1989

Implementing Object Support in the RAID Distributed Database System

Bharat Bhargava
Purdue University, bb@cs.purdue.edu

Prasun Dewar

James G. Mullen

Jagannathan Srinivasan

Report Number:
89-899

TR 899

Implementing Object Support in the RAID Distributed Database System *

Bharat Bhargava Prasun Dewan James G. Mullen Jagannathan Srinivasan

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

Abstract

Raid is a distributed database system based on the relational model. O-Raid is an extension of the Raid system that supports complex data objects. Its data model combines properties of the relational model in Raid and the object model in C++. In this paper we discuss the details of implementing O-Raid. In particular, we describe the organization of objects on secondary storage, indexing of relations containing objects, processing of queries involving objects, and handling of method invocation. We also discuss how the components of the server-based organization in Raid are reused in the implementation of O-Raid.

1 Introduction

The various components of a traditional computing system offer a wide variety of data models. Typically, a conventional programming language offers a Pascal-like type system for defining data, an object-oriented programming language supports objects [17], an operating system provides a hierarchical system of files and directories, a database system supports relational schema, and a programming environment encapsulates data as syntax trees.

Supporting different data models increases the "impedance mismatch" problem in the system. This is caused by the need to translate between different kinds of data structures. Several tools need to do this kind of translation. For instance, a tool that allows users to execute queries on programs [12] may need to translate between syntax trees and the database model. Similarly, an interactive application displaying database information needs to translate between the programming language and the database structures.

*This research is supported in part by NASA, AIRMICS, and the UNISYS Corporation.

One approach towards increasing the level of data model integration in a system is to support a complex data model in a database management system (DBMS), thereby reducing the need for specialized models. Traditional relational DBMS's are unsuitable for describing complex information such as syntax trees, geometric and geographical databases, and VLSI circuits. Recently, there has been much interest in supporting the object model in the database as a mechanism to store complex information in a general purpose DBMS. We have adopted this approach towards systems integration. Unlike many systems such as Orion [1] and GemStone [13] which have taken the "revolutionary" approach of replacing the relational model with the object data model, we have taken the "evolutionary" approach of extending the relational model with objects. Our basis for this research is Raid [3], a distributed database system based on the relational model.

We have designed a system called O-Raid, which extends Raid relations with objects. There are many approaches towards the object model, as mentioned in [17], where several "dimensions" of object-oriented design are enumerated. In the design of O-Raid, we have adopted the programming language dimensions from C++ [16], thereby supporting an existing, well-accepted language for defining major components of database object model. C++ is a programming language and is not adequate for defining all parts of the object database model. It does not support, for instance, persistence, sharing, or indexing. Therefore, we have added several "database dimensions" to the C++ object model in our design of O-Raid.

By providing objects in O-Raid we are not only able to support a complex integrated data model but also triggering of methods in response to updates to the database [8], flexible displays of data structures, and customizable user interfaces [7]. By building on

Raid, we are able to support in O-Raid many important database facilities provided by Raid including distribution and replication of data. Since we use C++ as a basis for the database model, there is no impedance mismatch when data are transferred between C++ applications and the database.

In [8], we presented a detailed design of O-Raid and some brief notes on how it may be implemented. This paper describes our implementation plan in more detail. Several other works [2, 4, 10, 14, 15] have also discussed approaches for implementing the object database model. There are two distinguishing features of our scheme. First, our starting points for the implementation consisting of Raid and C++ are different from other implementations. Second, the implementation supports several features that are unique to O-Raid such as relations and columns as objects, static database variables, and private pointers.

This paper is organized as follows. Section 2 gives an overview of the O-Raid design, explaining its major distinguishing features including the query language SQL++. Section 3 briefly explains the implementation of Raid. Section 4 gives the details of the O-Raid implementation, including object storage and addressing, indexing, and method determination and invocation. Finally, section 5 presents our plans for future work and conclusions.

2 Overview of the O-Raid Design

Like Raid, O-Raid allows a user to define relations to store data. Unlike Raid, these relations may contain database objects which are instances of *database classes*. A database class is created by first creating a C++ base class such as *shape*

```
class shape {
  public:
    // coordinates of the centroid of the object
    int x_coord, y_coord;

    shape (float x, float y);

    int distance (shape other_shape);

    int operator < (shape other_shape);
};
```

This class is registered with the DBMS using the NEWCLASS declaration

```
NEWCLASS {
  class shape;
  file shape.h;
};
```

Instances of a database class are similar to the corresponding instances of the base class. Thus they are associated with private and public variables and methods, can refer to other objects directly or via pointers, and can access static variables defined by their class. Unlike their language counterparts, they are persistent, can be stored in relations, and can also be created independently as *global variables*.

A pointer to a database object may be *shared*, that is shared equally among all objects that have a reference to it, or *own* [1, 4], that is accessible to several objects but owned by a single object. The referent of an own pointer is deleted with its parent. In O-Raid, a pointer can also be a *private* pointer, which is like an *own* pointer except that its owner is the only object that has a reference to it. Thus, referents of private pointers are like values of non-pointer variables except that they can have a variable (including recursive) structure.

O-Raid supports the SQL++ query language, which extends SQL with constructs for manipulating objects. It allows a user to:

create global variables.

```
CREATE VARIABLE shape S
    SET (S = shape(1,2));
```

manipulate global variables,

```
UPDATE S SET (x_coord = 0);
```

define relations containing objects,

```
CREATE TABLE shapes
    (char[20] name, shape *obj);
```

add objects to relations

```
INSERT INTO triangles VALUES
    (name = "S1", obj = shape(0,0));
```

and select objects from relations

```
SELECT *obj FROM shapes
    WHERE *obj < %S;
```

where %S refers to the contents of the global variable S.

As in C++, a variable declared as a pointer to a particular class in O-Raid can refer to objects of that class and all its subclasses. The class of a variable changes either when it is assigned a new object of a different (but legal class) or when it is *reclassified*. When a variable is reclassified, a new object of a user-specified class is created, which has copies of the variables of the old object that are common to both classes.

All attributes of a relation, including those containing objects, can be used as indices:

```
CREATE INDEX objindex ON shapes (obj);
```

Moreover, instance variables of these objects and their instance variables and so on can be used as indices. If an instance variable is declared to be pointer to some class C, then only those variables of its referent that are defined by class C can be used as indices. In particular, instance variables defined by subclasses of C cannot be used as indices. The class of those attributes/instance variables that are used as indices must define the "<" operator.

In O-Raid, each relation is considered an object since it is associated with instance variables which can be manipulated by methods defined in its class. The state of the relation is separate from the collection of tuples contained in it. In this respect, the relation is like a Smalltalk-80 [9] class. A Smalltalk-80 class is also associated with its own state, which is distinct from the state of its instances. In Smalltalk-80, the state of a class contains common properties of its instances. Similarly, the state of a relation object contains common properties of the objects stored in it, such as, for a relation containing shapes, the boundaries of the plane to which the shapes are constrained. Like a relation, a column of a relation is also an object associated with its own variables and methods, which are defined in its class.

Finally, O-Raid allows SQL++ queries to be executed from a C++ application and data may be transferred between the application program and the database without impedance mismatch. Thus the following query may be executed from an application

```
SELECT ($n = name, $s = obj)
    FROM triangles
    WHERE (name = "S1");
```

to read data into the program variables $n, and $s.

3 Raid Overview

Raid is a robust and adaptable distributed database system for transaction processing [3]. Raid is a message passing system with server processes on each site. Raid divides the functions of transaction processing into software modules called servers. An operating system process can implement the capability of a single server or a collection of servers. The server design has facilitated the implementation effort by providing for flexibility, and by explicitly defining the interfaces between servers. This architecture provides for modularity and extensibility. The modularity of Raid facilitates adding support for objects, because much of the servers' code can be reused. A high level, layered communication package provides a clean, location independent interface between servers. The organization of the Raid system is shown in figure 1.

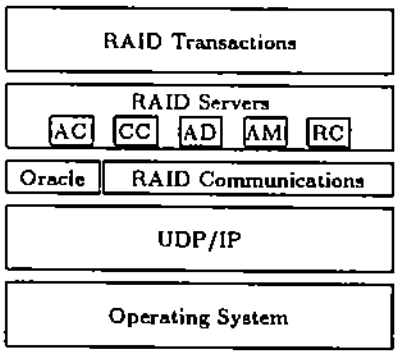


Figure 1: The Raid Architecture.

Figure 2 shows the communication paths among the servers in a two site Raid instance. The version of Raid described in [3] has evolved to support a new control flow for efficient communication among the concurrency controller, atomicity controller and replication controller. The roles of the servers in the Raid system are:

- **User Interface (UI)**: a front end invoked by the user to process SQL++ queries.
- **Action Driver (AD)**: accepts a parsed query from the UI, formats the query as a transaction

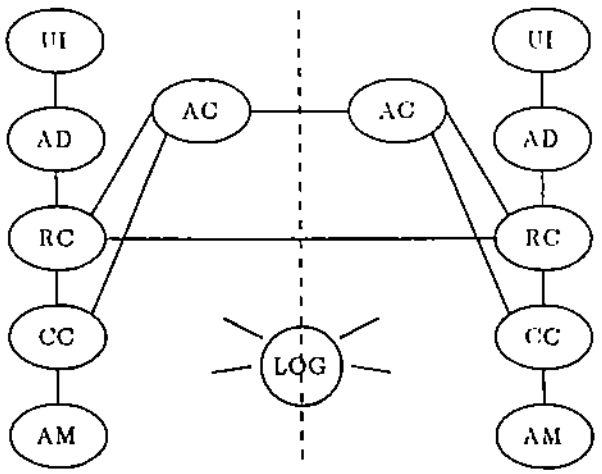


Figure 2: The Server links in a Raid instance with two sites.

(read and write actions), and executes the transaction.

- **Access Manager (AM)**: provides access to the local database and ensures that updates are posted atomically to stable storage.
- **Atomicity Controller (AC)**: manages the commit phases of transaction processing to ensure that a transaction commits or aborts globally.
- **Replication Controller (RC)**: maintains consistency of replicated copies of the database in the event of failures.
- **Concurrency Controller (CC)**: maintains serializability among concurrent transactions.

Facility exists to merge selected servers into a single process to improve performance [11]. In a typical implementation, the transaction processing servers CC, RC, AC, and AM are merged to run as a single process. The Raid servers communicate with each other using high-level operations . For example the AD uses the routine `AD_Read_RC()` to send a read request to the RC. The Raid communications package uses UDP (User Datagram Protocol) to implement these operations. It provides a number of extensions to UDP, including arbitrary sized messages.

4 Implementing the O-Raid System

The organization of Raid into a set of communicating servers is preserved in O-Raid, since it provides a modular and extensible approach to implementing databases. Raid is expected to evolve further as we gain more usage experience.

We retain the basic set of Raid servers in O-Raid. As detailed in the following sections, this organization of servers "works" for distributed object management. Extensive measurements and experiences will be used to restructure the architecture.

In Raid, the physical organization of the database consists of a set of files, each of which stores the tuples of a relation in fixed-length records. We retain this organization in O-Raid which may seem unintuitive, since objects containing pointers can have variable structure, and be created independently of relations. In the following sections, we discuss the details and consequences of this organization.

Besides physical organization of objects, a major concern in the implementation of an object-oriented system is the mechanism for method invocation. In O-Raid, we reuse the implementation of C++ method invocation by ensuring that a call to a method defined in a database class is translated into a call to the corresponding method in the base class.

Reuse of components from Raid and C++ makes an implementation of O-Raid tractable. We do not need to address the implementation of several complex components such as efficient communication among servers, concurrency control, inheritance, or overload resolution. The following are the main issues we address in the remainder of this section:

- How objects are organized on secondary storage.
- How objects are addressed, both in secondary and primary storage, and how the address translation scheme works.
- Support for indexing.
- The mechanism for translating an SQL++ method invocation in a database class into a C++ method invocation in a base class.
- How data are transferred between applications and the database.
- How concurrency control and other components of Raid can be reused directly in O-Raid.

4.1 Object Storage

In O-Raid, objects may be created as:

- values and referents of relation attributes
- values and referents of instance variables of objects
- values and referents of global variables
- relation and column objects
- values and referents of static variables of a class

Objects that are values of attributes of a relation are stored in subcolumns of the relation column corresponding to the attribute. Each hierarchical column has a subcolumn for each instance variable of the object. Objects that are values of these instance variables are stored within subcolumns of the column corresponding to the parent object. Thus there is a hierarchy of columns. Figure 3 illustrates a two-level hierarchy. The attribute *Scores* is stored in a hierarchical column containing subcolumns corresponding to the exam and homework scores, which in turn contain subcolumns corresponding to specific exams and homeworks respectively. Storing object values within the parent object promotes clustering, since these values can be transferred between disk and main memory using a single disk operation.

In O-Raid, this approach to clustering applies only to subobjects of an object that are the values of instance variables/attributes (called own values of their parent). It does not apply to subobjects that are referents of instance variables/ attributes, for two reasons:

1. Referents of pointers can be of different types, and therefore different sizes. Supporting variable-sized tuples in relation columns requires either storing them in fixed-sized records, thereby bounding their size, or a complex relation update scheme that handles insertion/deletion of variable-sized records. Moreover, it requires changes to all components of Raid that assume fixed-size records.
2. Referents of shared and own pointers can have multiple parents.

We keep database pointers (called RID's, as discussed in next section) to referents of attributes/instance variables within the parent object, and store these objects in special system-defined relations called *class relations*.

Each class has a *class relation* that holds all instances of the class that are not stored directly in another relation. The class relation has several pieces of information including:

- The OID (discussed in the next section) of the object.
- A field indicating how the object is referenced by other objects, either shared, owned, or private.
- A flag indicating whether the object is currently owned.
- The values of the instance variables of the object, stored directly or as pointers.

The name of the *class relation* is the same as the name of the class it is used for. In figure 3 examples of two class relations *Students* and *Professors* are shown. RTYPE refers to how the object is referenced by other objects.

As mentioned in section 2, the class of the referent of a pointer may change either by (a) assigning a referent of a different class to the pointer, or (b) reclassifying the referent. The first case is simple to handle in O-Raid, since the instance variable/attribute is simply assigned the new RID referring to an object in a different class relation. The second case is handled by creating an object of the new class with the same OID and values of instance variables of the old referent that are common to the two classes, and placing the object in the new class relation.

Objects named by global variables are also stored in the appropriate class relations. The system defines a special relation which is used to translate the names of these variables to RID's. It contains the variable name, type, and RID of the value stored in the class relation, and a field indicating whether the variable has been declared as a pointer variable such as

CREATE VARIABLE shapes *S;

or a "direct" variable such as

CREATE VARIABLE shapes S;

For all relations its instance variables and the instance variable of its column objects are stored in a fixed-area in the file allocated for the relation. For a class relation the static variables [1] of the associated class are also stored in its file (as shown in figure 4).

[1] In C++ a static variable of a class has only one copy which is shared among all its instances.

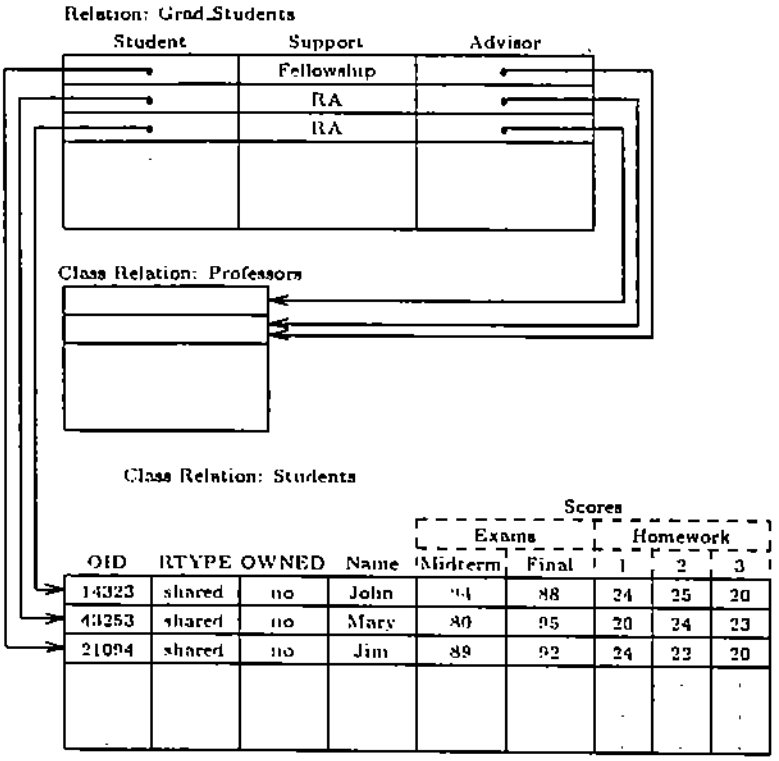
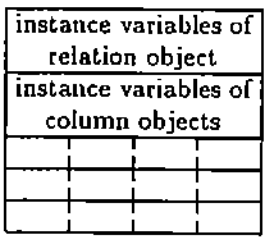
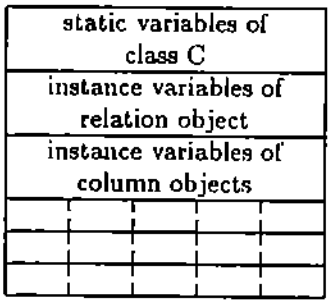


Figure 3: Storing Objects in Relations.



Ordinary Relation



Class Relation for Class C

Figure 4: Structure of Relations.

4.2 Addressing Objects

Object Addresses. In secondary storage, an object refers to another object via an RID (Reference Identifier). An RID contains an OID (Object Identifier) which uniquely identifies the object and a PID (physical identifier). The PID is a *hint* about the physical address of the object and consists of a file name and an offset within the file. It can be directly used to locate the object. It is not always a true indication of the physical location of the object since the offset within the file may change as a result of deletions and compactions while the relation itself may change as a result of object reclassification. However, it is a true indication "most" of the time, if objects are rarely moved.

Address Translation. Each class relation stores the OID of the object as the first attribute. Given a reference to an object (which will contain the OID and PID of the object), the PID is extracted and the corresponding tuple read from the disk. The OID field in this tuple is compared with the OID in the reference. If they match then we have accessed the object. If the OIDs do not match, a special relation

RID		
OID	PID	
4 byte integer	file name	offset

Figure 5: Structure of Reference Identifiers in O-Raid.

(*oid2pid$*) is searched for this OID and the correct PID extracted. This PID is used to fetch the object. In the reference the old PID is replaced by the correct one so that future references will not require a lookup.

4.3 Indexing

Indexing a non-pointer field is supported in O-Raid by using the "<" operation defined in the class of the field. Indexing the pointer fields is more complex. One approach to supporting pointer indices is illustrated in [13], which creates two kinds of index structures for these fields: an *identity index* which uses the OID stored in the field, and a *value index* which uses the value of the referent. In O-Raid we use an adaptation of this approach that handles private pointers differently from other pointers. Suppose the following declaration is used for creating a value index:

CREATE INDEX objindex ON shapes (obj);

The system builds an identity index for relation *shapes* on the OID's stored in the *obj* attribute, and a value index for the class relation *shape* on the shapes stored in it. To process the query

SELECT name FROM shapes
WHERE *obj = %S;

the class relation *shape* is searched for OID's of all objects that satisfy the predicate. Next the relation *shapes* is searched for these OID's using the identity index on *obj*, and the corresponding names are printed out.

In this example, both the relation *shapes* and the class relation *shape* are searched to process the query. We reduce this overhead for private pointers by building a single value index in the parent relation, and keeping in the parent copies of appropriate instance variables of the referent. A back pointer from the referent to the parent containing the private pointer is used to keep these copies consistent. For instance, if the pointer attribute *obj* was declared as private, then all instance variables of *obj* that are needed to evaluate the "<" operation in class *shape* are stored in the relation *shapes* and these are used to directly build a value index on *shapes*. This approach could also be used for a shared or own pointer but that would require multiple backpointers from the referent to all objects that refer to it. Figure 6 summarizes our approach to indexing.

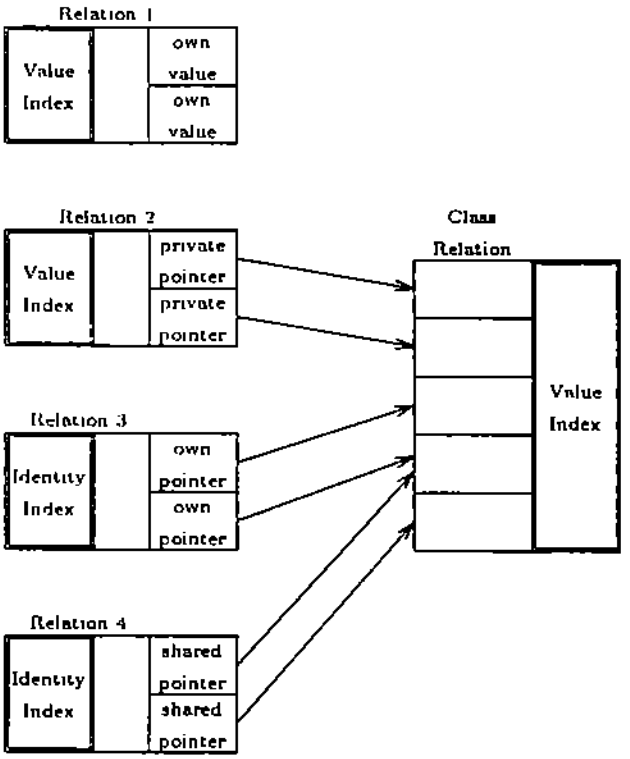


Figure 6: Location of Object Indices.

4.4 Method Determination and Invocation

Method Determination. A relation *methods$* with the attributes *method_name, class, binary_file, offset* and other attributes giving information about the return types and parameters for the method is maintained by the system. This relation gives the location of the code for the method to be called for each valid method_name, class_name pair. This relation is used for method determination.

Method Invocation. The AD invokes a method on an object by dynamically loading both the object and the corresponding method binary in its address space and uses the C++ mechanism for invoking the method on the object. Before it invokes the method, it ensures that static variables of the class

are loaded from the class relation and references to them in the method appropriately adjusted. To reduce the method invocation time, the AD keeps a cache of most recently used methods and objects. It copies objects into memory using the approach described in the next section.

4.5 Transferring Data Between Applications and the Database

As mentioned in section 2, a C++ application can invoke queries such as

```
SELECT ($n = name, $s = obj)
    FROM triangles
    WHERE (name = "S1");
```

These queries are translated by a preprocessor, and converted into C++ routines that perform the transfer.

Transfers between the application and the database require the cooperation of the AD, RC, CC, and AM. The RC, CC, and AM work as in Raid and are responsible for replication control, concurrency control, and access management respectively at the tuple-level. The AD (which is linked to the application) is responsible for receiving requests for reading/writing objects from the application and mapping these requests to reading/writing the corresponding tuples. When it is asked to read an object, it only reads the tuple corresponding to the top-level variables of the object. As in PS-Algol [5], pointer fields in these top-level variables are initially set to invalid addresses (persistent storage addresses) that identify RID's for the corresponding objects. The AD keeps a hash table mapping a persistent address to the corresponding RID. When a persistent address is accessed, a trap handler is executed in the AD which converts the pointer to an RID and then to an identifier for the tuple storing the top-level variables of the referent, sends the tuple identifier to the AM, receives the tuple, converts it into the corresponding in-memory data structure, and replaces (swizzles) the persistent storage address with a reference to the data structure. The data structure may in turn contain persistent pointers. When these persistent pointers are dereferenced, the corresponding objects are brought into the memory in a similar manner.

Before the AD reads a database object into a memory variable, it first consults the hash table to see if the object has already been loaded into memory. If the object is loaded, then the in-memory copy of the value/reference to the object is copied into the variable, otherwise the object is read from the database. The hash table is reset whenever the current transaction terminates. This ensures that (a) the program always works with the most recent value of a database object, (b) the structure of a database object is isomorphic to the corresponding memory object (thus cycles are retained), and (c) extra disk transfers are not made to read a database object.

At the end of a transaction, the AD converts all swizzled pointers to the corresponding RID's and builds a writeset of tuple identifiers and the corresponding values to be written to disk. It sends this writeset to the AM, which atomically does the updates, maintaining consistency among replicated data.

4.6 Reuse of Raid Software

Many components of the Raid implementation are initially reused in the implementation of O-Raid.

The AM and CC work at the tuple level and can be reused. This is possible since all database objects are converted to tuples at the operational level in the relational implementation. The AC and RC remain unchanged since the data structures for transactions are the same in Raid and O-Raid.

In the initial implementation of O-Raid the servers that are changed are the User Interface (UI) and Action Driver (AD). The UI translates SQL++ queries into an internal representation, which is executed by the AD using the data structures and algorithms described in the previous sections.

The communication package is adequate since it supports arbitrary sized messages and reasonable delays. This feature is useful for object-oriented systems, which will most likely have very small to very large objects.

We have a protocol *LDG*, for Long DataGram. This protocol has no restriction on packet sizes. LDG is currently built on top of User Datagram Protocol (UDP). Each LDG packet is fragmented if necessary, and then sent using UDP. At the destination, fragments are collected and reassembled. Normally we use fragments of 8000 bytes, which is the largest possible on our Sun [2] workstations. Since IP gateways usually fragment messages into 512 byte packets we also have a version of LDG with 512 byte fragments.

Table 1 compares UDP, LDG, and Raid round-trip communication times for datagrams of various

[2] Sun is a registered trademark of Sun Microsystems, Incorporated.

Bytes:	64	512	2048	8192	32768	500000
UDP	7.2	10.6	16.5	48.8	-	-
LDG-512	10.2	17.2	41.9	147.4	550.0	8,630
LDG-8000	9.6	12.8	19.2	65.1	224.8	4,200
Raid	11.9	20.1	46.7	153.3	-	-

Table 1: Raid Communication Time by Packet Length (in milliseconds)

lengths. LDG-512 is the version of LDG with 512 byte fragments, while LDG-8000 is the version with 8000 byte fragments. LDG is about three milliseconds more expensive than UDP for packets that do not need to be fragmented. LDG-512 becomes much more expensive for larger packets, since UDP and LDG-8000 are only transmitting a single packet. The numbers given for the Raid layer are based on LDG-8000. A new implementation of the Raid layer is expected to perform almost as well as LDG, because of changes that completely avoid buffer copying.

5 Future Work and Conclusions

Our present effort is to put together a complete implementation of O-Raid and run a variety of SQL++ queries. We plan to identify additional changes to the Raid system as we progress. This will give us an opportunity to assess the effort required to extend a relational system with objects, evaluate the efficiency and usability of some of the implementation decisions we have taken, and gain experience in using database objects.

We plan to use O-Raid as a testbed to study the feasibility of replicating *parts* of an object as opposed to replicating the *entire* object. This would be useful especially if objects are very large and consist of many subobjects. Providing support for replicating parts of an object raises interesting issues such as what does consistency among replicated copies mean? How updates to objects are handled? A single update may trigger a sequence of updates. Moreover, the behavior may be different at different sites at which the replicated parts of the object are stored.

We plan to change the CC to support class-specific validation methods and take advantage of semantics associated with objects. Moreover, support for long transactions that operate on complex objects will be implemented.

Data model integration, or "back-end integration", is one step towards uniformity and standardization in the system. It is also important to provide integration of user interfaces, or "front-end integration". The query model described here provides a uniform interface for manipulating data. We are currently exploring how an integrated data model combining the features of query, text editing, structure editing, query-by-example, and hypertext can be automatically supported by the system [6]. Such a model would reduce the need to provide application-specific user interfaces, thereby providing front-end integration.

The O-Raid design supports a complex data model integrating features of the relational and object models. In this paper we have described how the data model may be implemented. The implementation borrows many features from implementation of analogous features in other systems such as Postgres [15] and Orion [1]. There are two distinguishing features of our implementation scheme. First, it uses all components of the implementation of the Raid distributed relational system and the C++ programming language. Second, it supports several features that are unique to O-Raid such as relations and columns as objects, static database variables, and private pointers.

6 Acknowledgement

We would like to thank Ashish Vikram for discussions on the design and implementation of O-Raid.

References

[1] J. Banerjee, H.-T. Chou, J. F. Garza, W. Kim, D. Woelk, N. Ballou, and H.-J. Kim. Data Model Issues for Object-Oriented Applications. *ACM Transactions on Office Information Systems*, 5, Jan 1987.

[2] D. Beech. A Foundation for Evolution from Relational to Object Databases. In *Advances in Database Technology - EDBT '88*, Mar 1988.

[3] B. Bhargava and J. Riedl. The RAID Distributed Database System. *IEEE Trans. Softw. Eng.*, 15(6), June 1989. (New version report in preparation).

[4] M. J. Carey and D. J. DeWitt. A Data Model and Query Language for EXODUS. In *Proceedings of the SIGMOD International Conference on Management of Data*, Jun 1988.

[5] W. P. Cockshot, M. P. Atkinson, K. J. Chisholm, P. J. Bailey, and R. Morrison. Persistent Object Management System. *Software-Practice and Experience*, 14:49–71, 1984.

[6] P. Dewan. Interacting with database objects. Technical Report Technical Report SERC-TR-42-P, Software Engineering Research Center, Purdue University, May 1989.

[7] P. Dewan. Object-Oriented Editor Generation. *Journal of Object-Oriented Programming*, expected to appear in the May/June 1990 issue.

[8] P. Dewan, A. Vikram, and B. Bhargava. Engineering the Object-Relation Model in O-Raid. In *Proceedings Of The International Conference on Foundations of Data Organization and Algorithms*, pages 389–403, June 1989.

[9] A. Goldberg and D. Robinson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, Reading, Mass., 1983.

[10] W. Kim and F. H. Lochovsky, editors. *Object-Oriented Concepts, Databases, and Applications*. ACM Press, 1989.

[11] C. Koelbel, F. Lamaa, and B. Bhargava. Efficient Implementation of Modularity in RAID. In *USENIX Workshop on Experiences with Distributed and Multiprocessor Systems*, pages 127–143, Oct 1989.

[12] M. A. Linton. Implementing Relational Views of Programs. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 132–140, April 1984.

[13] D. Maier and J. Stein. Development of an Object-Oriented DBMS. In *Proceedings of OOPSLA '86*, Sep 1986.

[14] Preliminary Report, A Survey of Object-Oriented Database Systems, May 1989, Purdue University.

[15] M. Stonebraker and L. A. Rowe. The POSTGRES Papers. Technical Report UCB/ERL M86/85, University of California, Berkeley, Jun 1987.

[16] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, Mass., 1986.

[17] P. Wegner. Dimensions of Object-Based Language Design. In *OOPSLA '87 Proceedings*, pages 168–182, October 1987.