to each descriptor, which when invoked faults the corresponding object and forwards the invocation to it. In this case, invocations on non-resident objects can be trapped without an explicit residency test. Note that this latter approach to object faulting alone is insufficient since Modula-3 allows clients of an object direct access to its fields.

Objects can be unmapped incrementally or at program termination. On unmapping, swizzled objects are replaced with indirect blocks pointing to a descriptor for the object. Memory reclamation is based on the use of remembered sets to keep track of all pointers into the persistent heap.

Various possible compile-time optimisations for persistent Modula-3 were suggested including techniques for the elimination of redundant residency checks, dead code elimination, code motion, procedure inlining (including message splitting), and customised compilation [30]. Having programers use co-residency annotations to specify objects that should be mapped together was also suggested as a way of eliminating residency checks between objects and of driving the clustering of related objects on disk. However, it is not clear which, if any, of these optimisations were actually implemented.

2 Languages for Distributed Programming

Many object-oriented programming languages for distributed programming have been designed either as completely new languages or as extensions to existing languages. Early object-support operating systems such as Argus [36] and Eden [1] typically mapped every remotely-accessible object onto a single process making such objects rather heavyweight when compared to the fine-grained objects supported by conventional object-oriented programming languages. The result was that the supported programming languages – the Argus programming language and EPL in the case of Eden – were forced to provide a two-level object model in which different type hierarchies were provided for remotely-accessible objects and for local objects respectively. In Argus, *guardians* represent remotely-accessible entities while CLU clusters are used to represent the local state of an individual guardian[5]. Likewise, EPL provides *ejects* to represent remotely-accessible entities and Concurrent Euclid data structures to represent the local state of an eject[6]. In each case, different syntax is used to define locally and remotely-accessible objects. It may argued that such a model more closely reflects the architecture of the underlying distributed system and makes the relative costs associated with access to local and remote resources explicit, thereby forcing programmers to consider carefully which objects should be remotely accessible [35].

Most recent object-support operating systems support remote access to fine-grained objects, I.E. objects that are potentially much smaller that the size of an address space, and optimise access to remotely-accessible objects that happen to be local thereby making a uniform object model feasible. Since access to objects that are known to be local can often be more efficient than access to objects that are potentially remote, language designers may still choose to make the distinction explicit by providing a two-level object model. Clearly, this decision is analogous to the decision as to whether or not to support orthogonal persistence in a persistent programming language. Of the languages reviewed later in this section, Amadeus/C**, PANDA/C++, and FOG provide two-level object models while Emerald, Amber, and Orca provide uniform object models.

Whether or not separate type hierarchies are supported for local and remotely-accessible objects, most object-oriented languages for distributed programming support a model is which objects are the unit of distribution, I.E. the programmer of an object perceives that object as always being wholly located at a single node. Some languages allow objects whose states can be partitioned over a number of nodes to be defined. Of the languages described below, Amadeus/C**, PANDA/C++, Emerald, Amber, and Orca have objects as their unit of distribution while FOG supports partitioned objects. Note that the model provided to programmers may not match the distribution strategy employed in the implementation of the language. For example, even though objects are the

[5] The Argus programming language is an extension of CLU.
[6] EPL is an extension of Concurrent Euclid.

unit of distribution, most implementations of Orca replicate objects at different nodes in order to improve performance for certain classes of application.

The remainder of this section describes a number of the most influential object-oriented languages for distributed programming. Many more such languages exist. However, the aim of this section is to demonstrate the major issues that arise and the techniques that are used in the implementation of object-oriented languages for distributed programming. Thus, the languages reviewed here were chosen to illustrate a range of different implementation techniques. See [22] for another survey of object-oriented programming languages for distributed programming.

2.1 Amadeus/C** – Distributed and Persistent Programming in C++

Amadeus (see section **??**) is a general-purpose object-support operating system supporting distributed and persistent programming in multi-user distributed systems. Amadeus is one of a number of object-support operating systems developed within the context of the ESPRIT Comandos project [17]; others include the IK [51], Guide [29], and CHORUS/COOL [34] systems. Although Amadeus was designed to allow a range of object-oriented programming languages to be supported simultaneously via its Generic Runtime Library (GRT) [16], this section describes only one of the supported languages – an extension to C++ known as C**[7] [52]. Among the other languages supported by Amadeus are extensions to Eiffel and E known as Eiffel** [39] and E** [38] respectively.

C** distinguishes between ordinary C++ objects, *persistent objects*, *global objects*, and *atomic objects*. In C**, persistent objects exhibit the property of potential persistence. Global objects are objects that are remotely accessible and atomic objects are those for which the standard transaction properties of atomicity, consistency, isolation, and durability [28] can be guaranteed. Such objects are the units of concurrency control and recovery; an atomic object can be taken from one consistent state to another within a transaction.

Potential persistence and remote accessibility are not orthogonal to type. A class whose instances can be potentially persistent is known as a *permclass* while a class whose instances can be remotely accessible is known as a *global class*. A given class may be both a permclass and a global class. However, only dynamically allocated instances of permclasses or global classes are persistent or global objects respectively. In contrast, atomicity is a property of individual persistent or global objects. Any persistent or global object can be made atomic.

A permclass is declared by using the new keyword `permclass` in place of the C++ keyword `class` in a class declaration. A global class is any class that has a

[7]A number of other researchers have developed extensions to C++ for distributed or parallel programming, which they called C**. Throughout this thesis, the name "C**" refers to the C++ extension supported by Amadeus.

global method. Global methods are declared by using the keyword `global` in the method declaration. A global class may have non-global methods but can only be accessed remotely via its global methods and hence should be strongly encapsulated. Potential persistence and remote accessibility are inherited. Moreover, a permclass or global class can inherit from a non-permclass or non-global class respectively.

A major goal of Amadeus was to allow supported languages to retain their own local object reference format and invocation mechanism thereby allowing their existing compilers to be retained. Thus, C** employs an object faulting strategy in which object references contained in local objects are always virtual memory addresses and referenced objects that are not mapped locally are either represented by local proxies (called *O-proxies*) or protected virtual memory pages (called *C-proxies*).

An absent global object is represented by an O-proxy that occupies the same amount of memory as the object itself and has remote object invocation (ROI) stub code bound to it. Object faults resulting from attempts to invoke absent global objects are detected by the stub code bound to their O-proxies. When invoked, the O-proxy code marshals the invocation request and calls Amadeus to resolve the object fault. In the case of a global object, Amadeus may either carry out a ROI on the target object (which it always does if the object is already mapped remotely) or map the object locally overlaying its O-proxy.

In Amadeus, the unit of storage and mapping is not a single object but a cluster of (related) objects. A cluster consists of an integral number of pages and contains either a set of global objects or a set of non-global but persistent objects.

An absent cluster containing non-global persistent objects is represented by a C-proxy that occupies the same number of pages as the cluster itself. Object faults resulting from attempts to access the persistent objects contained in such a cluster are detected as memory protection violations. These object faults are always resolved by mapping the target cluster in place of its C-proxy since, by definition, a non-global persistent object must always be accessed locally.

Object references stored in persistent objects or transmitted over the network are called *global references*. A global reference holds not only the identifier of the referenced object but also enough information to create an O- or C-proxy for it when required: its class and size for global objects, or the identifier and size of its cluster and its offset within the cluster for non-global persistent objects. Object identifiers are globally unique identifiers that can be used to locate an object anywhere in the distributed system including, in the case of a persistent object, on disk.

When a cluster containing global objects is mapped into an address space as a result of an object fault, the required object is copied out of its cluster to overlay its O-proxy so that local references to the object, and any embedded global objects contained within it, remain valid. The object is then *prepared*, I.E. made ready to accept invocations. This involves swizzling the global references contained within the object to the appropriate virtual addresses and binding the object's class code to it. Swizzling a reference to an absent object may result in an O- or C-proxy for the object being created as appropriate. Swizzling a reference to a local object that has not yet been prepared (I.E. because it was mapped as a result of a fault on a different object) results in the object being put into a state in which any future attempt to invoke it will cause it to be prepared. Thus, a single object at a time can be swizzled without requiring that all the objects that it references be swizzled. In the case of a cluster containing non-global persistent objects, the cluster overlays its C-proxy and the objects belonging to the cluster are prepared.

Clustering of objects is controlled explicitly by applications and primitives are provided to allow new clusters to be created, objects to be created in specific clusters, and objects to be moved between clusters. Clusters are normally unmapped at process termination but applications may explicitly unmap clusters at any time.

Finally, C** also provides support for management of atomic transactions as described in [40].

2.2 PANDA/C++ – Distributed and Persistent Programming in C++

The PANDA project at the University of Kaiserslautern [3] also addressed the development of an environment for parallel and distributed programming in C++. Major goals of PANDA included the provision of support for parallelism, distribution, and persistence in C++ with as few restrictions as possible on the use of existing C++ language constructs. In addition, the support provided by PANDA was intended to be easily customised and extended by applications to meet their own requirements. A major disadvantage of PANDA is that it is restricted to networks of homogeneous (multiprocessor) workstations.

PANDA provides of a small operating system kernel together with an associated runtime library and hence may be seen as an example of an application-specific operating system as described in section **??**. The kernel is intended to provide minimal functionality in order to reduce the frequency of kernel calls. Most of the functionality provided by a typical operating system is implemented in user space allowing it to be more easily tailored to the needs of specific applications. The kernel only provides those features that are necessary to support secure sharing of hardware resources between applications. In particular, the kernel provides *protection domains*, which provide the ability to control access to address spaces on a per-page basis, and *virtual processors*, which abstract physical processors. The PANDA kernel is referred to as a *picokernel* in order to emphasise its small size and minimal functionality.

Support for parallel, distributed, and persistent programming is provided by the runtime library. The runtime library is implemented as a set of C++ class hierarchies that can be specialised by applications if required. Included in the runtime library are class hierarchies supporting very lightweight user-level threads and various thread synchronisation mechanisms.

In order to support a high degree of transparency for C++ programmers, PANDA implements a single distributed address space using page-based distributed shared memory (DSM) techniques [4, 2]. Hence, virtual addresses are valid across all nodes of the system and can be used as global references for objects. In fact, the address space is divided into a number of partitions such that each node owns at least one private partition and at least one shared partition that is accessible to other nodes. In particular, pages of shared partitions may be mapped at other nodes.

Objects may be created in private or shared partitions under application control. A new keyword (`DSM`) is used to indicate object creations that should allocate the object in a shared partition. Faults on remote objects are detected as memory protection violations. The protection fault handler can use any of a number of different policies to resolve the fault. For example, it can fetch the required page or migrate the faulting thread to the current location of the page [4]. If the fault address refers to a private segment then the only option is to migrate the thread. In any case, since the page fault handler is provided by the runtime library, it can be specialised to implement whatever policy is appropriate to the application. As described in [2], the default protection fault handler uses a single writer/multiple reader write-invalidate DSM protocol. The ability to migrate a thread to a specified node is provided by the thread class hierarchy and hence applications may also explicitly migrate threads. PANDA uses page-differencing in order to reduce the volume of data to be transmitted in response to a page fault. In addition, in order to reduce the negative effects of false sharing (when unrelated objects are allocated on the same shared page) and to support prefetching of related objects, PANDA allows so-called DSM *clusters* to be used as a means of grouping related objects on disjoint sets of pages. PANDA also provides the ability to collocate objects by *tying* and *untying* objects that are likely to be accessed together [2]. An object can also be tied (*fixed*) to a specific node.

As in C**, persistence is not orthogonal to type. Classes whose instances may be persistent are labelled with the keyword `persistent`. Dynamically allocated instances of persistent classes are persistent objects. Persistent objects are allocated in dedicated shared memory partitions and hence are always remotely accessible. PANDA provides a basic name service allowing symbolic names to be attached to persistent objects and references to persistent objects to be retrieved given their symbolic names. Transactions over persistent objects are supported using a shadow-paging scheme with pages of persistent partitions being stored in a simple block-oriented file system. As far as can be determined, the implementation does not employ swizzling and only requires that a mapping from page address to disk location be maintained. In addition, garbage collection of instances of classes derived from the base class `GC_obj` is supported by means of weighted reference counting. The implementation uses smart pointers with garbage-collectable references being implemented by a template class `GC_ref`.

2.3 Emerald – Fine-Grained Object Mobility

Emerald is an object-oriented language for distributed programming developed at the University of Washington [13, 14, 33]. Emerald was intended to make distributed programming easy by providing complete network transparency while at the same time allowing programmers to exploit both knowledge about object location and the ability to migrate fine-grained objects easily.

Emerald was very influential having been among the first distributed programming languages to provide uniform object models. However, although Emerald programmers are provided with a uniform view of objects, objects can be implemented in different ways depending on their intended use. Three different styles of implementation are supported – *direct, local, and global* – as described below.

In Emerald, objects are not instances of pre-defined classes. Instead, an object is created by executing an *object constructor*. While each individual object conceptually has its own private set of methods, objects created by the same object constructor may share code.

Emerald is strongly-typed. Every object and every variable has an abstract type. An assignment of an object to a variable can only proceed if the abstract type of the object *conforms* to the abstract type of the variable. An abstract type is defined by a collection of method signatures each of which describes one method – its name, parameters, and results. Conformance of abstract types is defined as a relation between the signatures of the types. Basically, a type `A` conforms to another type `B` if objects of type `A` can be used everywhere that an object of type `B` can be used. Conformance allows multiple implementations of a given abstract type to exist.

An Emerald object may optionally include a process and a new process is created whenever such an object is created. The newly-created process initially invokes a specified method on the object in the course of which it may invoke methods on other local and remote objects. An Emerald process is defined by its stack, which contains the stack frames for its incomplete invocations. If a process invokes methods on objects at remote nodes then its stack will become segmented with segments being located at various nodes. Thus, Emerald processes are potentially distributed. Concurrent execution of the methods of an object is permitted and may be synchronised using monitors.

At compile time, the Emerald compiler chooses the appropriate implementation style for each object depending on the object's use. *Global objects* are objects that can be referenced from anywhere in the distributed system, can be accessed remotely, and can move independently. Such objects are heap allocated and an invocation on such an object may require a ROI. *Local objects* are known only by one other object in the system, I.E. they are logically contained within another object. They cannot move independently of their containing object but always move with it. Since they are always local to their containing object, they are accessed using local procedure calls. Local objects are also heap allocated. Finally, *direct objects* are local objects that are allocated directly within the representation of their containing object, I.E. embedded objects. Direct objects are used for objects whose representation can be deduced at compile time, including objects of built-in types.

Global objects are identified by globally unique object identifiers. However, a local reference to a global object gives the address of a local *descriptor* for the target object. The descriptor indicates whether or not the object is resident locally and, if so, includes a pointer to its data. If not, it gives a hint for the object's current location, which is treated as a forwarding pointer. Thus, local object references are indirect pointers and object faults are detected by checking the residency flag in the descriptors of global objects. Object faults are resolved by performing a ROI on the target object. References to global objects are translated into the corresponding object identifiers when transmitted between nodes in invocation requests or within migrating objects. On the receiving side, object identifiers are translated into the corresponding local object references, possibly resulting in the creation of local descriptors for the referenced objects. Since it is necessary to translate object references when an object migrates, the Emerald compiler generates a template describing the layout of each object that allows the references within the object to be located.

An important aspect of Emerald is that it provides an explicit notion of object location and allows fine-grained object mobility. At any time an object may be moved to a new location specified by a node name or the location of another object. It is likewise possible to inquire about the current location of an object. Other primitives allow programmers to *fix* the location of an object, *unfix* its location, and *refix* its location (I.E. atomically unfix the object, move it, and fix it at its new location). An object may migrate at any time even if there are outstanding invocations on the object in progress, although this requires the migration of the stack frames currently associated with the object as described in [33].

One of the advantages of such fine-grained mobility is that it allows objects to be passed efficiently by reference. A local object to which a reference is passed as a parameter in a ROI request can be transmitted along with the corresponding request, thereby avoiding the overhead of a further ROI to access that object should it be necessary. Emerald provides two parameter passing modes that allow applications to direct the system as to which parameters should be transmitted with ROI requests: the *by move* and *by visit* parameter passing modes.

Associated with object mobility is the issue of which of its component (global) objects should be transmitted along with an object to be migrated. For this purpose Emerald allows an application to declare *attached variables*. Objects referenced by attached variables will be transmitted along with the object to which they are attached.

2.4 Amber – C++ Meets Emerald

Amber is an object-oriented language for distributed and, especially, parallel programming that, like its predecessor Emerald, was developed at the University of Washington [21]. Amber is intended to allow a single application program to use a network of homogeneous shared-memory multiprocessors as a single integrated multiprocessor. Although Amber provides no support for persistence, reliability, or sharing between multiple programs and is based on an existing operating system, it is similar in many respects to the PANDA system described above.

Like PANDA, Amber supports the use of C++ for programming parallel and distributed applications and provides the C++ programmer with the illusion of a network-wide shared object space. In general, Amber objects are fine-grained, passive entities that can be manipulated by multiple threads by means of location-independent object invocation. In addition, objects are mobile in that they can be explicitly migrated between nodes under program control.

Amber programmers are expected to be aware of and take advantage of the organisation of the network. Application programmers can control the location of data and processing using object migration primitives provided by Amber. Moreover, object placement is *entirely* under program control – objects never move unless a program explicitly moves them. A thread that invokes a method on an object always moves to the node at which the object is currently located.

Pre-defined classes support thread management and synchronisation, and provide the object mobility primitives. Providing such facilities via extensible class hierarchies is also intended to support application-specific customisation. For example, Amber allows its thread scheduler to be replaced at runtime by an instance of some subclass of the pre-defined scheduler class. Likewise, although the Amber class hierarchy supports a variety of synchronisation mechanisms such as relinquishing and non-relinquishing locks, barrier synchronisation, monitors, and condition variables, applications are free to extend the class hierarchy to provide other application-specific synchronisation mechanisms.

Amber's mobility primitives are based on those of Emerald and include primitives to locate an object (`locate`) and to move an object to a specified location (`moveTo`). In addition, applications can dynamically form groups of objects that will be moved together by means of the `attach` and `unattach` primitives. Objects are moved even if there are active invocations on them. In this case, the corresponding threads are identified and moved with the object. Amber also supports replication of read-only objects and the ability to dynamically specify that an object is `immutable`. The `moveTo` primitive makes copies of immutable objects rather than moving them.

As in PANDA, Amber implements a single distributed address space. Virtual addresses are valid across all nodes in the system and can be used as global object references. Program code and statically initialised program data are automatically replicated at the same addresses on all nodes. The remainder of the address space is (dynamically) partitioned between the nodes making up the system and new objects are created in a partition belonging to the creating node. Thus, the node at which an object was created can always be determined from its virtual address.

Unlike PANDA, Amber does not use page-based DSM to implement object fault detection and resolution. Instead, every Amber object is represented by a descriptor on every node. Each descriptor indicates whether or not its associated object is locally resident. On the node where the object is located, the descriptor doubles as the object's header. On other nodes, the descriptor may contain a hint as to the current location of the object. Residency checks (inserted by a preprocessor) are performed whenever an object is invoked and on return from every invocation (to ensure that the calling object has not been moved)[8].

When an object fault is detected, it is resolved by performing a ROI on the target object. ROI is implemented by migrating the invoking thread to the node where the object is located. Migrating a thread involves copying the local control information for the thread and part of its stack to the target node. Objects are located by means of a forwarding protocol starting from the location (if any) specified in the object's descriptor or from the node on which the object was created. In order to avoid remote communication when a new object is created, any unwritten memory is interpreted as being an uninitialised descriptor that is, in turn, interpreted to mean that the object is remote (and can be located starting from the node at which it has created).

Unlike PANDA, which tries to support all of the features of C++ over the network, in Amber a program may never directly manipulate the internals of a remote object since object faults can essentially only be detected on an invocation. Hence, as in Amadeus/C**, any data that may be referenced remotely must be encapsulated within an object. It is still possible to optimise interactions between objects that are known to reside on the same node (such as attached objects, embedded objects, and stack-allocated objects) using the various features of C++ that provide direct access to the internals of an object.

2.5 Orca – Transparent Replication and Migration

Orca [7] is a distributed programming language developed at the Vrije University in Amsterdam. Orca was designed from scratch as a language in which to implement parallel applications on loosely-coupled distributed systems [9]. In Orca, shared data structures are encapsulated in so-called *data objects* that are accessed through a set of user-defined operations. Thus, data objects may be regarded as instances of user-defined abstract data types. Note however that Orca is not strictly speaking an object-oriented language since it does not

[8] An additional residency check is required on every context switch into a preempted thread. In order to ensure that threads migrate along with the objects that they are using, moving an object preempts all the threads that are executing on its source node in order to force them to check if the object that they are accessing is being moved.

support inheritance or dynamic binding. In contrast to Emerald and Amber, Orca explicitly tries to hide the distribution of data from the programmer. Objects may however be migrated or replicated by the Orca runtime library transparently to applications.

In Orca, multiple processes execute by invoking type-specific operations on objects. Multiple processes, perhaps on different machines, may (logically) share objects. An Orca program declares one or more process types and parallelism is achieved by the explicit creation of new processes of those types (optionally on specified nodes). The declaration of a process type specifies the parameters required by processes of that type. For an object to be shared by a process, that object must have been passed as a `shared` parameter to the process when it was created. `shared` is essentially a call-by-reference parameter passing mode – only objects can be passed as `shared` parameters. Passing objects as `shared` parameters is the only way to express sharing and hence an object can only be shared between a process and its descendents.

Mutual exclusion synchronisation to shared objects is taken care of automatically in that all invocations on an object execute indivisibly. Individual invocations are serialised in an undefined order[9]. However, while single invocations are indivisible, sequences of invocations are not. In addition, Orca does not explicitly support indivisible invocations on multiple objects. Condition synchronisation is integrated with operation invocation by allowing invocations to block using guarded commands.

A number of implementations of Orca are described in the literature – see, for example, [12] and [10]. Unlike most of the systems reviewed in this section, most implementations of Orca make aggressive use of replication of shared objects to decrease access times and increase parallelism. For example, the implementation described in [9] replicates all objects on all processors and updates replicas using a reliable totally-ordered broadcast protocol. In this implementation, the compiler analyses the implementation of every operation to determine if it modifies the object to which it applies and categorises operations as read operations or write operations. This information is then stored in a descriptor for the operation. At runtime, every processor keeps a local copy of every shared object. Invocations on an object are compiled into calls to a routine in the Orca runtime library that takes the operation descriptor as one of its parameters. Operations that do not change the object are applied directly to the local copy of the object. Operations that do change the object are broadcast to all the processors in the system (including that of the caller) using the reliable totally-ordered broadcast protocol and applied to the local copy of the object at each processor in the order in which they are received. Thus, all processes observe changes to shared objects in the same order. Since read access is always handled locally, this implementation favours parallel applications in

[9] It is still possible to execute multiple invocations on the same object in parallel as long as the effect of those invocations is the same as some serial execution, E.G. multiple reads can execute in parallel.

which processes share data that is read frequently and modified infrequently.

The strategy of replicating objects at every node is inefficient in some circumstances – most notably when objects are shared by only a few of the processes in the system or when objects only have write operations. To address these issues a more recent implementation of Orca [8] uses a combination of replication, migration, reliable ordered broadcast, and ROI to implement access to shared objects. In this implementation, objects are either replicated everywhere (and updated using reliable ordered broadcasts) or are located at exactly one processor (and accessed remotely using ROI). A combination of compile-time analysis and run-time heuristics are used to decide whether or not to replicate an object, and, if not, to decide where it should be located. Whether or not an object is replicated, and, if not, its location, can change dynamically as processes are created.

Essentially, the Orca compiler has been extended to determine the expected access patterns of processes to shared objects. The compiler passes a summary of this information to the Orca runtime library, which uses it to decide which objects to replicate and where to locate non-replicated objects.

The compiler computes two values for each object that a process can access: an estimate of the number of read operations that the process performs on the object and an estimate of the number of write operations that it performs. This information is provided to the runtime library by means of a descriptor for each process type that describes how processes of that type are expected to access their `shared` parameters. When a new process is created the runtime library uses this information to decide if the current mapping of objects to processors is still desirable. To do this, the runtime library maintains the expected total number of reads and expected total number of writes on every object by every process. This information is replicated in the runtime library at every processor and must be updated whenever a new process is created. Thus, all process creation requests are broadcast to all processors. As a result, all processors can make the same decisions in parallel.

To decide whether or not to replicate an object, the (expected) number of messages that will be generated as a result of attempts to access that object, both with and without replication, is calculated. With replication, one broadcast is needed for every write operation by any process. Without replication one ROI is needed for every access to the object by processes at nodes other than that at which the object is located. The option resulting in the lowest communication overhead is chosen. If the decision is not to replicate the object, then it is placed at the processor that is expected to make the greatest number of accesses to the object. As noted, these decisions are re-evaluated whenever a process is created, giving rise to the possibility that an object might by replicated, unreplicated, or migrate whenever a process is created.

In practice, the use of these heuristics lead to better performance than was achieved by the implementation based on full replication. However, they do not handle well situations in which the access patterns to objects vary over time.

2.6 FOG – Fragmented Objects in C++

FOG (for Fragmented Object Generator) [27] is yet another distributed programming language based on C++, this time developed by the SOR group at INRIA Rocquencourt. Rather than trying to make distribution transparent to C++ programmers by supporting a model in which objects are the units of distribution, FOG supports *fragmented objects* [37]. A fragmented object (FO) is one that is composed of a number of distributed fragments. The external (or client) view of an FO is of a single shared object, I.E. the distribution of the object is transparent to its clients. However, the programmer of an FO can control the distribution of data and processing among the fragments of the object as well as the communication between the fragments. Thus, while the standard shared object model makes distribution transparent to both the clients and the programmer of an object, the FO model makes distribution explicit to the programmer of an FO.

An FO encapsulates a set of cooperating *fragments*. Each fragment is located at a single node. The fragments of an FO cooperate to maintain a consistent view of the object as a single logical entity to its clients. Fragments cooperate using lower level FOs to communicate. A number of primitive FO types, including FOs supporting point-to-point and multicast communication channels, are provided by the FOG runtime library.

Clients access an FO by means of a local (to the client) interface fragment or *proxy* as defined in [49]. A client cannot distinguish between the interface of the proxy and that of the FO of which it is a part. Moreover, a single FO can offer different interfaces to different clients via different proxies. As in the traditional client/server model, clients must bind to an FO before invoking it. The FO then provides the client with a proxy supporting the interface that the client expects. Since the binding is a runtime operation, a combination of both compile-time and run-time checks are necessary to ensure that the interface provided conforms to that expected by the client.

From the programmer's point of view, an FO is an object with a fragmented representation composed of a set of fragments (each located in a single address space), one or more *connective objects* (the lower-level FOs used for communication between the fragments), and a binding interface to allow clients to bind to the FO. The interface to a fragment is composed of three parts. The public interface provides methods accessible to clients. The private interface provides methods that are only accessible to the fragment itself. Finally, the group interface provides methods that can be invoked by other fragments of the FO via its connective objects. Typically, a client invocation on a method provided by the public interface results in the invocation of one or more methods from the private interface that, in turn, initiate communication between the fragments via methods in the group interface.

The FOG language extends C++ to allow the components of an FO and the connections between them to be declared. The designers of the language refer to it as being "essentially a declarative interface language" [37]. In FOG, an FO is an instance of a *class group* that lists the classes of its fragments. Each class declares the public, private, and/or group interfaces supported by fragments that are instances of the class, as well as their component objects (including any connective objects). The FOG compiler provides support for binding as well as for marshalling data that crosses the group interface of a fragment into whatever form is appropriate for transmission via the interface of the connective object(s) used.

3 Conclusions

This paper presents a review of some of the most influential research into implementation models for distributed and persistent objects. In particular, a range of object-oriented programming languages for distributed or persistent programming were described in order to illustrate the various different object models and implementation techniques.

For object models which support persistent programming, the major distinctions between the various languages concern whether or not they support orthogonal persistence, how persistent values are identified, whether garbage collection or explicit deletion is supported, how the interface to the underlying persistent store is presented within the language, and, finally, whether object clustering (if supported by the underlying runtime support) is made visible to programmers. In terms of the underlying implementations, major distinguishing features include whether object faulting or object fault avoidance is used, in the case of object faulting what strategy is used for object-fault detection, whether or not swizzling is used (and if so whether eager or lazy swizzling is employed), and, whether and how clustering is supported.

In the case of object models supporting distributed programming, the major distinctions between the various languages concern whether or not they support a uniform object model, what the unit of distribution presented to programmers is, whether the distributed nature of the system is hidden or explicitly exposed to programmers, and what primitives (if any) are provided for object clustering and migration. In terms of the underlying runtime support, many of the same issues arise as in the implementation of persistent programming languages. Important distinctions include whether or not object faulting is used, what strategy is used for object-fault detection when applicable, and, finally, whether or not swizzling of global references is employed. In addition, while the runtime libraries of persistent programming languages always resolve or avoid object faults by mapping the target object locally, the runtime libraries of languages for distributed programming can use a number of other mechanisms, such as ROI or thread migration, to allow absent objects to be accessed.

References

[1] Guy T. Almes, Andrew P. Black, Edward D. Lazowska, and Jerre D. Noe. The Eden system: A technical review. *IEEE Transactions on Software Engineering*, SE-11(1):43–58, January 1985.

[2] Holger Assenmacher, Thomas Breitbach, Peter Buhler, Volker Huebsch, Holger Peine, and Reinhard Schwarz. Parallel programming in PANDA. Submitted for publication.

[3] Holger Assenmacher, Thomas Breitbach, Peter Buhler, Volker Huebsch, and Reinhard Schwarz. The PANDA system architecture - a pico-kernel approach. In *Proceedings of the* 4^{th} *Workshop on Future Trends of Distributed Computing Systems*, pages 470–476. IEEE Computer Society Press, September 1993.

[4] Holger Assenmacher, Thomas Breitbach, Peter Buhler, Volker Huebsch, and Reinhard Schwarz. PANDA - supporting distributed programming in C++. In Oscar M. Nierstrasz, editor, *Proceedings of the* 7^{th}*European Conference on Object-Oriented Programming*, volume 707 of *Lecture Notes in Computer Science*, pages 361–383. Springer-Verlag, 1993.

[5] M. P. Atkinson, P. J. Bailey, K. J. Chisholm, W. P. Cockshot, and R. Morrison. An approach to persistent programming. *Computer Journal*, 26(4):360–365, 1983.

[6] Malcolm Atkinson, Ken Chisholm, and Paul Cockshot. PS-algol: An Algol with a persistent heap. *ACM SIGPLAN Notices*, 17(7):24–31, 1982.

[7] Henri Bal. *Programming Distributed Systems.* Silicon Press/Prentice Hall, 1990.

[8] Henri E. Bal and M. Frans Kaashoek. Object distribution in orca using compile-time and run-time techniques. In Paepcke [43], pages 162–177. Also SIGPLAN Notices 28(10), October 1993.

[9] Henri E. Bal, M. Frans Kaashoek, and Andrew S. Tanenbaum. A distributed implementation of the shared data-object model. In *Proceedings of theWorkshop on Experiences with Distributed and Multiprocessor Systems*, pages 1–19. USENIX Association, October 1989.

[10] Henri E. Bal, M. Frans Kaashoek, Andrew S. Tanenbaum, and Jack Jansen. Replication techniques for speeding up parallel applications on distributed systems. Technical Report IR-202, Department of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, December 1989.

[11] Henri E. Bal, Jennifer G. Steiner, and Andrew S. Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, 21(3):261–322, September 1989.

[12] Henri E. Bal and Andrew S. Tanenbaum. Distributed programming with shared data. In *Proceedings of theInternational Conference on Computer Languages*, pages 82–91. IEEE Computer Society Press, October 1988.

[13] Andrew Black, Norman C. Hutchinson, Eric Jul, and Henry M. Levy. Object structure in the Emerald system. In Norman Meyrowitz, editor, *Proceedings of the 1986 Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 78–85. Association for Computing Machinery, Association for Computing Machinery, September 1986. Published version of [**?**].

[14] Andrew Black, Norman C. Hutchinson, Eric Jul, Henry M. Levy, and Larry Carter. Distribution and abstract types in Emerald. *IEEE Transactions on Software Engineering*, SE-13(1):65–76, January 1987. Published version of [**?**].

[15] A. L. Brown and W. P. Cockshott. The CPOMS persistent object management system. Technical Report PPRR-13-85 (Persistent Programming Research Report 13), Universities of Glasgow and St. Andrews, 1985.

[16] Vinny Cahill, Seán Baker, Gradimir Starovic, and Chris Horn. Generic runtime support for distributed persistent programming. In Andreas Paepcke, editor, *Proceedings of the 1993 Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 144–161. ACM Special Interest Group on Programming Languages, ACM Press, September 1993. Also SIGPLAN Notices 28(10):144–161, October 1993 and technical report TCD-CS-93-37, Dept. of Computer Science, Trinity College Dublin.

[17] Vinny Cahill, Roland Balter, Xavier Rousset de Pina, and Neville Harris, editors. *The COMANDOS Distributed Application Platform*. ESPRIT Research Reports Series. Springer-Verlag, 1993.

[18] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. Modula-3 report (revised). Technical Report Research Report 52, Digital Systems Research Center, November 1989.

[19] Michael J. Carey, David J. DeWitt, Daniel Frank, Goetz Graefe, M. Muralikrishna, Joel E. Richardson, and Eugene J. Shekita. The architecture of the EXODUS extensible DBMS. In Klaus R. Dittrich and Umeshwar Dayal, editors, *Proceedings of the International Workshop on Object-Oriented Database Systems*, pages 52–65. IEEE Computer Society Press, 1986.

[20] Michael J. Carey, David J. DeWitt, Joel E. Richardson, and Eugene J. Shekita. Object and file management in the EXODUS extensible database system. In *Proceedings of* 12^{th} *International Conference on Very Large Data Bases*. Morgan Kaufmann, August 1986.

[21] Jeffrey S. Chase, Franz G. Amador, Edward D. Lazowska, Henry M. Levy, and Richard J. Littlefield. The Amber system: Parallel programming on a network of multiprocessors. In *Proceedings of the* 12^{th} *Symposium on Operating Systems Principles*, pages 147–158. ACM Special Interest Group on Operating Systems, December 1989. Also Operating Systems Review, 23(5).

[22] Roger S. Chin and Samuel T. Chanson. Distributed object-based programming systems. *ACM Computing Surveys*, 23(1):91–124, March 1991.

[23] W. P. Cockshot, M. P. Atkinson, K. J. Chisholm, P. J. Bailey, and R. Morrison. Persistent object management system. *Software Practice and Experience*, 14:49–71, 1984.

[24] W. P. Cockshott. Addressing mechanisms and persistent programming. Origin unknown.

[25] Alan Dearle, Gail M. Shaw, and Stanley B. Zdonik, editors. *Implementing Persistent Object Bases: Principles and Practice*. Morgan Kaufmann, September 1990. Proceedings of the 4^{th}International Workshop on Persistent Object Systems.

[26] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, Reading, MA, 1980.

[27] Yvon Gourhant and Marc Shapiro. FOG/C++: A fragmented-object generator. In *Proceedings of the* USENIX *C++ Conference*, pages 63–74. USENIX Association, April 1990.

[28] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[29] Daniel Hagimont, P. Y. Chevalier, A. Freyssinet, S. Krakowiak, Serge Lacourte, J. Mossiére, and Xavier Rousset de Pina. Persistent shared object support in the Guide system: Evaluation and related work. In Carrie Wilpolt, editor, *Proceedings of the 1994 Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 129–144. ACM Special Interest Group on Programming Languages, ACM Press, October 1994. Also SIGPLAN Notices 29(10), October 1994.

[30] Antony L. Hosking and J. Eliot B. Moss. Towards compile-time optimisations for persistence. In Dearle et al. [25], pages 17–27. Proceedings of the 4^{th}International Workshop on Persistent Object Systems.

[31] Antony L. Hosking and J. Eliot B. Moss. Object fault handling for persistent programming languages: A performance evaluation. In Paepcke [43], pages 288–303. Also SIGPLAN Notices 28(10), October 1993.

[32] Antony L. Hosking, J. Eliot B. Moss, and Cynthia Bliss. Design of an object faulting persistent smalltalk. Technical Report COINS Technical Report 90-45, Department of Computer and Information Science, University of Massachusetts at Amherst, May 1990.

[33] Eric Jul, Henry M. Levy, Norman C. Hutchinson, and Andrew Black. Fine-grained mobility in the Emerald system. Technical Report 87-02-03, Department of Computer Science and Engineering, University of Washington, February 1987.

[34] Rodger Lea, Christian Jacquemot, and Eric Pillevesse. COOL: System support for distributed programming. *Communications of the ACM*, 36(9):37–46, September 1993.

[35] Barbara Liskov. On linguistic support for distributed programs. *IEEE Transactions on Software Engineering*, SE-8(3):203–210, May 1982.

[36] Barbara Liskov and Robert Scheifler. Guardians and actions: Linguistic support for robust distributed programs. *ACM Transactions on Programming Languages and Systems*, 5(3):381–404, July 1983.

[37] Mesaac Makpangou, Yvon Gourhant, Jean-Pierre Le Narzul, and Marc Shapiro. Fragmented objects for distributed abstractions. In Thomas L. Casavant and Mukesh Singhal, editors, *Readings in Distributed Computing Systems*. IEEE Computer Society Press, 1994.

[38] John McEvoy. E**: Porting the E database language to Amadeus. Master's thesis, Department of Computer Science, Trinity College Dublin, 1993.

[39] Colm McHugh and Vinny Cahill. Eiffel**: An implementation of Eiffel on Amadeus, a persistent, distributed applications support environment. In Boris Magnusson, Bertrand Meyer, and Jean-Francois Perot, editors, *Technology of Object-Oriented Languages and Systems (TOOLS 10)*, pages 47–62. TOOLS/SOL, Prentice Hall, 1993. Also technical report TCD-CS-93-36, Dept. of Computer Science, Trinity College Dublin.

[40] Michael Mock, Reinhold Kroeger, and Vinny Cahill. Implementing atomic objects with the RelaX transaction facility. *Computing Systems*, 5(3):259–304, 1992. Also technical report TCD-CS-92-24, Dept. of Computer Science, Trinity College Dublin.

[41] J. Eliot B. Moss. Design of the Mneme persistent object store. *ACM Transactions on Information Systems*, 8(2):103–139, April 1990.

[42] J. Eliot B. Moss. Working with persistent objects: To swizzle or not to swizzle. *IEEE Transactions on Software Engineering*, 18(8):657–673, August 1992.

[43] Andreas Paepcke, editor. *Proceedings of the 1993 Conference on Object-Oriented Programming Systems, Languages and Applications.* ACM Special Interest Group on Programming Languages, ACM Press, September 1993. Also SIGPLAN Notices 28(10), October 1993.

[44] Texas Project. Documentation accompanying Texas persistent store release. Retrieved by FTP, October 1993.

[45] Joel E. Richardson. Compiled Item Faulting: A new technique for managing I/O in persistent languages. In Dearle et al. [25], pages 3–16. Proceedings of the 4^{th}International Workshop on Persistent Object Systems.

[46] Joel E. Richardson and Michael J. Carey. Persistence in the E language: Issues and implementation. *Software Practice and Experience*, 19(12):1115–1150, December 1989.

[47] Joel E. Richardson, Michael J. Carey, and Daniel T. Schuh. The design of the E programming language. *ACM Transactions on Programming Languages and Systems*, 15(3):494–534, July 1993.

[48] Daniel T. Schuh, Michael J. Carey, and David J. DeWitt. Persistence in E revisited – implementation experiences. Technical Report TR957, Computer Sciences Department, University of Wisconsin-Madison, 1990?

[49] Marc Shapiro. Structure and encapsulation in distributed systems: The proxy principle. In *Proceedings of the* 6^{th}*International Conference on Distributed Computing Systems*, pages 198–204. IEEE Computer Society Press, May 1986.

[50] Vivek Singhal, Sheetal V. Kakkad, and Paul R. Wilson. Texas: an efficient, portable persistent store. In Antonio Albano and Ron Morrison, editors, *Persistent Object Systems*, Workshops in Computing, pages 11–33. Springer-Verlag, September 1992. Proceedings of the 5^{th}International Workshop on Persistent Object Systems.

[51] Pedro Sousa, Manuel Sequeira, André Zúquete, Paulo Ferreira, Cristina Lopes, José Pereira, Paulo Guedes, and José Alves Marques. Distribution and persistence in the IK platform: Overview and evaluation. *Computing Systems*, 6(4):391–424, Fall 1993.

[52] B. Tangney and D. O'Mahony. *Local Area Networks And Their Applications.* Prentice-Hall (UK), Hemel Hempstead, 1988.

[53] David Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceeding of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 157–167, April 1984. Also SIGPLAN Notices, 19(5).

[54] Paul R. Wilson and Sheetal V. Kakkad. Pointer swizzling at page fault time: Efficiently and compatibly supporting huge addresses on standard hardware. In Luis-Felipe Cabrera and Eric Jul, editors, *Proceedings of the* 2^{nd}*International Workshop on Object-Orientation in Operating Systems*, pages 364–377. IEEE Computer Society, IEEE Computer Society Press, September 1992.