Supporting Persistent C++ Objects in a Distributed Storage System

Anand Ranganathan, Yury Izrailevsky, Sai Susarla, John Carter, and Gary Lindstrom
Department of Computer Science, University of Utah

Abstract

We have designed and implemented a C++ object layer for Khazana, a distributed persistent storage system that exports a flat shared address space as its basic abstraction. The C++ layer described herein lets programmers use familiar C++ idioms to allocate, manipulate, and deallocate persistent shared data structures. It handles the tedious details involved in accessing this shared data, replicating it, maintaining consistency, converting data representations between persistent and in-memory representations, associating type information including methods with objects, etc. To support the C++ object layer on top of Khazana's flat storage abstraction, we have developed a language-specific preprocessor that generates support code to manage the user-specified persistent C++ structures. We describe the design of the C++ object layer and the compiler and runtime mechanisms needed to support it.

1 Introduction

The goal of our research is to develop language, compiler, and runtime mechanisms that make it easy for programmers to develop complex distributed systems. The primary abstraction that we propose using is *shared persistent data.* Over the past two years, we have gained experience with Khazana, a distributed persistent storage system that exports a flat shared address space as its basic abstraction [2, 17]. While Khazana's flat shared storage abstraction works well for some services (e.g., file systems and directory services), it is a poor match for applications that use "reference-rich" data structures or legacy applications written in object-oriented languages. We believe that programmers should be able to write programs in their preferred programming language (e.g., C, C++, or Java) and be provided with mechanisms that make it easy to *store objects persistently* and *share objects between different programs or different instances of the same program.* We use the term "object" here in the generic sense, rather than any specific language's notion of an object. For example, a C programmer may wish to manipulate arbitrary C data structures, which are not "objects" in the OO sense. Towards this goal, we have designed and implemented the Khazana Object Layer (KOLA), which provides an object abstraction on top of Khazana's page-based shared address abstraction. In this paper, we describe the design of a KOLA C++ object layer and the compiler and runtime mechanisms that support it.

Our goals for the object layer include:

- object-based consistency at arbitrary granularity rather than fixed pages;
- memory management on a persistent heap, including transparent retrieval, caching, and update of persistent objects;
- object-grained synchronization (locking and unlocking);
- automatic object loading on access; and
- object polymorphism and run-time type checking.

KOLA consists of two parts, a *language-independent layer* and a *C++ language-specific layer*, as illustrated in Figure 1. The language-independent layer supports basic operations to manipulate arbitrary-sized (not just page-sized) data objects. These operations include mechanisms to allocate/free persistent objects, retrieve objects into virtual memory in a synchronized way, store them persistently, convert persistent references from/to in-memory virtual addresses, and manage an in-memory cache of objects.

By placing the language-independent object manipulation routines in a separate layer, it should be fairly easy to provide support for other object-oriented languages, such as Java. Specifically, the language-independent layer does not address such vital issues as dynamic type identification and checking, object access detection and loading, class inheritance, or transparent concurrency control (via locking). Instead, these issues are handled by each language-specific layer. Our requirements for a language-specific layer include:

email: {anand, izrailev, sai, retrac, gary}@cs.utah.edu

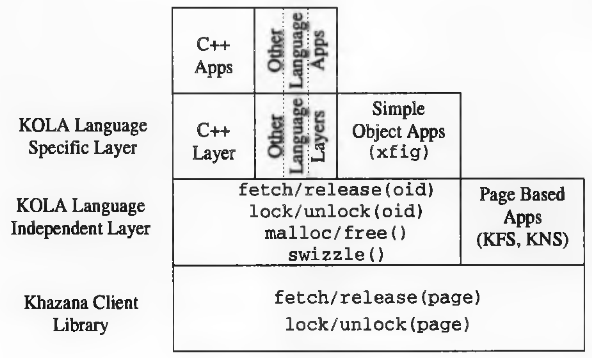

Figure 1: Layered Design of the Khazana Object LAyer (KOLA)

- support for fetching and releasing objects from/to the persistent distributed store;
- the ability to convert an object from its flat form in the persistent store into an in-memory form with references, virtual function tables, etc., and
- support for remote method invocation (RMI) facilities.

We report herein our experience implementing one such layer for C++.

At the core of our current C++ version of KOLA is a language-specific preprocessor. This preprocessor parses conventional class declarations and augments them with overloaded `new` and `delete` operators, special constructors, and synchronization methods. Objects created from these augmented classes can then be stored and retrieved transparently to/from Khazana's shared storage facility. Unlike some previous efforts to develop persistent C++ systems [16, 13], we took great pains not to force programmers to describe their persistent objects using a special data definition language (DDL); instead, we chose to use C++ itself as our DDL. However, the current C++ object layer implementation does not support RMI, and we may be forced to add several keywords to the DDL as part of that effort.

The outline of the rest of the paper is as follows. Section 2 gives a more thorough description of the C++ language-specific layer of KOLA, concentrating on the compiler support utilized by the layer. A qualitative discussion of KOLA in the context of related work is presented in Section 3. We conclude in Section 4.

2 The C++ Language-Specific Layer of KOLA

The primary motivation behind KOLA is to create a system that makes it efficient to manage persistent objects in distributed applications. The idea is not new, and a number of attempts to solve this problem in the past have resulted in implementations that often require the use of a special programming language as the Data Definition Language (DDL) for defining persistent classes. This approach, favored by Shore [1] and COOL [9], requires a significant effort on the part of the user to learn the DDL, apply it consistently, and coordinate its compilation and execution with other program modules written in standard programming languages like C++. Needless to say, porting existing applications on top of such systems can be very tedious.

We have adopted a different approach, also used in Objectstore [8] and Goofie [10], that allows the use of a standard programming language as the DDL. We then utilize a special-purpose preprocessor that parses through the class declarations and converts them into a persistable format. Various language-specific layers can be developed to provide bindings for different object-oriented languages; currently we have implemented a binding for C++ only. Another essential feature of our implementation is compiler independence, since the declaration code and the supporting code generated by the preprocessor are fully compliant with standard C++. The rest of this section discusses how the C++ binding utilizes the preprocessor and interfaces exposed by the language-independent layer to enable C++ programmers to write programs that use familiar object-oriented idioms.

There are two sets of requirements for a language-specific layer. The first set of requirements ensure that a programmer can continue to use familiar C++ idioms. To this end, we wanted a language-specific layer that:

- provided familiar hooks for a programmer to create and initialize objects in the shared Khazana space,

- supported familiar means of access to objects stored in Khazana, in particular the ability to follow pointers and references to objects in Khazana as if they were really in local memory,
- provided a means for concurrency-control between objects,
- provided a means to implement Remote Method Invocation (RMI), and
- required few changes to with legacy code to work.

The second set pertains to the expectations concerning the language-independent layer. Essentially, this refers to that component of object-support that has been deferred by the language-independent layer to a higher layer for want of complete knowledge at the lower layers. This includes facilities to convert objects between their Khazana representation and their in-memory representation (e.g., with vtbl pointers) and transmit locking information to the independent layer so that the language-independent layer can lock the objects in the appropriate mode.

To meet these requirements, the language-specific layer must have access to the definitions of the classes and objects that it is expected to load. In addition, it must be able to insert hooks in these classes and objects so that it is able to provide transparent access to the objects. In fact, in our design, the programmer is oblivious to whether the object with which they are dealing is a local object or one obtained from the shared, distributed Khazana space.

To write a C++ program that uses KOLA, a programmer first declares the classes whose objects will be stored and manipulated by Khazana. Running this header file through our C++ pre-processor produces a set of header and C++ files that are used to support these classes. In addition, the pre-processor adds several non-virtual methods to the classes that have been declared. Some of these functions will be transparent to the programmer. Some others, like the `lock` and `unlock` methods, are provided for programmers to insert in their code to implement concurrency control, when and where appropriate. Our C++ language-specific layer is essentially the C++ preprocessor supplemented by the supporting class methods and other code that it generates.

The set of support files produced by the preprocessor extends a number of object-oriented features of C++ to the persistent distributed environment of Khazana. It supports automatic class loading, which is accomplished at compile time. Persistent object creation and management is done transparently by utilizing special constructors, overloaded new and reimplemented `delete` operators. The C++ layer appends a special type information tag to each instance of an object stored in Khazana, and uses it for object loading and dynamic type checking. Persistent references (Object IDs, or oids) are generally hidden from the programmer and are swizzled and unswizzled automatically through the use of smart pointers. Concurrency control handlers are generated by the preprocessor for each class. Finally, the preprocessor will generate additional code to support RMI in Khazana, which has been designed but is currently not supported. The rest of this section discusses how the C++ layer provides the various features mentioned above. It also presents a more elaborate description of the C++ preprocessor.

2.1 Class Loading

In our implementation, class usage detection and loading occurs at compile time. Instead of storing the class metadata (including the member and reference layout, method information, and class type and hierarchy information) as a separate persistent schema object, classes are linked at compile time. The C++ pre-processor parses user-specified class declarations and adds several methods that allow transparent object loading and synchronization. However, such changes do not alter the class hierarchy, so the programmer does not see any semantic difference in the way that the classes behave. This approach requires that the application retrieving an object has all class information linked into the application. There have been solutions that load objects belonging to classes about which incomplete information is available [3], but we do not currently provide any such mechanism.

If class metadata were platform-independent, we could store class information in KOLA, and thus dynamically load the class on demand. Indeed, this scheme will work for a Java binding, but it will not work for C++ due to the fact that the C++ classes have different formats on different platforms or when generated by different compilers.

2.2 Object Loading

The Khazana representation of an object, currently, is the same as its in-memory representation. This is, however, only an implementation choice of convenience, and not one dictated by the design. Our design does allow for the Khazana representation to be different from the in-memory representation.

```
   // Language-independent layer code
   ...
   obj_fetch(kh_addr, &obj);
   (*reinit_callback)(obj);
   ...

// C++ layer code
ObjectReinit(void *mem)
{
    //Find type of the object.
    KhDummyClass _kh;
    switch(type) {
        case TYPE_CLASS_A:
             obj = new (obj) ClassA(_kh);  return;
        ...
    }
}

void* ClassA::operator new(size_t size, void *mem) { return mem; }

ClassA::ClassA(KhDummyClass& _kh):
        Component1(_kh), Component2(_kh) { }
```

Figure 2: Example Use of Callback Mechanism

Objects are brought in from Khazana to virtual memory at the time of first access. Objects in Khazana are addressed by a unique Object-ID (oid). Currently, an oid is the same as the 128 bit Khazana address where the object has been stored. However, the language-independent layer may choose to bring in more objects than the one requested, e.g., to derive caching benefits. All objects that are brought in to local memory must be reinitialized, which is done via a callback mechanism. The language-specific layer registers a callback that is called any time a new object is mapped into local memory. This function reinitializes the object (e.g., setting up vtbl pointers) so that it resembles an object of the same class that has been created locally. Similarly, the C++ layer provides a callback that is called every time an object is flushed to Khazana. This design allows any conversion between the in-memory representation and the persistent (unswizzled) Khazana representation to be made as lazily as possible. As an optimization, objects can be fetched and converted before they are referenced if the programmer (or compiler) determines that access to a particular object is highly likely. Currently, we do not convert objects to/from a canonical Khazana form, a design decision made to simplify the implementation.

2.2.1 Object Reinitialization

Every object in Khazana is identified by an object ID (OID) and associated with a typeid. In our current design, typeids are 32-bit integers allocated on a per-application basis by the pre-processor, which means that different applications using the same class library may be unable to interoperate. Future versions of the C++ layer will make typeids unique across the entire Khazana space to eliminate this problem. The typeid is stored in Khazana at the time the object is created as part of the object's metadata, along with other supporting data as the object's size.

When the language-independent layer invokes the callback function supplied by the C++ layer, the callback function checks the typeid to determine the type of the object and reinitializes it appropriately. This reinitialization is performed using a `new` operator with placement syntax on the memory area occupied by the object, as shown in Figure 2. In this example, `obj` represents an object that has been fetched from Khazana. The language-independent layer invokes the callback function for reinitialization of the object that has been fetched from Khazana's persistent store. The C++ layer callback function `ObjectReinit` finds the type of the object and invokes a special version of `new` on that area of memory. This invocation of `new` does nothing to the object's data values, but it has the side effect of setting up compiler-dependent pointers (e.g., the vtbl) in the object. It also calls a benign constructor on the object and all of the embedded objects in this object. Both the overloaded `new` operator and the special constructors are generated by the C++ pre-processor.

```
extern __KhDummyClass __kh;  //Declared in a standard place.
class Foo {
    ...
    public:
        /* Persistent new */
        void* operator new (size_t size, __KhDummyClass __kh) {...}
        /* Regular new */
        void* operator new (size_t size) {return ::new(size);}
        void  operator delete (void* local_ref) {
            if (local_ref is known)  deallocate(local_ref);
            else ::delete local_ref;
        }
        ...
};

int main() {
    Foo *local_foo = new Foo;            //Allocate a regular Foo.
    Ref<Foo> shared_foo = new (_kh) Foo; //Allocate a shared Foo.
    ...
    delete local_foo;
    delete shared_foo;
}
```

Figure 3: New and Delete Operators for Objects to be Stored in KOLA

2.3 Object Creation

Any program built on top of Khazana is likely to use persistent objects stored in the shared address space, as well as purely local objects. However, doing all heap allocation in Khazana, even for temporary data structures, would have a prohibitive performance overhead. Thus, the programmer is given the choice of designating certain object references as instances of persistent Khazana objects, while others can be instantiated and treated as traditional C++ objects residing exclusively in the local virtual memory of the application, and only for the application's lifetime.

To do this, a programmer needs some means of distinguishing between local instantiations and "shared instantiations," preferably with as few changes to the code as possible. In our design, persistence is designated by declaration (i.e., fixed at object creation time). We achieve this by overloading the `new` operator to handle instantiations in Khazana space. These operators are added to the class by the C++ pre-processor. C++ does not permit overloading of the `delete` operator, but we can implement it to discriminate between local instantiations and shared ones and act appropriately. Figure 3 gives an example of a class that has been declared for storage in KOLA and has been processed by the C++ preprocessor.

2.4 Pointer Swizzling

Each object in KOLA is addressed by a unique oid. Objects might hold references to other objects via their oids. The C++ layer is responsible for transparently converting oids into virtual memory pointers, which it achieves using smart pointers.

We implement smart pointers using a template class, called `Ref<T>`, where `T` is the type of the object to which the reference points. Smart pointers have been discussed previously, so we will not go into the details of the implementation here [4]. The overloaded dereference operators enable us to trap object access and thereby transparently convert the oid into a local memory reference. This may involve fetching and reinitializing the object.

While smart pointers ensure that programmers can use pointers to KOLA objects just as they would use pointers to regular objects, there is some performance overhead. In particular, every pointer access involves a table lookup. Programmers can alleviate this problem by caching the pointer to the local copy of the object, rather than always accessing the object's contents via its persistent reference (smart pointer). While an object is locked, it is guaranteed not to move in local memory, so during this time the pointer value may be cached and a per-access table lookup can be avoided.

2.5 Support for Remote Method Invocation

The basic programming abstraction of Khazana/KOLA is that of shared data/objects. The "natural" way for different instances of a program (or different programs) to manipulate a shared object is to lock it, which has the side effect of retrieving the most up to date version of the object's contents, access it locally, and then unlock, which has the side effect of making the local modifications visible to other applications accessing the object. There are, however, instances where a programmer might prefer to access an object wherever it currently resides, e.g., because the object is large and moving it is inefficient or because there are security concerns that restrict an object's ability to move. For such instances, KOLA is designed to allow programmers to specify that a particular object's methods should be invoked remotely, so-called remote method invocation (RMI). Note that RMI is not currently implemented, so this section reports on our RMI design only.

KOLA builds on the underlying Khazana coherence management hooks to implement RMI. In particular, we extended Khazana's notion of coherence to include a coherence policy whereby nodes could send "update" messages to remote nodes with copies of a particular object. These messages are propagated to the "coherence manager" registered for a particular object. To implement RMI, KOLA passes a marshaled version of the method parameters to one or more remote coherence managers for the object, which interpret these "update" messages as RMI invocations. The result of the method invocation is return to the invoking application via a similar mechanism. A description of Khazana's coherence management mechanisms sufficient to explain adequately the nuances of how the RMI mechanism works is beyond the scope of this paper, but is described elsewhere [17]. However, our C++ preprocessor does play a role in our RMI implementation. Specifically, for each class that has been annotated to be "immobile," the preprocessor generates two classes: "stubs" and "real classes," similar to a normal RPC stub generator. Which sets of routines (stub or real classes) gets linked to an application depends on whether or not it indicated that the objects were immobile (stubs) or normal (real classes).

2.6 The C++ Pre-Processor

The C++ preprocessor is the centerpiece of the language-specific layer. While it was clear that a DDL was required for defining persistent classes, we chose C++ as our DDL for two reasons. First, we wanted to be able to convert legacy applications to persistent distributed versions of those applications with little source code change, and without having to reverse-engineer DDL declarations from the existing source. Second, we wanted programmers to have the convenience of expressing themselves in the language that they were going to program in rather than a new DDL. The C++ preprocessor parses class declarations and augments them with an overloaded `new` operator and a new `delete` operation, the special constructors, and synchronization methods, as described above.

To implement the preprocessor, we used a modified version of Jim Roskind's C++ acceptor [14]. Given regular C++ class declarations (usually in a header file), the preprocessor generates a new set of declarations that are compatible with KOLA, and may be used by the programmer for further application development. A set of routines containing internal swizzling and type-identification and checking routines are also generated by the preprocessor based on the class hierarchy specified by the programmer.

We are able to process most standard C++ declarations, with a few exceptions such as anonymous classes/structs/unions. Also, our implementation is unable to deal with classes that have static data members. This is due to what we consider to be the unclean semantics of C++ static data members that does not lend itself to persistence. However, we do not believe this is a significant problem, because programmers can create static data members by creating objects of singleton classes. In addition, our in-memory representation of object is the same as the persistent one in our current implementation, which means that all applications that wish to access a particular object in Khazana should use compilers that agree on object layout. This limitation arises due to the absence of an object layout standard for different C++ compilers. KOLA does not impose any specific object layout. However, we could work around this layout limitation in future versions of the C++ layer by storing type descriptor *objects* in KOLA instead of simply a typeid. Finally, although we attempt to minimize the amount of source code that needs to be modified in legacy applications, we require that declarations for references to shared or persistent objects be converted to smart pointers, which can be tedious in large legacy codes.

3 Related Work

There is a long history of support for object-oriented languages built upon persistent storage platforms. This work can be divided into essentially three categories: (i) object oriented database systems, (ii) generic persistent stores, and (iii) distributed object systems. Since generic persistent stores are currently of little interest in large scale application development, albeit making somewhat of a comeback in the Java world, we will focus on the first and last category of related work. We focus on several typical examples of each category of related work, rather than attempting a comprehensive survey.

The ObjectStore OODBMS by ObjectDesign, Inc. is our representative object-oriented database [8]. Since it is a true DBMS, it has functionality outside the design goals of Khazana, such as recovery and features deriving from logging (e.g., versioning and multi-valued concurrency control). In areas where direct comparison is appropriate, Khazana shares many of ObjectStore architectural features, such as coherent client caching and gathering of class ("schema") information by preprocessing. Like Khazana, ObjectStore makes extensive use of C++ extensibility features, such as smart pointers for address space expansion and dereferencing of inter-segment pointers. Like Khazana, ObjectStore presumes a closed world where all persistent object classes must be known at compile time. Persistent references are stored in virtual memory (32-bit) format, with segment-based relocation tables to support swizzling.

The Esprit Project's PerDiS system is an excellent comparison with Khazana in the distributed object arena [5]. Unlike Khazana and ObjectStore, PerDiS obtains type information by compiler modification (gcc). Unlike both, persistence in PerDiS is by reachability rather than declaration — a requirement with profound design and implementation consequences, including distributed garbage collection. The latter is achieved by means of the Stub/Scion Pair technique developed by the Projet SOR at INRIA, which logs exported references as roots for local GC, and supports a distributed GC algorithm for deleting these roots when safe. PerDiS provides two concurrency control modes: explicit locking and locking by touch (the "compatibility interface"), and anticipates customizable concurrency control modes. PerDis provides precise type information with every reference – an obvious requirement for safe and thorough GC. Both pessimistic and optimistic concurrency control are provided.

Distributed object systems (e.g., Clouds [12], Emerald[7], Monads [6], and CORBA [11]) provide uniform location-transparent naming and access to heterogeneous networked objects. In these systems, services can export a well-typed set of object interfaces to clients, which can invoke operations on service objects by *binding* to a particular instance of a service interface and invoking said service. The addition of *object brokers*, such as are present in CORBA [11], provided a degree of location transparency, and the addition of an object veneer made it easier for servers to change their internal implementation without impacting their clients. However, these systems proved to be effective primarily when used to support the same type and granularity of services previously supported by ad hoc client-server systems: large servers exporting coarse grained operations on large datasets (e.g., mail daemons and file servers). These are too heavyweight for fine- to medium-grained language objects.

COOL [9] provides support for medium- to coarse-grained objects with a similar layering to that of KOLA namely generic and language-specific runtimes. In COOL, the generic run-time obtains object layout information from the language runtime through an upcall mechanism. Also it eagerly swizzles all embedded references inside the object at loading time. In contrast, KOLA's language-independent layer provides swizzling routines and lets the language-specific layer choose lazy vs eager swizzling. Our current C++ implementation does lazy swizzling using smart pointers and does not need to locate embedded references at initialization time. In COOL, all method invocations on an object, including local references go through its interface object, which may be costly for fine-grained objects. Lastly, COOL object loading/unloading mechanism is closely tied to Chorus virtual memory mechanisms, while we tried to avoid making KOLA rely on virtual memory mechanisms for portability reasons.

Texas [16] provides persistence to C++ objects by explicit request. It extracts objects' type information (such as location of embedded references, including vtable pointers) from the compiler-generated debugging information in the application's executable code. They use this information to perform object-reinitialization (setting up vtable pointers) and other pointer swizzling. In contrast, we use a special constructor for object-reinitialization at loading time and language-level smart pointers instead of virtual memory techniques for pointer swizzling.

4 Conclusions

As has been demonstrated in this paper, special purpose compilers can be successful in supporting persistent and shareable objects in distributed shared storage environments. We have described the design and implementation of a C++ preprocessor used by the C++ object layer of Khazana, a distributed persistent storage system that exports a flat shared address space as its basic abstraction. The object layer handles the tedious details required to manipulate non-page-grained data objects on top of Khazana, as well as language-specific issues such as initializing vtbl entries when objects are fetched from the persistent store or from across the network. With this support and the underlying Khazana distributed shared storage facilities, programmers are able to program using their preferred language abstractions while at the same time being relieved from many of the most difficult tasks associated with programming distributed systems.

At the core of our current C++ version of KOLA is a language-specific preprocessor that parses conventional C++ class declarations and augments them with overloaded `new` and `delete` operators, special constructors, and synchronization methods. The preprocessor also generates support routines that implement transparent pointer and object swizzling and unswizzling, class loading, and type checking. Objects created from these augmented classes can then be stored and retrieved transparently to/from Khazana's shared storage facility, and will soon be able to be invoked remotely using built-in RMI facilities. We believe this combination of an easy-to-use preprocessor that uses C++ itself as the data description language and a distributed persistent storage facility that handles distribution, replication, and security issues gives programmers a powerful tool with which to develop applications.

Work is continuing on several fronts. As mentioned above, RMI is not yet supported, and may require us to augment the preprocessor's DDL with additional keywords. We are implementing several C++ applications, in particular an IMAP server, a version of Mozilla (the public Netscape browser) in which client HTTP caches are shared, and a groupware version of `xfig`. We are also making improvements to the underlying Khazana infrastructure with an eye towards improving performance (e.g., we recently added support for "update" messages in Khazana's coherence module, which we plan to leverage to implement RMI). Finally, adding support for other object-oriented languages has the potential of greatly increasing the range of applications that can take advantage of Khazana's distributed persistent capabilities, and thus we are in the process of designing a Java layer for KOLA.

References

[1] M. Carey, D. Dewitt, D. Naughton, J. Solomon, et al. Shoring up persistent applications. In *Proceedings, of the 1994 ACM SIGMOD Conf*, May 1994.

[2] J. Carter, A. Ranganathan, and S. Susarla. Khazana: An infrastructure for building distributed services. In *Proceedings of The Eighteenth International Conference on Distributed Computing Systems*, pages 562–571, May 1998.

[3] C. F. Clark. *The Dynamic Expansion of Class Hierarchy*. PhD thesis, University of Utah, 1995.

[4] M. Ellis and B. Stroustrop. *The Annotated C++ Reference Manual*. Addison Wesley, 1990.

[5] P. F. et al. PerDis: Design, implementation, and use of a PERsistent DIstributed Store. Technical Report RR 3525, INRIA, Oct. 1998.

[6] D. Henskens, P. Brossler, J. Keedy, and J. Rosenberg. Coarse and fine grain objects in a distributed persistent store. In *International Workshop on Object-Oriented Operating Systems*, pages 116–1123, 1993.

[7] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, Feb. 1988.

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

[9] R. Lea, C. Jacquemot, and E. Pillevesse. COOL: system support for distributed object-oriented programming. Technical Report CS-TR-93-68, Chorus Systèmes, 1993.

[10] R. Mecklenburg, C. Clark, G. Lindstrom, and B. Yih. A dossier driven persistent objects facility. In *Usenix C++ Conference Proceedings*, Apr. 1994.

[11] Object Management Group. The Common Object Request Broker: Architecture and Specification, 1996.

[12] P. Dasgupta et al. Distributed programming with objects and threads in the Clouds system. *Computing Systems Journal*, 3(4), 1991.

[13] T. Printexis, M. Atkinson, L. Daynes, S. Spence, and P. Bailey. The design of a new persistent object store for PJama. In *International Workshop on Persistence for Java*, Aug. 1997.

[14] J. Roskind. A YACC-able C++ grammar and the resulting ambiguities. http://www.empathy.com/pccts/roskind.tar.gz, Aug. 1991.

[15] M. Shapiro, P. Gautron, and L. Mosseri. Persistence and migration for C++ objects. In *Proceedings of the European Conference on Object-Oriented Programming*, July 1989.

[16] K. Singhal, S. Kakkad, and P. Wilson. Texas: An efficient, portable persistent store. In *Proceedings of the Fifth International Workshop on Persistent Object Systems Design, Implementation, and Use*, pages 13–28, Sept. 1992.

[17] S. Susarla, A. Ranganathan, and J. Carter. Experience using a globally shared state abstraction to support distributed applications. Technical Report UU-CS-98-016, University of Utah, Department of Computer Science, Aug. 1998.