Shoring Up Persistent Applications*

Michael J. Carey, David J. DeWitt, Michael J. Franklin,† Nancy E. Hall,
Mark L. McAuliffe, Jeffrey F. Naughton, Daniel T. Schuh, Marvin H. Solomon,
C. K. Tan, Odysseas G. Tsatalos, Seth J. White, Michael J. Zwilling

Computer Sciences Department
University of Wisconsin—Madison
shore@cs.wisc.edu

Abstract

SHORE (Scalable Heterogeneous Object REpository) is a persistent object system under development at the University of Wisconsin. SHORE represents a merger of object-oriented database and file system technologies. In this paper we give the goals and motivation for SHORE, and describe how SHORE provides features of both technologies. We also describe some novel aspects of the SHORE architecture, including a symmetric peer-to-peer server architecture, server customization through an extensible *value-added server* facility, and support for scalability on multiprocessor systems. An initial version of SHORE is already operational, and we expect a release of Version 1 in mid-1994.

1 Introduction

SHORE (Scalable Heterogeneous Object REpository) is a new persistent object system under development at the University of Wisconsin that represents a merger of object-oriented database (OODB) and file system technologies. While the past few years have seen significant progress in the OODB area, most applications (and application areas) have not chosen to leave file systems behind in favor of OODBs. We feel that more applications could benefit from OODB support, but are impeded by limitations in current technology.

1. Many current OODBs are closed and restricted to one language (most often persistent C++ or Smalltalk), unlike both file systems and relational database systems. Large-scale applications often require multilingual data access.

2. With most current OODBs, application programmers face an "either/or" decision—either they put their data in the OODB, in which case all of their existing file-based applications must be rewritten, or they leave their data in files.

3. Most current OODBs provide a fairly "heavy" solution in the area of transaction management, dictating the adoption of serializability and up-to-the-last-transaction data recoverability.

4. Most current OODBs have strongly client-server architectures, and are thus inappropriate for execution in peer-to-peer distributed systems or on the kinds of high-performance multicomputer hardware needed for certain large scale applications.

The goal of the SHORE project is to provide a system that addresses these issues, thereby enabling "holdout" applications to finally move their data (incrementally) out of files and into a modern persistent object repository. We also expect many current OODB clients to find SHORE to be an attractive alternative.

1.1 EXODUS

Many of us were involved in an earlier object-oriented database effort called EXODUS [CDF+86]. Version 3.0 of EXODUS provides a client-server architecture with page-level locking, log-based recovery based on the ARIES algorithm [FZT+92], and support for multiple servers and distributed transactions. The EXODUS package includes the E programming language [RCS93], a variant of C++ that supports convenient creation and manipulation of persistent data structures. The functionality, performance, robustness, and low cost (free!) of EXODUS has made it a popular piece of software. EXODUS and its associated toolkit have been used in several projects at Wisconsin and elsewhere. Over 350 different groups from over 30 countries have taken copies of it from our ftp site, it is used as the storage manager in the TI Open Object-Oriented Database System, it serves as the storage engine for at least one commercial product (MediaDB, a recently announced multi-media DBMS), and it has been shown to have commercially competitive performance on an OODBMS benchmark [CDN93].

---

*A version of this paper appeared in *Proceedings of the 1994 ACM-SIGMOD Conference on the Management of Data*, Minneapolis, MN, May 1994. This research is sponsored by the Advanced Research Project Agency, ARPA order number 018 (formerly 8230), monitored by the U.S. Army Research Laboratory under contract DAAB07-91-C-Q518.

†Current Address: Department of Computer Science, University of Maryland, College Park, MD

Nonetheless, EXODUS suffers from several limitations shared by many current persistent object stores. An exploration of these limitations may help to explain the motivation for SHORE.

EXODUS storage objects are untyped arrays of bytes; correct interpretation of their contents is the responsibility of application programs. Although E allows instances of any C++ type to be stored in the database, no type information is stored. This "compile-time" approach to data types has several disadvantages including the following:

- It is too easy to access objects under the wrong type, because of programming or configuration errors such as version mismatch.
- Restricting type support to the compiler locks users into single-language solutions.
- Sharing data between applications is difficult.
- Lack of stored types prevents the DBMS from providing such facilities as support for heterogeneous hardware platforms, data browsers, or garbage collectors.

At the time we designed EXODUS, we felt there was too much variability in type systems to legislate a common solution. Since then, there has been a growing consensus on the level of type support that an OODBMS system should provide [Cat93].

A second limitation of the EXODUS storage manager (ESM) is its client-server architecture. Users have constructed database servers or object servers as EXODUS client processes, leading to the "client-level server" problem illustrated in Figure 1. Even a query-shipping (as opposed to data-shipping) SQL server would be difficult construct efficiently with the existing software base. In contrast, a more open architecture would have allowed clients to customize the ESM server process directly. The ESM process architecture also fails to support a clean mapping onto parallel processors such as the Intel Paragon or IBM SP/2. Although one could simply run an EXODUS server on each node with mass storage attached, support for distributed transactions is not sufficient; efficient parallelism also requires the availability of extensive server-to-server communication facilities.

A third limitation of EXODUS is its lack of support for access control. As with other aspects of the system, our original thinking was that different clients might wish to implement very different protection models, and thus we provided no built-in protection support. Furthermore, EXODUS allows client processes to manipulate objects directly in cached copies of database pages, so an errant pointer can destroy not only client data but also metadata, rendering the entire database unusable. The original design of EXODUS

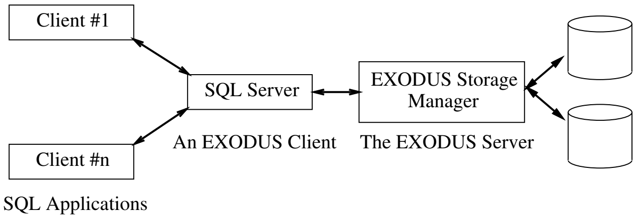

Figure 1: The client-level server problem.

envisioned client processes as being database systems and object servers (i.e., other trusted software layers). SHORE aims to support environments in which a single storage volume may be shared by mutually mistrusting applications.

Finally, while EXODUS objects are similar to Unix files (they are untyped sequences of bytes), the interface for manipulating them is completely different. As a result, existing applications built around Unix files cannot easily use EXODUS.

The design of SHORE strives to retain the good features of the EXODUS Storage Manager (such as transactions, performance, and robustness) while eliminating some of these limitations.

1.2 How SHORE differs from EXODUS

Each object in SHORE contains a pointer to a *type* object that defines its structure and interface. The SHORE Data Language (*SDL*) provides a single language-neutral notation for describing the types of all persistent data[1]

SHORE's process architecture is different from that of EXODUS in two key ways. First, SHORE has a symmetric, peer-to-peer structure. Every participating processor runs a SHORE server process regardless whether it has local disks. A client process interacts with SHORE by communicating with the local SHORE server (see Figure 2). The design is scalable; it can run on a single processor, a network of workstations, or a large parallel processor such as the Intel Paragon or IBM SP/2. Second, SHORE supports the notion of a "value-added" server. The server code is modularly constructed to make it relatively simple for users to build application-specific servers without facing the "client-level server" problem. For example, the Paradise project [DLPY93] is already using the SHORE server to build a geographic information system.

Finally, SHORE is intended to be much more of a complete system than ESM. In addition to a more flexible process structure and support for typed objects, SHORE provides other services that end users should find attractive, including a name space and access-

[1] SDL is very closely related to ODL, a data definition language recently proposed as a standard by ODMG, an OODB vendor consortium [Cat93].

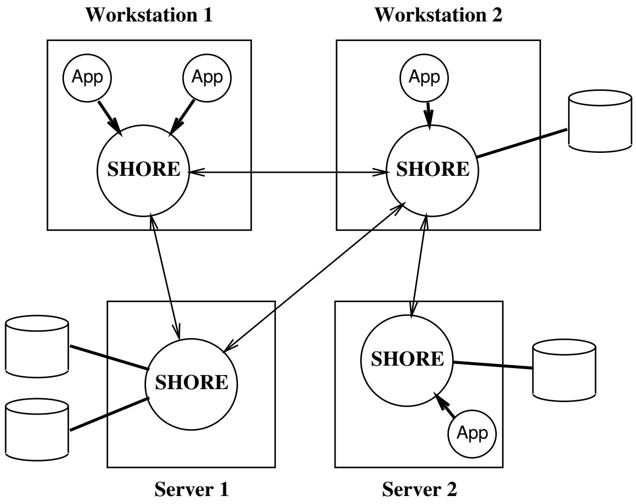

Figure 2: The SHORE process architecture.

control model similar to Unix, a Unix-compatible interface for legacy software tools, openness in the area of language bindings, and traditional database services such as associative data access, indexing, and clustering.

The remainder of this paper is organized as follows. Section 2 provides an overview of the services provided by SHORE, including both its file system and database features. The SHORE process architecture is described in Section 3. Section 4 describes the tools that we are developing for writing parallel, object-oriented applications using SHORE. We conclude in Section 5. The software described here is not simply "paperware." Much of the basic SHORE software is already operational; the system is now sufficiently complete that its C++ binding can run much of the OO7 benchmark [CDN93] in both client-server and parallel environments. We are expecting a full release of Version 1 of SHORE in mid-1994.

2 Basic SHORE System Concepts

As a hybrid system, SHORE may be described as a file system augmented with database features or a DBMS with file-system features. In this section, we will describe the basic features of SHORE, explaining how it combines important ideas from these two areas in order to arrive at a system capable of addressing the variety of application requirements discussed in the introduction.

2.1 The Big Picture

SHORE is a collection of cooperating data servers, with each data server containing typed persistent objects. To organize this universe of persistent SHORE objects, a Unix-like namespace is provided. As in Unix, named objects can be directories, symbolic links, or individual (typed) objects (the counterpart of Unix "plain" files). Unlike Unix, SHORE allows each object to be accessed by a globally unique Object Identifier (OID) that is never reused. SHORE also introduces a few new types of objects, including *types* and *pools*, as described in more detail in Section 2.3. The type system for SHORE objects is language-neutral, supporting applications in any programming language for which a language binding exists. For objects whose primary data content is textual or untyped binary data, Unix file system calls are provided to enable legacy applications (such as existing language compilers or CAD tools) to access their data content in an untyped manner. SHORE is structured as a peer-to-peer distributed system; each node where objects are stored or where an application program wishes to execute contains a SHORE server process that talks to other SHORE servers, interfaces to locally executing applications, and caches data pages and locks in order to improve system performance.

2.2 SHORE Object Basics

The SHORE object model, like many database object models, consists of *objects* and *values*. Every persistent datum in SHORE is an object, and each object has an identity denoted by a unique object identifier or *OID*. Structurally, an object is a container for a value; the value can be simple or structured, and may include references to (typed OIDs of) other objects. Every value has a type, as does every object. Behaviorally, each object has a set of methods through which its contents can be accessed and manipulated. The internal structure and methods available for a given object are dictated by the object's type, referred to as its *interface type*, and every SHORE object is tagged with a reference to a type object that captures this information.

A SHORE object is much lighter-weight than a Unix file, but it may still be too heavy to support fine-grained data structures that are customarily represented as linked lists, trees, or other graph structures in non-persistent programs. To support the flexibility of dynamic structures with the efficiency of (logically) contiguous blocks on secondary storage, SHORE allows each object to be extended with a variable-sized *heap* (see Figure 3). The *core* of an object is described by its type. The heap is used by the system to store variable-sized components of its value such as strings, variable arrays, and sets. The heap can also contain *dynamic values* which are similar to "top-level" objects, but do not have independent identity (for example, when the object is destroyed, all of its dynamic values are destroyed as well). Dynamic values can be linked together with *local references*, which are stored on disk as offsets from the start of the heap, but are swizzled in memory to actual memory addresses. The O2 commercial OODBMS [Deu91] provides a related facility with its objects/values distinction; the main difference is that in O2 the encapsulated values must form a set, list, or array, whereas in SHORE the heap can contain an arbitrary data structure. With demand-paging support for very large objects, each object heap closely resembles a small Object Store database [LLOW91].

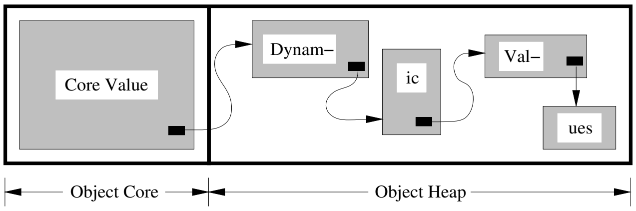

Figure 3: SHORE object structure.

2.3 File System Features

From a file system standpoint, SHORE provides two major services. First, to support object naming and space management in a world with many persistent objects, SHORE provides a flexible object namespace. Second, to enable legacy Unix file-based applications to continue to exist while new SHORE applications are being developed, mechanisms are provided that permit SHORE object data to be accessed via Unix file system calls.

2.3.1 SHORE Object Namespace

SHORE provides a tree-structured, Unix-like namespace in which all persistent objects are reachable, either directly or indirectly, from a distinguished root directory. By doing so, SHORE gives users a framework in which to register both individual persistent objects and the roots of large persistent data structures, a framework that provides a much richer naming environment than the single-level "persistent root" directory found in EXODUS and most other current OODBs.[2] The realization of this framework involves extending the set of familiar Unix object types (*directories*, *symbolic links*, and "regular files") with *cross references*, *pools*, *modules*, and *type objects*.

SHORE directory objects provide the same facilities as Unix directories. Familiar Unix concepts such as path name, subdirectory, parent directory, link (both hard and symbolic), and root directory are all defined as they are in Unix [RT74]. As in Unix, a directory is a set of $\langle name, OID \rangle$ pairs. The *OID* can refer to any other SHORE object, but the system maintains the Unix invariant that the set of directories forms a single rooted tree. Directories and the objects they contain are called *registered* objects. Each registered object contains a superset of the Unix attributes: ownership, access permissions, and timestamps.[3] To support lighter-weight objects, SHORE introduces a new kind of (registered) object called a *pool*. Members of a pool, called *anonymous* objects, are clustered near each other on disk and share most of the Unix attributes (ownership, etc.) with the pool. Anonymous objects do not have path names, but they can be accessed by OID like any other object. There is also an operation to enumerate the contents of a pool (which can be accessed by OID or path name). The *registered* property is orthogonal to type: Any type of object can be created either in a pool (as an anonymous object) or in a directory (as a registered object). We expect that in a typical SHORE database, the vast majority of objects will be anonymous, with a few registered objects serving as roots or entry points to graphs of anonymous objects.

To preserve the invariant that all objects are reachable from the root of the directory system, SHORE imposes different deletion semantics on registered and anonymous objects. As in Unix, a registered object is not explicitly deleted; it is reclaimed by the system when its link count (the number of directory entries referring to it) drops to zero. An anonymous object can be deleted at any time, but a pool can only be deleted when it is empty. An OID is thus a "soft" reference, in that it may dangle if the object to which it refers is deleted. (Since OIDs are never reused, however, it will never accidentally capture a new object.) Since OIDs can be stored in the contents of arbitrary objects, any stronger integrity guarantee would be impractical to enforce.

SHORE introduces three more fundamental kinds of objects, *modules*, *type objects*, and *cross references*. Modules and type objects are similar to pools and anonymous objects, respectively, but have different deletion semantics to preserve the existence dependency from objects to their types. Cross references are similar to symbolic links in that they provide a way to insert an alias for an object into the directory name space. While a symbolic link contains a path name for a registered object, a cross reference contains the OID of an arbitrary object. Cross references, like symbolic links, are "soft" (permitted to dangle). They are intended primarily for the Unix compatibility feature described in the following section.

Figure 4 illustrates these concepts. The directory `/u/smith` contains the entries `project`, `doc`, and `pool1`, referring to another directory, a cross reference, and a pool, respectively. The registered object `/u/smith/project/entries` contains pointers to members of `pool1`. It might be some sort of application-defined "directory" of entry points to a data struc-

[2] Many of the commercial systems use a tree-structured name space for naming *databases*, but not for naming or organizing individual persistent objects or collections.

[3] The semantics of timestamps are slightly different from those of Unix in order to make them efficiently maintainable while retaining their usefulness to applications that rely upon them.

ture. The symbolic link `/u/smith/project/README` is an alias for the cross reference `/u/smith/doc`, which is itself an alias for a member of `pool1`. An attempt to access either of these path names through the Unix compatibility interface will resolve to that anonymous object.

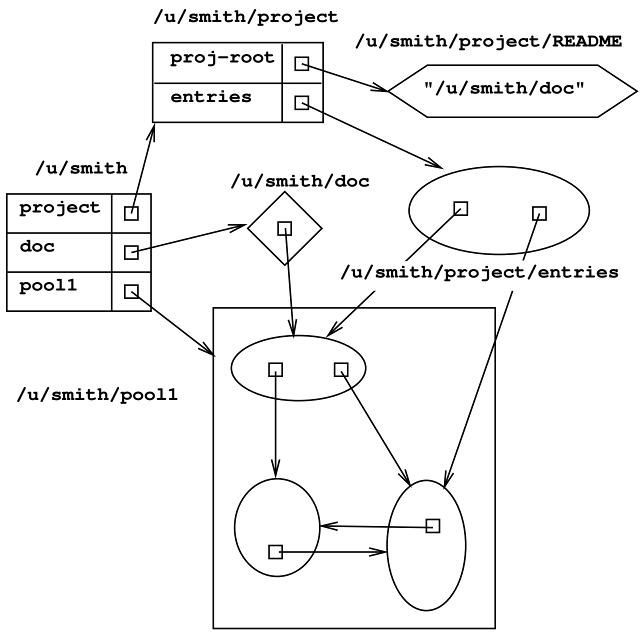

Figure 4: The SHORE name space.

2.3.2 Legacy Unix Tool Support

While SHORE provides a much richer environment than traditional file systems, there are many situations where tools designed to be used on files need to be invoked on database objects. A typical example is provided by the CAPITL project [AS93], which uses EXODUS. CAPITL improves on current software-development environments by maintaining a rich set of attributes and relationships for each object in its repository (program sources, object files, specifications, documents, etc.) It represents each object as a directed graph, with intra- and inter-object links represented by OIDs. While tools developed as part of CAPITL take full advantage of this rich structure, it is occasionally necessary to invoke existing tools such as compilers or editors on objects stored in the database. Three possible approaches were to rewrite the tools to access CAPITL objects, to copy the contents of an object to a file before operating on it (and copy back the results), or to keep the contents permanently in files, storing only metadata and file names in the CAPITL database. All of these approaches are unsatisfactory for various reasons. The solution found for CAPITL, which we have generalized and expanded in SHORE, is to provide a special *Unix compatibility* feature. Each SHORE object may optionally designate a range of bytes as its *text* field. A compatibility library provides versions of Unix system calls such as `open`, `read`, `write`, and `seek`, interpreting pathname arguments in the SHORE name space and satisfying requests by fetching or updating the text field of objects. Registered objects without text fields behave like `/dev/null` (they read as zero length and ignore attempts to change them). Anonymous objects can be accessed via cross references.

For applications that cannot even be re-linked, we have constructed an NFS file server [SGK+85]. An entire subtree of the SHORE name space can be "mounted" on an existing Unix file system. When applications attempt to access files in this portion of the name space, the Unix kernel generates NFS protocol requests that are handled by the SHORE NFS value-added server.

2.4 Object-Oriented Database Features

As we mentioned in Section 1.1, one important motivation for SHORE was to rectify some of the shortcomings of EXODUS, many of which are shared by other existing object-oriented databases. Access control and name space limitations were addressed in the previous section. Process structure is addressed in Section 3. In this section we describe the design and implementation of the SHORE type system and indicate how it supports hardware and language heterogeneity.

2.4.1 The SHORE Type System

The SHORE type system is embodied by the SHORE Data Language, *SDL*, the language in which SHORE types are defined. SDL is quite similar in nature to the Object Definition Language (ODL) proposal from the ODMG consortium [Cat93], which is descended from OMG's Interface Description Language (IDL), a dialect of the RPC interface language used in OSF's Distributed Computing Environment (DCE). Our work on SDL started at roughly the same time as ODMG's work, and we also used OMG's IDL as a starting point. We have been following the development of ODL, but we had to proceed as well rather than waiting for ODMG to complete their work. (At this time, the ODMG standards are still only in the late paper design stage, and portions are not yet entirely clear or internally consistent.) The goals of ODMG are also somewhat different from ours. They concentrate on a standardized interface to existing C++ oriented OODBs, while our focus has been support for inter-language object sharing within a large namespace of objects.

All objects are instances of *interface types*, types constructed with the *interface* type constructor. Interface types can have methods, attributes, and relationships. The attributes of an interface type can be of one of the primitive types (e.g., integer, character, real), or they can be of constructed types. SHORE provides the usual set of type constructors: enumerations, structures, arrays, and references (which are used to define relationships). In addition, SHORE provides a variety of *bulk types*, including sets, lists, and sequences, that enable a SHORE object to contain a collection of references to other objects. Finally, SHORE provides the notion of *modules*, to enable related types to be grouped together for name scoping and type management purposes. To provide a brief taste of SDL, Figure 5 shows how one of the OO7 benchmark [CDN93] types can be defined.[4]

```
module oo7 {
  const long TypeSize = 10;
  enum  BenchmarkOp { Trav1, Trav2, Trav3, etc };

  // forward declarations
  interface Connection;
  interface CompositePart;

  interface AtomicPart {
  public:
    attribute char            ptype[TypeSize];
    attribute long            x, y;
    relationship set<Connection>
      to                      inverse from;
    relationship set<Connection>
      from                    inverse to;
    relationship ref<CompositePart>
      partOf                  inverse parts;
    void swapXY();

    long traverse(in BenchmarkOp op,
          inout PartIdSet visitedIds) const;
    void init(in long ptId,
          in ref<CompositePart> cp);
  };
  // Connection, CompositePart, and other types
}
```

Figure 5: Contents of the file `oo7.sdl`.

2.4.2 SHORE Language Bindings

SHORE is intended to allow databases built by an application written in one language (e.g., C++) to then be accessed and manipulated by applications written in other object-oriented languages as well (e.g., CLOS). This capability will be important for large-scale applications, such as VLSI CAD; C++ might be used for efficiency in simulating large chips, while CLOS (or perhaps Smalltalk) might be more convenient for writing the associated design-rule checking and user interface code. In SHORE, the methods associated with SDL interfaces can therefore be written using any of the languages for which a SHORE language binding exists. Currently, only the C++ binding is operational, so we will illustrate SHORE's language binding concepts by briefly discussing the SHORE C++ binding.[5]

An application, such as the OO7 benchmark, is created as follows. The first step is to write a description of the types in SDL. In our OO7 example, this description is saved in a file called `oo7.sdl`. The next step is to use the SDL type compiler to create type objects corresponding to the new types. The type compiler is a SHORE application that creates type objects from SDL definitions.[6] A language-specific tool (in our case, *c++extract*) is then used to derive a set of class declarations and special-purpose function definitions from the type objects. In our example, this generated code is placed in two files: `oo7.h`, and `oo7.C`. The header file `oo7.h` is included both in the C++ source files that supply the (application-specific) implementation of member functions such as `traverse` and `swapXY`, and in source files that manipulate instances of `AtomicPart`, etc. The OID of the type object is compiled into these files and used to catch version mismatches at runtime.

A fragment of the generated `oo7.h` file is shown in Figure 6. Some of the data member types in Figure 6 correspond directly to SDL types, as C++ (like most languages) offers direct support for those simple types. For SHORE types with no corresponding C++ type, like sets and references, a language-appropriate presentation of the SDL type is generated. For C++, SHORE presents references, sets, and other collection types using pre-defined template classes (parameterized types) such as `Ref` and `Set` in Figure 6. The class `Ref<CompositePart>` encapsulates an OID; C++ overloading features make it behave like a pointer to a read-only instance of `CompositePart`. The class `Set<Connection>` encapsulates a data structure containing a set of OIDS, and it provides member functions that enable its contents to be accessed.

[4]In the interest of brevity, some of the details have been omitted.

[5]Other bindings are planned, of course, but work on them will not begin until SHORE is fully operational and delivering good performance through its C++ binding.

[6]Note, however, that any SHORE application can create type objects. For instance, one could write a graphical schema design tool to create type objects and install them in the database.

```
class AtomicPart {
 public:
  char ptype[10];
  long x;
  long y;
  Set<Connection> to;
  Set<Connection> from;
  Ref<CompositePart> partOf;
  virtual void swapXY();
  virtual long traverse(BenchmarkOp op,
               PartIdSet &visitedIds) const;
  virtual void init(long ptId,
               Ref<CompositePart> cp);
  // Additional SDL-generated members
  // are included here.
};
```

Figure 6: C++ Class generated from `oo7.sdl`.

Given the header file generated by the binder, the application programmer can implement the operations associated with the OO7 interfaces. In the C++ binding, access to simple data members is provided safely through the use of several techniques. As mentioned above, `Ref`-generated classes behave like *read-only* pointers, so information about an atomic part could be printed by a function as follows:

```
void printPart(Ref<AtomicPart> p) {
  cout << "Type " << p->ptype
       << " part at (" << p->x << ","
       << p->y << ")\n";
}
```

This function can directly access the `part_type`, `x`, and `y` data members of an atomic part, but it cannot update them. (Attempts to do so would be flagged as an error by the C++ compiler.) Similarly, member functions that do not update the contents of an object are flagged as `const` in their SDL definition, as illustrated in Figure 5 (and attempts to call a non-`const` member function through a `Ref` are also caught by the compiler).

To modify an object, the C++ application must first call a special generated member function, `update`, which returns a read-write reference. For example, the following code fragment directly exchanges the `x` and `y` attributes of an atomic part:

```
Ref<AtomicPart p> = ... ;
long tmp = p->x;
p.update()->x = p->y;
p.update()->y = tmp;
```

The function `update` coerces the type of `p` from `Ref<AtomicPart>` to (*non* `const`) `AtomicPart *`. It also has the runtime effect of marking the referenced object as "dirty" so that changes will be transmitted to the server when the transaction commits. Since the member function *swapXY* is not declared to be *const* in Figure 6, another legal way to accomplish this exchange would be to define this member function as follows:

```
void AtomicPart::swapXY() {
    long tmp = x;
    x = y;
    y = tmp;
}
```

Given this definition, *swapXY* could then be invoked to do the job.

```
Ref<AtomicPart p> = ... ;
p.update()->swapXY();
```

The SHORE C++ binding implements collection types similarly to C++ OODBs [LLOW91, Obj92, Ont92, Ver92]: A template type such as `Set<Connection>` in Figure 6 contains a member function `members` that returns an iterator. For example, the `printPart` function could be extended to print an atomic part's outgoing connections as follows:

```
void printPart(Ref<AtomicPart> p) {
  cout << "Type " << p->ptype << " part at ("
       << p->x << "," << p->y
       << ") with outgoing connections\n";
  Iter<Connection> m = to.members();
  for (Ref<Connection> c = m.first();
       c!=NULL; c = m.next())
  { c->print(); cout << "\n"; }
}
```

2.4.3 Other OODB-Like Services

SHORE provides support for concurrency control (via locking) and crash recovery (via logging); these services are integrated with the support for data caching described below. Shore will also provide users with a choice of lower levels of consistency and recovery. Details of these reduced levels are still being worked out. Other SHORE services include optimized object queries over bulk types and a flexible, user-controllable notion of "sticky" object clusters to permit users to cluster (and later recluster) related objects.

3 The SHORE Architecture

3.1 Peer-to-Peer Server Communication

Figure 2 in Section 1.2 illustrates the process structure of SHORE. SHORE executes as a group of communicating processes called *SHORE servers*. SHORE servers consist exclusively of *trusted* code, including those parts of the system that are provided as part of the standard SHORE release, as well as code for Value Added Servers (VASs) that can be added by sophisticated users to implement specialized facilities (e.g., a query-shipping SQL server) without introducing the "client-level server" problem described earlier. Application processes (labeled "App" in Figure 2) manipulate *objects*, while servers deal primarily with fixed-length *pages* allocated from disk *volumes*, each of which is managed by a single server.[7] Applications are not trusted, in the sense that a buggy or malicious application can only modify objects that it is authorized to access; in particular, it cannot corrupt metadata such as slot tables, indexes, or the directory structure.

Each SHORE server plays several roles. First, it is a page-cache manager. The cache may contain pages from local volumes as well as pages from remote servers containing objects that were requested by local client applications. Second, the server acts as an agent for local application processes. When an application needs an object, it sends an RPC request to the local server, which fetches the necessary page(s) and returns the object. (More details are provided in the following section.) Finally, the SHORE server is responsible for concurrency control and recovery. A server obtains and caches locks on behalf of its local clients. The *owner* of each page (the server that manages its volume) is responsible for arbitrating lock requests for its objects, as well as logging and committing changes to it. Transaction management is described in more detail below.

This process structure provides a great deal of flexibility. When acting as an owner of a page, the SHORE server performs the role of the server in a traditional data-shipping, client-server DBMS; when acting as the agent for an application, it plays the role of client. Letting the SHORE server assume both roles allows data placement to be optimized according to workload. For example, data that is largely private to a single user could be owned by the SHORE server on that user's workstation. The location-transparency (from the application's viewpoint) provided by the caching-based architecture allows an application on such a workstation to access both local and remote persistent data in an identical manner. Furthermore, the ability to cache pages at the local server can greatly reduce any observed performance penalty for accessing remote data. In Figure 2, applications running on Workstation 2 can access data that is largely private through the local SHORE server, while obtaining other shared data from the other SHORE servers. With a query-shipping architecture implemented by a "higher level" value-added server (such as an SQL server), applications would communicate directly with remote servers.

3.2 SHORE Software Components

3.2.1 The Language Independent Library

Figure 7 depicts the components of the SHORE software linked with each application. When the application attempts to dereference an "unswizzled" pointer, the language binding generates a call to the object-cache manager in the *language-independent library* (LIL). If the desired object is not present, the LIL sends an RPC request to the local server, which fetches the necessary page(s) if necessary by reading from a local disk or sending a request to another server.[8] If the local operating system supports shared memory, the server uses it to deliver a page of objects to the LIL more quickly. We are experimenting with cache-management strategies that cache objects which come "for free" on the same page as a requested object.

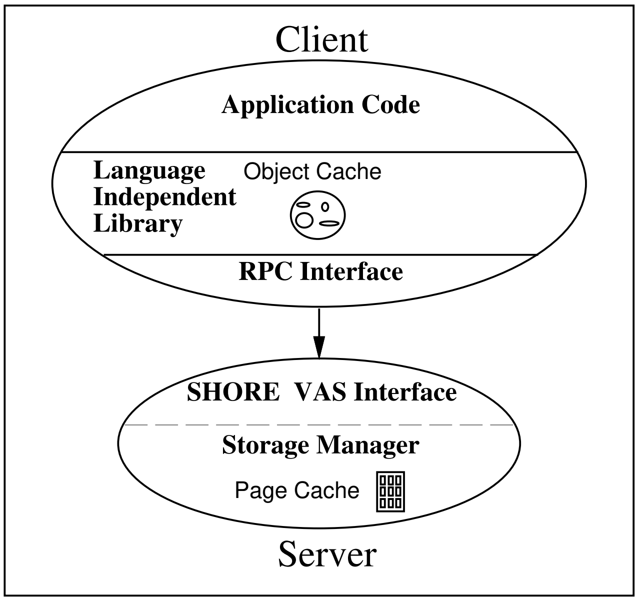

Figure 7: Application/Server Interface

To avoid paging, the object cache manager locks the cache in memory and uses LRU replacement if it grows too large. All OIDs in the cache are swizzled to point to entries in an *object table*. This level of indirection allows objects to be removed from memory before the transaction commits, without the need to track down and unswizzle all pointers to them.

The LIL also contains the Unix compatibility library, with procedures that emulate common file system calls such as `open`, `read`, and `seek`. Finally, the LIL is responsible for authenticating the application to the server using the Kerberos authentication system [MNSS87].

3.2.2 The SHORE Server

Figure 8 shows the internal structure of the SHORE server in more detail. It is divided into two main components: a *Server Interface*, which communicates with applications, and the *Storage Manager* (SM), which manages the persistent object store.

The Server Interface is responsible for providing

[7] At present, disk volumes are not replicated.

[8] An OID contains a volume identifier. The server uses a global volume-location service to find the appropriate server and establishes a network connection if necessary.

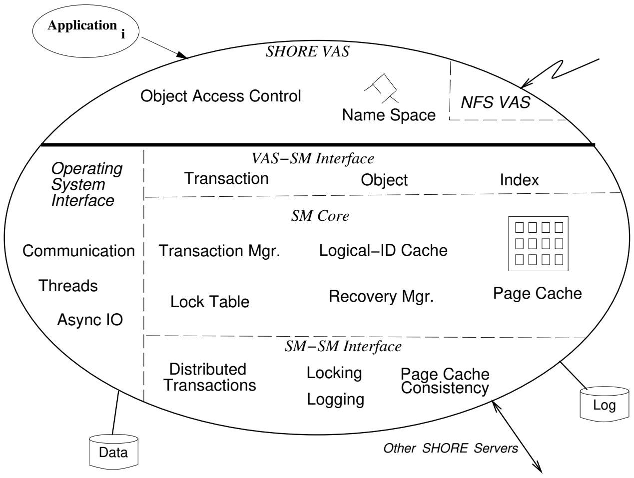

Figure 8: SHORE Server Components

access to SHORE objects stored in the SM. It manages the Unix-like name space and other structures described in Section 2.3. When an application connects with the server, the server associates Unix-like process state (such as a user ID and current directory name) with the connection. User ID information is checked against registered objects when they are first accessed to protect against unauthorized access. As in Unix, the current directory name information provides a context for converting file (path) names into absolute locations in the name space.

The Server Interface is actually an example of a *value-added server* (VAS). Another VAS is the NFS server described in Section 2.3.2. Each VAS provides an alternative interface to the storage manager. They all interact with the storage manager through a common interface that is similar to the RPC interface between application processes and the server. It is thus possible to debug a new VAS as a client process and then migrate it into the server for added efficiency when it is completely debugged. Another example of a VAS could be an SQL server that provides a query-shipping interface to a relational database.[9]

Below the server interface lies the Storage Manager. As shown in Figure 8, the SM can be viewed as having three sub-layers. The highest is the VAS-SM interface, which consists primarily of functions to control transactions and to access objects and indexes. The middle level comprises the core of the SM. It implements records, indexes, transactions, concurrency control, and recovery. At the lowest level are extensions to the core that implement the distributed server capabilities described in Section 3.1. In addition to these three layers, the SM contains an operating system interface that packages together multi-threading, asynchronous I/O, and inter-process communication.

[9] An SQL server VAS is an example of a rather different use of the SHORE Server; the upper layers and type system of SHORE would essentially be thrown away, and the facilities provided by the SHORE Storage Manager would be used in the construction of a completely different, customized server.

3.3 Some Implementation Details

A detailed description of the storage manager is beyond the scope of this paper. However, in this subsection we highlight three of the important technical issues that arise in its implementation: cache consistency, transaction management, and object identifier implementation.

3.3.1 Cache Consistency

In SHORE, there are two types of caches—the object caches used by applications and the page caches maintained by SHORE servers. These two types of caches are managed very differently. The SHORE servers' page caches are allowed to retain their contents across transaction boundaries (called *inter-transaction caching*). Cache consistency is maintained through the use of a *callback locking* protocol [HMN$^+$88, LLOW91, WR91, FC92]. The application/server interface, however does not support "upcalls." Requiring application processes to respond to remote procedure calls would interfere with other synchronization mechanisms used by many application programs such as threads packages, graphics (X11 or InterViews), and networking interfaces. Therefore, the object cache is invalidated (and locks are released) at the end of a transaction. We plan to explore techniques to extend the use of the object cache across transaction boundaries later in the SHORE project.

To balance efficiency against the need for fine-grain concurrency, SHORE uses an *adaptive* version of callback locking that can dynamically adjust the granularity (e.g., page vs. object) at which locking is performed depending on the presence of data conflicts [CFZ93]. This adaptive algorithm is based on the notion of *lock de-escalation* [LC89, Jos91].

3.3.2 Transaction Management

When an application wishes to commit a transaction, a commit request is sent to its local server. If the transaction has modified data that is owned by multiple servers, then a two-phase commit protocol is used among the relevant servers. If the local server has a log, it will coordinate the distributed commit protocol; otherwise, it will delegate the coordinator role to another server. Transactions that only access data that is owned by the local server can commit locally. Thus, the peer-to-peer architecture incurs the additional overhead of distributed commit only when it is necessary.

The transaction rollback and recovery facilities of SHORE are based on the ARIES recovery algorithm

[MHL+92] extended for the client-server environment of SHORE. The client-server distinction reflects the roles played by the server with respect to an object. A server that owns an object is the one that stores the log for that object and that performs all recovery operations on the object. Servers caching the object behave as clients and generate log records that are shipped to the owner of the object. The initial implementation of SHORE relies on a simple extension of ARIES that we call *redo-at-server*. In this extension, a client never ships dirty pages back to the server, only log records; when the server receives log records from a client, it redoes the operations indicated by the log records. This is easy to implement, and it has the advantage of eliminating the need to send dirty pages back to the server.[10] The primary disadvantage is that the server may need to reread pages if it has flushed them from its cache. In the future, we plan to implement the client-server extension to ARIES that was developed and implemented for the EXODUS Storage Manager [FZT+92] and compare its performance to our simpler redo-at-server implementation.

3.3.3 OID Implementation

The implementation of object identifiers (OIDs) has a considerable impact on how the rest of an object manager is implemented and on its performance. The SHORE Storage Manager uses two types of OIDs. A *physical OID* records the actual location of an object on disk, while a *logical OID* is position independent, allowing transparent reorganization such as reclustering. The higher levels of SHORE (including the object cache manager) use logical OIDs to represent object references.

A logical OID consists of an 8-byte volume identifier and an 8-byte serial number. The former is designed to be long enough to allow it be globally unique, allowing independently developed databases to be combined. The latter is large enough to avoid reuse of values under any conceivable operating conditions. When an OID is stored on disk, only the serial number is recorded. The volume identifier is assumed to be the same as the volume containing the OID. For cross-volume references, the serial number identifies a special forwarding entry that contains the full OID of the object (the identifier of the volume that contains it and its serial number relative to that volume).

To map serial numbers to physical OIDs or remote logical OIDs, each volume contains a B+ tree index called its *LID index*. An in-memory hash table is used to cache recently translated entries. The server also eagerly adds translations to this per-transaction translation cache. For example, whenever a server receives a request for an object whose logical OID is not currently in the cache, it requests the page containing that object from the object's server. When that page arrives, the server enters mappings for *all* of the objects on that page into the translation cache. This technique effectively reduces the number of LID index accesses from one lookup per object to one lookup per page of objects.

[10] Generally, the computational cost of the redo is small enough to be ignored, especially when compared to the cost of receiving a page of data via the network.

4 Parallelism in SHORE

Among the goals of SHORE is to be able to support parallel applications as well as single-threaded applications. As in any parallel environment, the challenge is to identify the available sources of parallelism, to define services and interfaces that allow applications to exploit this parallelism, and to provide high-performance implementations of these services. We divide parallelism into inter-transaction parallelism and intra-transaction parallelism.

Inter-transaction parallelism merely means running independent transactions concurrently on multiple processors. Our target architecture for Parallel SHORE is a shared-nothing multiprocessor. (Such a multiprocessor could either be a commercial shared-nothing multiprocessor or a network of workstations). The SHORE symmetric peer-to-peer server architecture is an ideal basis for constructing a parallel persistent object store on such a platform; as in client-server SHORE, a process on one node of the multiprocessor can obtain an object stored anywhere in the multiprocessor by presenting the object's OID to its local server.

Intra-transaction parallelism is not easy to identify *a priori*. One way we can address the needs of large-scale parallel applications is by noting that persistent object store applications become large and slow by accessing large amounts of data, and that object bases grow large by storing large collections of homogeneous objects. Thus, the primary target of our work in Parallel SHORE is to provide a framework under which operations over these large collections of objects can be run in parallel.

4.1 SHORE ParSets

Currently, the basic parallel construct in SHORE is the ParSet (short for "Parallel Set.") The ParSet concept was proposed by Kilian [Kil92] as a way of adopting the data parallel approach to object-oriented parallel programming. There are two ways in which ParSets expose parallelism. First, set-oriented queries over ParSets can be parallelized in the same way that relational queries are parallelized in relational systems. Second, when coupled with object-oriented programming, ParSets can be used to provide a parallel "set-apply" operation, which invokes a method on every element of the ParSet in parallel. A similar approach (in parallel, apply an arbitrary function to every member of a set) was used in the "filter" operation in the Bubba project at MCC [BBKV87].

For Parallel SHORE, we distinguish between two forms of ParSets: primary and secondary. We use the terms "primary" and "secondary" by analogy to their common use with database indexes. Primary ParSets have a physical implication, in that primary ParSets are used for data partitioning. In contrast, secondary ParSets are just logical collections of objects; they can denote a set of objects over which an "apply" is to be executed, but they do not imply anything about where the objects actually reside. Thus, an object can be in any number of secondary ParSets, but it can only be in one primary ParSet.

4.2 Using ParSets

Due to space constraints, instead of giving a detailed description of ParSet semantics and implementation [11], we will give an informal description of how we have used ParSets to parallelize the OO7 benchmark. This code is currently running on our prototype ParSet implementation on a network of workstations. For the purposes of the following discussion, it suffices to know that the OO7 database contains a set of *Composite Parts*, each of which has an associated subgraph of *Atomic Parts*. In the parallel OO7 implementation, the composite parts are stored in a ParSet that is distributed over the nodes of the system by hashing on composite part ID[12]. At any given node, the portion of the ParSet at that node looks like any other SHORE collection. That is, if there are N processors, node 1 has a collection with $1/N$ of the composite parts in it, node 2 has another $1/N$ of the composite parts, and so forth.

Figure 9 illustrates the process and communication structure of a Parallel SHORE (PSHORE) application. Like any PSHORE application, parallel OO7 has a designated "main" or "master" process running on one of the nodes of the system. In addition to this "master" process, slave processes will be running on all of the other nodes of the multiprocessor; slaves are forked by the master when required, and have exactly one master throughout their lifetimes. Slave processes contain all of the methods that could be invoked on objects in ParSets, and they loop waiting for messages from the master process. For example, suppose that the main program executes the ParSet method "apply T1 to the composite parts in the composite part ParSet." ("T1" applied to a composite part traverses the subgraph of atomic parts contained within that composite part.) This will cause the master to send messages to all of the slaves, saying "apply T1 to all composite parts in your partition of the ParSet." The slaves will execute this request in parallel by talking to their local servers, fetching composite part objects into their object caches, and calling the T1 method on each one.

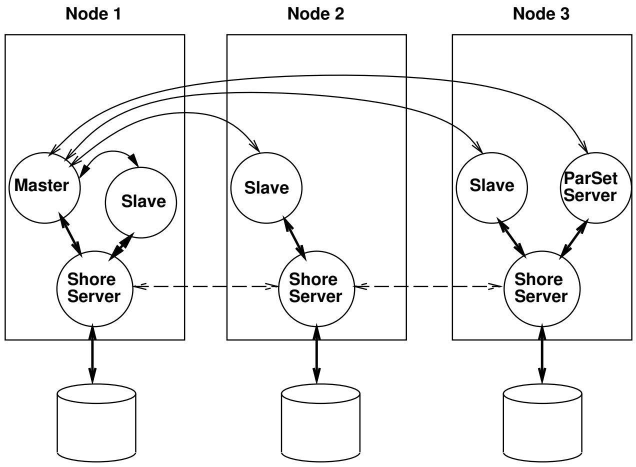

Figure 9: Parallel Shore Architecture

The parallelism just described is largely transparent to the application programmer, who simply writes a single-threaded C++ SHORE program containing ParSet "apply" calls. Slave code is generated at compile time, and all of the necessary communication and synchronization between the master and the slaves is handled by the PSHORE runtime system. State information (such as ParSet catalog information, process ids, and port numbers) can be obtained by the runtime system by consulting the *ParSet server*. Note that executing T1 on a composite part at one node may require access to atomic objects residing on another node; this is also transparent, in this case to the slave executing T1. The slave just requests the objects from its local SHORE server, which is then responsible for contacting other SHORE servers for any remote objects that are needed. Since all slaves (and the master) share a global OID space, this gives PSHORE applications a shared-memory flavor even though the processors do not actually share any memory. We are currently designing synchronization primitives to support this programming model (although the OO7 implementation does not currently require these primitives).

4.3 Portability

Our goal is that PSHORE, like SHORE itself, will run on a wide range of hardware platforms. To support this goal, the ParSet implementation uses the "new threads" package [FM92], which in turn uses PVM [Sun90] for interprocess communication. (PVM is a public-domain message passing library for writing parallel programs that runs on platforms ranging from networks of workstations to multiprocessors.) By using this portable parallel programming environment in the PSHORE implementation we hope to ensure that, like

[11] More detail appears in [DNSV93].

[12] CompositePart has an attribute called *partID*, which is separate from the OID of the composite part object

SHORE, PSHORE will also be usable by anyone with a network of workstations.

5 Conclusion

SHORE is an integration of file system and OODB concepts and services. From the file system world, SHORE draws object naming services, support for lower (and cheaper) degrees of transaction-related services, and an object access mechanism for use by legacy Unix file-based tools. From the OODB world, SHORE draws data modeling features and support for associative access and performance acceleration features. To provide scalability and a basis for parallelizing the system, SHORE also employs a novel architecture, including support for symmetric peer-to-peer server communication and caching; in addition, it includes support for extensibility via the value added server facility. Like our previous system, EXODUS, we will make the SHORE system publicly available via anonymous FTP; we expect the first release of SHORE to occur in mid-1994.

References

[AS93] Paul Adams and Marvin Solomon. An overview of the CAPITL software development environment. In *Proceedings of the Fourth International Workshop on Software Configuration Management*, Baltimore, MD, 1993.

[BBKV87] F. Bancilhon, T. Briggs, S. Khoshafian, and P. Valduriez. FAD, a powerful and simple database language. In *Proc. VLDB Conf.*, Brighton, England, 1987.

[Cat93] R. Cattell. *The Object Database Standard: ODMG-93*. Morgan Kaufmann, San Mateo, CA, 1993.

[CDF+86] 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 *Proceedings of the Twelfth International Conference on Very Large Data Bases*, pages 52–65, 1986.

[CDN93] Michael J. Carey, David J. DeWitt, and Jeffrey F. Naughton. The OO7 benchmark. In *Proceedings of the 1993 ACM-SIGMOD Conference on the Management of Data*, Washington D.C., May 1993.

[CFZ93] M. Carey, M. Franklin, and M. Zaharioudakis. Fine-grained sharing in a page-server OODBMS. Submitted for publication., December 1993.

[Deu91] O. Deux et al. The o_2 system. *Communications of the ACM*, 34(10), October 1991.

[DLPY93] D. DeWitt, J. Luo, J. Patel, and J. Yu. Paradise — a parallel geographic information system. In *Proceedings of the ACM Workshop on Advances in Geographic Information Systems*, November 1993.

[DNSV93] D. DeWitt, J. Naughton, J. Shafer, and S. Venkataraman. ParSet design document. Unpublished manuscript, November 1993.

[FC92] M. Franklin and M. Carey. Client-server caching revisited. In *Proceedings of the International Workshop on Distributed Object Management*, Edmonton, Canada, August 1992. (published as *Distributed Object Management*, Ozsu, Dayal, Vaduriez, eds., Morgan Kaufmann, San Mateo, CA, 1993).

[FM92] Edward W. Felten and Dylan McNamee. Newthreads 2.0 user's guide. August 1992.

[FZT+92] Michael J. Franklin, Michael J. Zwilling, C. K. Tan, Michael J. Carey, and David J. DeWitt. Crash recovery in client-server EXODUS. In *Proceedings of the ACM-SIGMOD Conference on the Management of Data*, pages 165–174, June 1992.

[HMN+88] J. Howard, M. Kazarand S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1), February 1988.

[Jos91] A. Joshi. Adaptive locking strategies in a multi-node data sharing environment. In *Proc. 17th VLDB Conf.*, Barcelona, Spain, Sept. 1991.

[Kil92] Michael F. Kilian. *Parallel Sets: An Object-Oriented Methodology for Massively Parallel Programming.* PhD thesis, Harvard Center for Research in Computing Technology, Cambridge, MA, 1992.

[LC89] Tobin J. Lehman and Michael J. Carey. A concurrency control algorithm for memory-resident database systems. In *Proc. of the 3rd Int'l. Conf. on Foundations of Data Organization and Algorithms*, Paris, France, June 1989.

[LLOW91] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The ObjectStore database system. *Communications of the ACM*, 34(10), October 1991.

[MHL+92] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems*, March 1992.

[MNSS87] S. P. Miller, B. C. Neuman, J. I. Schiller, and J. H. Saltzer. Section E.2.1: Kerberos authentication and authorization system. Technical Report Project Athena Technical Plan, M.I.T. Project Athena, Cambridge, MA, December 1987.

[Obj92] Objectivity, Inc. Objectivity reference manual. 1992.

[Ont92] Ontos, Inc. Ontos reference manual. 1992.

[RCS93] 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), July 1993.

[RT74] Dennis M. Ritchie and Ken Thompson. The unix time-sharing system. *Communications of the ACM*, 17(7):365–375, July 1974.

[SGK+85] R. Sandberg, D. Goldberg, S. Kleiman, D.Walsh, and B.Lyon. Design and implementation of the sun network filesystem. In *USENIX Summer Conference Proceedings*, 1985.

[Sun90] V. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4), December 1990.

[Ver92] Versant, Inc. Versant reference manual. 1992.

[WR91] Y. Wang and L. Rowe. Cache consistency and concurrency control in a client/server dbms architecture. In *Proceedings of the ACM SIGMOD Conference*, Denver, CO, June 1991.