OdeFS: A File System Interface to an Object-Oriented Database

N. Gehani
H. V. Jagadish
W. D. Roome

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

OdeFS is a file-like interface to the Ode object-oriented database. Database objects are accessed and manipulated like files in a traditional file system using standard UNIX® commands. For example, the `ls` command can be used to list the objects in a directory and the `cat` command can be used to display the contents of an object. Editors such as `vi` and `emacs` can be used to update objects.

OdeFS is implemented as a network file server, using the NFS protocol. OdeFS commands are translated into calls to the underlying UNIX file systems (to manipulate files or directories) or the Ode object manager (to manipulate persistent objects). OdeFS requires no storage of its own. In consequence, OdeFS is object-compatible with the other Ode interfaces: O++ (C++ interface), CQL++ (SQL interface), and OdeView (graphical interface). Objects created with one interface can be manipulated with the other interfaces. In this paper, we describe OdeFS, its user interface, and its implementation.

1. INTRODUCTION

Ode [1] is an object-oriented database based on the C++ object model. The programming interface to Ode is the O++ database programming language, which extends C++ with facilities for creating and manipulating persistent objects, querying the database, specifying constraints and triggers, and running transactions.

Using O++ to access any object(s) in the Ode database requires writing an O++ program. This can be just as inconvenient as writing a C program for every ad hoc access to a UNIX® file. One solution is to develop a set of interactive utility programs for displaying and manipulating Ode objects, just as UNIX systems have a large set of tools for manipulating files. For example, we could have specialized tools such as an ‘‘Ode editor,’’ ‘‘Ode grep,’’ ‘‘Ode print,’’ etc.

The drawback of this approach is that it takes a great deal of effort to develop and document these programs. Furthermore, it is almost impossible to keep all these tools up to date as the database, the file system, user interface technology, and application needs all change over time.

We propose a different approach. Instead of moving knowledge of Ode into utility programs, we move it into the file system. The *Ode File System,* or *OdeFS* (rhymes with Oedipus), provides a file-like interface to the Ode database. To users, OdeFS looks like part of the UNIX file system. In OdeFS, objects look like files, and object collections, including type extents, look like directories. Users can access and manipulate Ode objects using the same programs they use to manipulate files. For example, the standard UNIX `ls` command lists the objects in a directory, and the `cat` command displays the contents of an object. Editors such as `vi` and `emacs` display or update objects. A file-oriented Graphical User Interface (GUI) can display and select Ode objects. Because no code modification or recompilation is required, proprietary applications can also access Ode objects.

In this paper, we describe OdeFS, its design goals and the implementation. We assume that the reader is familiar with C++ [11] and the UNIX operating system [5].

® UNIX is a registered trademark of USL.

2. DESIGN GOALS AND DECISIONS

2.1 Goals

The fundamental goal of OdeFS is to allow existing file manipulation programs to manipulate objects. To see why this is important, consider Figure 1, which illustrates the conventional approach to manipulating Ode objects, using a set of Ode utility programs, written in O++. These programs would be similar to existing programs for files, but they cannot be identical, so we would have to write and maintain them as part of Ode. The O++ compiler and run-time storage manager would translate the O++ operations into operations on the Ode database files in which the objects are actually stored.

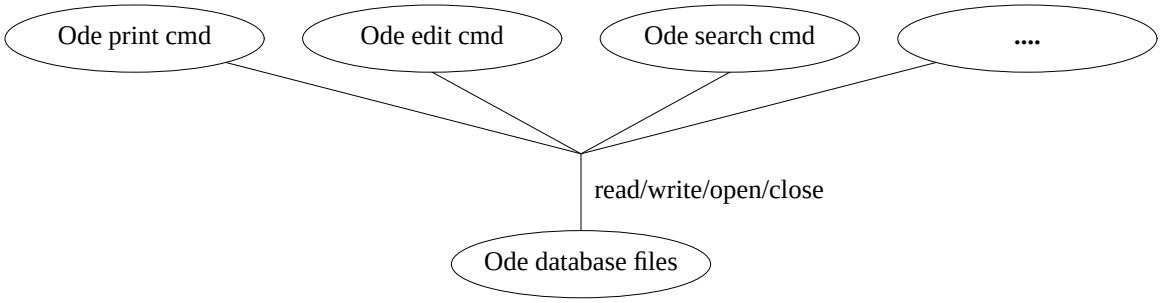

Figure 1: Conventional Approach To Object Manipulation

Figure 2 illustrates the OdeFS approach. Instead of a separate set of Ode utility programs, we just use the standard file manipulation programs. When run on files in OdeFS, the operating system directs those file operations to the OdeFS server. The OdeFS server translates those into O++ operations, which the O++ compiler and storage manager apply to the Ode database files.

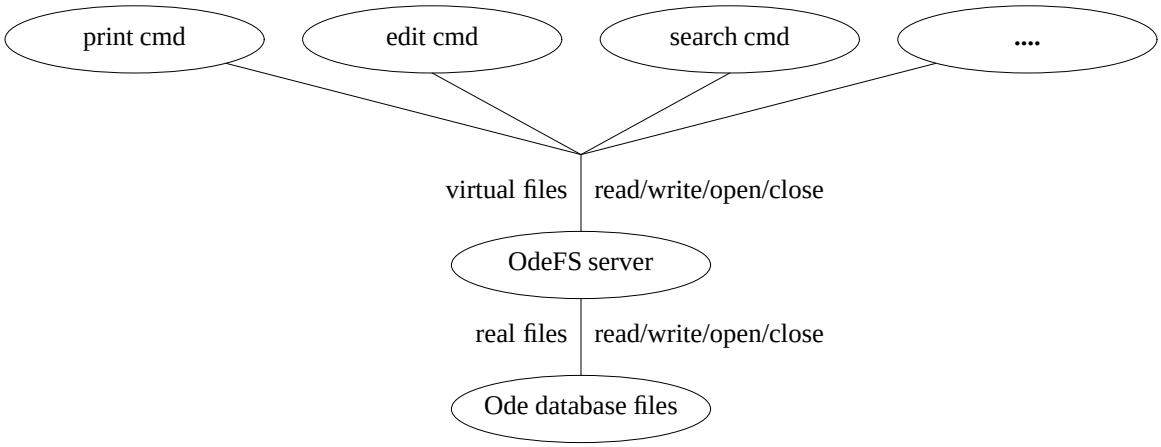

Figure 2: OdeFS Approach

To see the advantage of OdeFS, suppose we have a large, popular, file manipulation program, such as a text editor, and we want to use it to manipulate objects. With OdeFS, we can simply run the program: anything that manipulates files can manipulate objects. Without OdeFS, we would have to rewrite it, turning file references into O++ operations—and then maintain the rewritten version.

Besides OdeFS, the Ode object database provides several object-compatible interfaces to Ode that are targeted to different kinds of users:

1. O++ [1, 2, 4, 8, 9]: A programming interface for programmers, especially C++ programmers.
2. CQL++ [7]: An interactive SQL-like interface for the relational database user.
3. OdeView [3, 6]: A user-friendly graphical interface for the non programmer.

An additional goal is that OdeFS should be object-compatible with these other Ode interfaces. It should be possible to create objects with one interface, and manipulate them with other interfaces. As for efficiency, our goal is that OdeFS should ‘‘feel like a file system’’ to an interactive user.

2.2 NFS

We decided to implement OdeFS as a network file server, using the NFS protocol [10]. As in Figure 3, client computers treat OdeFS just like any other NFS server. The kernel in a client computer translates system calls on OdeFS files into NFS requests to the OdeFS server.

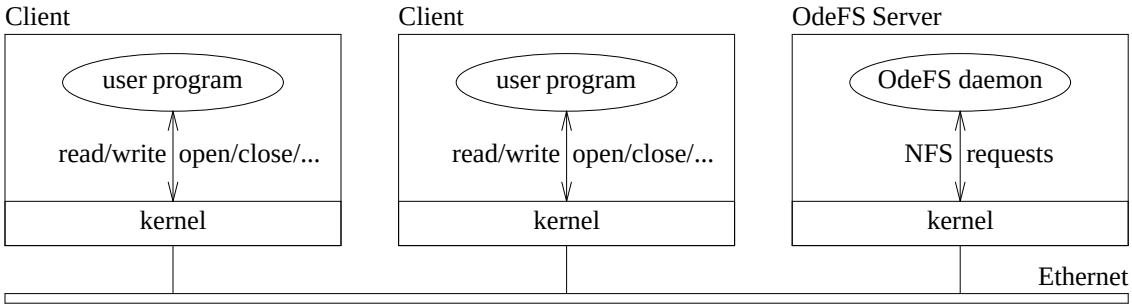

Figure 3: OdeFS as NFS Server

To be precise, when a program on a client computer issues a system call for an OdeFS file, the client’s kernel translates the call into an NFS request message, sends it to the OdeFS computer, and waits for a reply. The kernel of the OdeFS computer gets these request messages from the network and delivers them to the OdeFS deamon process. This is part of a general remote procedure call facility; the OdeFS computer’s kernel does not know that the OdeFS deamon process is really providing NFS service. The deamon process could be in the kernel, but for OdeFS, it is easier for it to be a user-level process. The OdeFS deamon process implements an NFS request by issuing various system calls, and when done, the deamon sends a reply message to the client. This uses standard services provided by the operating systems of the client and server computers; no kernel modifications are required. To the client’s kernel, the OdeFS server looks like any other NFS server; to the OdeFS kernel, the OdeFS deamon looks like any other server process.

The advantage of implementing OdeFS as a network file server is that it can run as a user-level program, and multiple clients can easily access it. The advantage of using an existing protocol, such as NFS, is that many computers already use it; we don’t have to change the kernels of the client computers. We chose NFS over other protocols because it is simple and is widely used.

There are some disadvantages to using NFS. For example, message delivery is not guaranteed, so an NFS client repeats a request if it doesn’t get a reply quickly enough. One consequence is that OdeFS must reply quickly to each request: OdeFS cannot have arbitrary delays. Another consequence is that OdeFS can see the same request several times. NFS servers must ensure that operations can be repeated safely; we have to design the OdeFS semantics accordingly.

3. OVERVIEW OF ODE AND O++

O++ is C++ extended with persistence: objects may exist between invocations of programs. In O++, persistence is a property of the object instance, not the class: objects of any class can be persistent, and some objects of a class can be persistent, while others are transient. In O++, the `pnew` operator creates a new persistent object, and is otherwise identical to the C++ `new` operator.

Here is an example of a simple `employee` class, with name and salary attributes.

```
class employee {
public:
    char name[32];
    int salary;
    employee(char* n, int s) {strcpy(name,n); salary=s;}
                                              // constructor function
};
```

We now illustrate some basic manipulations of objects in the O++ language: creating, locating, updating, and deleting a persistent object. In the next section we will show how the same operations are performed in OdeFS.

To create employee objects, we have to write and compile an O++ program that calls `pnew` with the appropriate names and salaries:

```
main()
{
    pnew employee("Joe", 50000);
    pnew employee("Susan", 75000);
    pnew employee("Nam", 120000);
}
```

The `pnew` operator returns a persistent pointer, in this case of type `persistent employee*`. O++ groups all persistent objects of the same type into a *type extent*, or just *extent.* The O++ `for` statement can be used to iterate over all persistent objects in a given extent. Thus the following program prints the name and salary for all persistent `employee` objects:

```
main()
{
    persistent employee* pe;
    for (pe in employee) {
        cout << "Name =" << pe->name <<
                ", salary =" << pe->salary << endl;
    }
}
```

The O++ `for` statement takes an optional `suchthat` clause. This specifies a predicate, and O++ only executes the body of the `for` statement for objects that statisfy the predicate. Thus the following program finds Joe's object, and then sets his salary to 70,000:

```
main()
{
    persistent employee* pe;
    int found = 0;
    for (pe in employee) suchthat (strcmp(pe->name, "Joe") == 0) {
        pe->salary = 70000; found = 1; break;
    }
    if (!found)
        cerr << "Can't find Joe!" << endl;
}
```

Finally, the `pdelete` operator deletes a persistent object; the following program removes Joe's object:

```
main()
{
    persistent employee* pe;
    for (pe in employee) suchthat (strcmp(pe->name, "Joe") == 0)
        pdelete pe;
}
```

4. ODEFS EXAMPLES

We now demonstrate how OdeFS may be used to manipulate `employee` objects. First we perform the same create, access, update, and delete operations as in Section 3; then we illustrate some additional features of OdeFS.

4.1 Simple Operations

In OdeFS, an *extent directory*, or *edir*, corresponds to an Ode type extent. The following commands change to a previously-created extent directory for `employee` objects and lists its contents (the `$` is the shell prompt):

```
$ cd employee
$ ls
+addobj     +class    +class.h  +methods.c  +objects
```

The names beginning with + are reserved by OdeFS (the + is for O++). These *control files*, each have some special semantics. For example, OdeFS requires the user to provide a `+class.h` file to define the objects in this extent.

```
$ cat +class.h
class employee {
public:
    char name[32];
    int salary;
    employee(char* n, int s) {strcpy(name,n); salary=s;}

    static persistent employee* ofsCreate(const char* buff, int len);
    int ofsWrite(const char* buff, int len);
    int ofsRead(char* buff, int maxlen);
    void ofsName(char* buff, int maxlen);
    static persistent employee* ofsFind(const char* name);
};
```

This is the same as the `employee` class defined earlier, with a few additional member functions (`ofsCreate`, etc.), which are required by OdeFS. The `+methods.c` file gives the code for the member functions of this class, and is given in the Appendix.

`+addobj` is a directory that is used for creating new objects in the extent. To create a new object, the user creates a file in the `+addobj` directory and writes to the file. The following creates three new `employee` objects:

```
$ echo "Name=Joe, Salary=50000" > +addobj/joe
$ echo "Name=Susan, Salary=75000" > +addobj/susan
$ echo "Name=Nam, Salary=120000" > +addobj/nam
```

When the user creates a new file in `+addobj`, OdeFS calls the class's `ofsCreate` function, passing it the data written by the user. `ofsCreate` parses that data, creates a new persistent `employee` object, initializes it appropriately, and returns a persistent pointer to it.

If we list the `+addobj` directory after this, we see files for the new objects:

```
$ ls +addobj
joe    nam    susan
```

These are called *object files* (or *ofiles*). An object is displayed by reading its ofile:

```
$ cd +addobj
$ cat joe
Name=Joe, Salary=50000
```

When the user reads an ofile, OdeFS calls the class's `ofsRead` function on that object. That function determines the representation of the object which OdeFS returns as the contents of the ofile. The user can run any command on these ofiles: `grep`, `pr`, `spell`, etc. File name expansion can be used, so the following displays all three ofiles:

```
$ cat *
Name=Joe, Salary=50000
Name=Nam, Salary=120000
Name=Susan, Salary=75000
```

An object is updated by writing its ofile. OdeFS then calls `ofsWrite`, which (like `ofsCreate`) parses the data written by the user, and updates the object appropriately. Thus the following changes Joe's salary:

```
$ echo "Salary=70000" > joe
```

We didn't need ''Name=Joe'' because the name didn't change; the `ofsWrite` function for this class doesn't change omitted fields. The user can think of the data written to an object file as update instructions to be interpreted by `ofsWrite`, rather than a complete new specification of the object. This command updates Joe's object, instead of creating a new object, because we wrote to an *existing* ofile in `+addobj`.

Thus OdeFS makes an ofile look like an ordinary file by translating ofile read and write operations into calls to that object's `ofsRead` and `ofsWrite` functions. Those functions determine the structure of the data in an ofile; OdeFS itself doesn't specify the ofile format. Different classes can have different ofile formats, and ofiles do not have to be printable.

An ofile is an alias for an object. Removing an ofile removes that alias, but does not remove the object itself. To remove the object as well as the alias, use the special command `rmobj`.

```
$ rm joe      # remove alias, but not object
$ rmobj nam   # remove object plus alias
```

Clearly, two different operations are required, one to remove the alias, and the other to remove the object itself. We chose the specific operation names because we do make the users aware of the fact that what they have is a handle on an object so that doing an `rm` should be like removing a link in UNIX. When an object is to be removed, using the handle, a new command, `rmobj` is introduced, parallel to the `rmdir` command in UNIX.

4.2 Advanced Operations

Ofiles exist in *object directories*, or *odirs.* `+addobj` is a built-in odir. Each odir contains a set of (handles to) objects. To create a new odir, create an empty directory, and put ofiles into it with the `mv` or `ln` (link) commands:[1]

1. This example assumes that `nam` and `joe` have not been deleted.

```
$ mkdir myset
$ mv susan nam myset
$ ln joe myset/my_joe
$ ls myset
my_joe    nam    susan
$ ls .
joe
```

The two ofiles named `joe` and `my_joe` are aliases for the same object. The ofile names are arbitrary; users can choose any names they like. An object can have any number of aliases: all refer to the same object.

Each extent directory has a built-in odir named `+objects`, which has an ofile for every object in the extent:

```
$ cd ..
$ ls +objects
Bill    Carol   Joe     Nam     Susan
```

Here `Bill` and `Carol` are employees who existed from before. `Joe`, `Nam`, and `Susan` are the employees we just created. When these objects were created, entries for them automatically appear in the `+objects` directory. The ofile names in `+objects` are determined by the `ofsName` member function; for the `Employee` class, it returns the `name` field. This is why the name here is `Joe`, rather than the name `joe` used when the file was created in `+addobj`. Ordinary commands work on the ofiles in `+objects`: the user can search for strings in the objects, print them, etc.

Since all standard UNIX tools work on ofiles, we could locate a specific ofile by using `grep`'s option to report the names of matching files instead of matching lines:

```
$ cat `grep -l Susan *`
Name=Susan, Salary=75000
```

This is fine as a quick and dirty way to locate an object, but it has two potential problems. One is that in a large collection, `grep` would have to read every object. We would like to be use index structures, where available. The second problem is that `grep` does a string match with no regard for attribute semantics. Thus the following attempt to locate Nam's ofile unexpectedly displays all employees:

```
$ cat ‘grep -l Nam *‘
Name=Bill, Salary=60000
Name=Carol, Salary=85000
Name=Joe, Salary=50000
Name=Nam, Salary=120000
Name=Susan, Salary=75000
```

The problem is that every ofile starts with the string ‘‘Nam,’’ so all ofiles match the simple search criteria.

A better way to get an ofile for a particular object is to use a *file name query*. In an extent or object directory, a name of the form `+Name=Nam` is an ofile for Nam's object:

```
$ cat +Name=Nam
Name=Nam, Salary=120000
```

The user can treat this like any other ofile. This isn't magic; when the user opens a file name that starts with + and contains =, OdeFS calls that class' `ofsFind` function, with the file name as the argument. That function interprets the file name as a simple query, locates the matching object, and returns it. OdeFS then calls `ofsRead` to read the object. If the query syntax is invalid, or there is no such object, `ofsFind` returns NULL, and OdeFS gives the user a ‘‘file not found’’ error.

NFS clients timeout and repeat a request if a response is not received quickly (for Sun workstations, the default timeout is 0.7 seconds). Therefore `ofsFind` must locate the object quickly. As a result, for large extents, `ofsFind` should only search on indexed fields.

In general the user can treat ofiles and odirs like ordinary files and directories, but there are some differences, particularly with respect to the `cp` (copy) command. When `cp` copies an ofile, the result is an ordinary UNIX file (*ufile*), not an ofile. The reason is that when a file is copied, OdeFS just sees a series of writes to the destination file. OdeFS cannot tell whether this data came from an ofile or a UNIX file or from the keyboard. This differs from the `mv` command; `mv` maps into a single file system operation (`rename`) which specifies the source and destination files. For example, consider:

```
$ cp +objects/Susan Susan
```

This creates a ufile named `Susan`. The ufile is independent of the object. The ufile's initial contents are the same as the object's representation, but updating the ufile won't change the object, and updating the ofile won't change the ufile. The only exceptions are copying an ofile into the `+addobj` directory, which creates a new object, and copying it into an existing ofile, which updates that ofile's object.

5. DESIGN ISSUES

Before describing the OdeFS architecture in detail, we would like to discuss, in this section, a few of the key issues that we faced, and the design decisions we made.

5.1 Typed Files and Directories

OdeFS has more complex semantics than a standard file system. To deal with this, OdeFS defines several file and directory types. These types simplify the description of the semantics of OdeFS and control the operations which OdeFS allows on files and directories. These directory and file types are logical types. Client operating systems do *not* know about these logical types: as far as the operating systems are concerned, all OdeFS directories are directories, and all OdeFS files are files.

The central goal of OdeFS is to have objects look like regular files to the user. However, OdeFS cannot store objects in individual files; such storage would be prohibitively expensive, and also would be incompatible with the Ode storage manager. Therefore, OdeFS distinguishs between files representing objects (called *ofiles*), and regular UNIX files (called *ufiles*). An ofile provides a user-readable representation of an object. OdeFS converts reads and writes to an ofile into calls to member functions on that object. Those member functions determine the format of the ofile. A ufile is an ordinary UNIX file: OdeFS simply transmits operations on such a file to the standard UNIX file system. In addition, users must communicate with OdeFS. Since file system operations are the only way to reach OdeFS, we introduce the notion of a *control file* or *cfile* type. Users read cfiles to get information from OdeFS; users update/create/delete cfiles to give information to OdeFS, or to ask OdeFS to perform various actions. The type of a file can be determined by its name and the type of the directory in which the file appears.

OdeFS directories are also divided into three types. UNIX directories (or *udirs*) contain regular UNIX files, while object directories (or *odirs*) contain object files. Ufiles are not permitted in odirs, and ofiles are permitted only in odirs: this restriction helps users distinguish ofiles from ufiles. When creating a new type extent, the user must provide OdeFS with various information about that extent. A natural way to do that is by inventing another directory type: an *extent directory* (or *edir*). There is one edir for each extent. Users create an extent by defining cfiles in an edir; users get information about an extent by reading cfiles in the edir.

The type of a directory is specified by creating a (zero-length) control file, one of `..edir`, `..odir`, or `..udir` in that directory. An empty directory is untyped. Whenever possible, OdeFS attempts to deduce the type of a directory from the first operation on the directory. For example, if the user moves an object file into an empty directory, OdeFS assumes the user want it to be an object directory, and automatically creates a `..odir` file. If the user create an ordinary file in an empty directory, OdeFS automatically makes it a UNIX directory (udir).

Here is a summary of the file and directory types used by OdeFS:

file type	explanation
ufile	UNIX file
ofile	object viewed as a file
cfile	control file, for communication

directory type	explanation
udir	UNIX directory
odir	object directory for ofiles
edir	extent directory, defines a class

5.2 Naming

The type system in Ode can be an arbitrary DAG (directed acyclic graph) matching the C++ type derivation tree. A UNIX file system, on the other hand. must be a tree. Therefore the OdeFS directory hierarchy cannot mimic the object class hierarchy. Furthermore, users often want to work in directories that are ''close'' to arbitrary sets of objects. Such work would not be possible if the objects were maintained in an isolated hierarchy. Therefore OdeFS allows edirs and odirs to appear anywhere in the file system hierarchy, with arbitrary names.

The same philosophy carries over into the naming of object directories, object files, and so forth. The user can always choose a name of his or her liking, even if it differs from what the class designer or object creator had in mind.

However, OdeFS—and users—also need to identify cfiles, and the only practical way to do that is by the file name. Therefore OdeFS reserves all names that start with `+`, or which contain a `++`, for use as cfile names or other built-in names. This rule is more restrictive than necessary, but it is simple, easy to remember, and allows for expansion. For example, in an edir, `+class.h` is a cfile that defines the extent's class. OdeFS could allow a user to create a ufile named `+class.h` in a udir; however, we think that flexibility would confuse users more than it would help them.

Also, OdeFS reserves all names starting with a period, but only in odirs and edirs. The `ls` command will not normally list such names, and they do not normally participate in file name expansion. Therefore OdeFS uses such names for cfiles which should not normally be visible to users.

5.3 Pseudo-files

OdeFS uses the notion of a *pseudo-file*. to provide information about some file attributes. some cfiles and ofiles are pseudo-files, others are real files. Ufiles are never pseudo-files. Pseudo-files can be accessed explicitly by name, but they do not appear in directories, and they do not participate in file name expansion. In particular, they are not visible when a directory is listed, and cannot be made visible. Pseudo-files differ from files whose names which start with a period; those names exist, and `ls` will display them if given the appropriate argument.

OdeFS uses pseudo-files in two situations. The first is for names that would cause too much clutter when enumerated in a directory listing. For example, if `joe` is an ofile, then OdeFS has a number of pseudo-files which give additional information about the object: `joe++object`, `joe++class`, `joe++err`, etc. These pseudo-files are useful occasionally, but most of the time users do not need them, and do not want to see them[2].

OdeFS also uses pseudo-files for names that are impractical to enumerate. An example is a file-name query, such as `+Name=Joe`, as mentioned in Sec. 4.2. Because such queries are evaluated by a class member function, OdeFS cannot possibly enumerate all valid names of that form.

---

2. We considered hiding these names by using a prefix that starts with a period, such as `.object.joe`. We decided not to use prefixes because shell procedures often take ofile names, like `joe`, as arguments and generate these names internally. With a prefix convention, a shell would have insert the prefix before the last leaf name in the argument (e.g., transform `myodir/joe` into `myodir/.err.joe`). This is much harder than appending a suffix.

5.4 User Interface

Although it is possible to do any OdeFS operation with ordinary file system commands, we have found it useful to provide a few ‘‘user interface’’ shell commands for OdeFS, to automate common operations.

For instance, the user interface commands `mkedir`, `mkodir`, and `mkudir` create a directory and specify its type. These commands first create a new directory, and then specify the type to OdeFS by creating a file named `..edir`, `..odir`, or `..udir` in that directory. We have done so in the belief that a user may prefer to say

```
$ mkodir foo
```

even though this is not a normal UNIX shell command, rather than say

```
$ mkdir foo
$ touch foo/..odir
```

5.5 OdeFS Member Functions

OdeFS works by translating file system operations on ofiles into calls to member functions for the class. Section 6.2 describes these functions. OdeFS itself does not ‘‘look inside’’ a class; instead, these required member functions tell OdeFS all it needs to know about the class. To use a class with OdeFS, someone must write those functions. They are relatively simple, and OdeFS provides a package for dealing with common ofile formats, but using OdeFS does require that extra programming effort up front. However, once those functions have been written, any existing program can manipulate objects of that class. In most cases, this more than makes up for the extra programming effort.

To minimize this programming burden, in the future, we may provide an ‘‘OdeFS member function generator’’ tool. This would be a separate program; it would read a class definition, and would create initial versions of the required OdeFS functions, using a specified ofile format. We think this would best be done via a separate program, rather than as part of OdeFS itself.

An alternative would be to require all OdeFS classes to be derived from a common base class (say `ofsBase`). This base class could define these required functions and provide default versions. There are two reasons we have not done that. First, these functions depend heavily on the data members in the class itself, and we cannot provide useful default versions. And second, that would make OdeFS objects incompatible with the other Ode interfaces. For example, suppose we have `employee` objects in an existing Ode database. We can access them via OdeFS by creating an edir with the OdeFS functions added to the class definition (adding member functions to a class doesn’t bother the Ode storage manager). However, the Ode storage manager does not allow us to change the derivation of the class; if OdeFS uses an `employee` that is derived from a `ofsBase` class, then the Ode storage manager would treat that as a different class from the existing `employee` class.

5.6 Long Duration Requests

When a client sends a request to an NFS server, if the client does not get a response quickly enough, the client repeats the request, and gives up after a certain number of retries. Some client operating systems allow users to specify the timeout, but some do not. The timeout is usually fairly short, appropriate for reading or writing of a few blocks. This is reasonable for simple operations like reads and writes. However, it is risky to assume that complex operations, such as evaluating an arbitrary querys or compiling a new class, can be done within this time limit.

To get around this problem, we have designed a simple protocol, which we use for all (potentially) lengthy operations. The user initiates an operation by updating a cfile whose name ends in `.b` (for ‘‘button’’ or ‘‘begin’’). When done, OdeFS updates a cfile whose name ends in `.e` file (for ‘‘end’’ or ‘‘error’’). OdeFS puts the error messages for this operation, if any, in this file. To wait for OdeFS to finish, the user periodically checks the `.e` file until it exists and is newer than the `.b` file. If there are no errors, the `.e` file will be empty.

While this is a simple protocol, it is annoying for users to do manually. Therefore we provide user interface shell procedures for such operations. For example, `ccedir` compiles a class definition in an edir, and

reports the errors, if any:

```
$ ccedir
```

`ccedir` is a simple shell procedure, and looks something like:

```
touch .compile.b
while [ ! .compile.e -nt .compile.b ]
do
    sleep 1
done
if [ ! -s .compile.e ]
then
    echo "Compilation successful"; exit 0
else
    echo "Compilation failed"; cat .compile.e; exit 1
fi
```

This shell script loops until `.compile.e` exists and is newer than `.compile.b`, and then tests if `.compile.e` is zero-length.

5.7 Ownership and Protection

In general, OdeFS provides standard UNIX file system permissions for all directories and files. Thus there are separate read, write, and execute permissions, available to owner, group, and other, with each of these being independently settable. Standard commands (`chmod`, `chgrp`, etc.) set these attributes.

OdeFS does apply additional restrictions to some cfiles. Usually these are natural and obvious. For example, `..edir` is a zero-length cfile that marks a directory as an edir. OdeFS will not allow a user to write this file, and OdeFS will not allow a user to delete it unless it is the only file left in the directory.

Finally, for an ofile, OdeFS currently uses the owner, group, and read/write permissions of the object's edir. Thus all objects in an extent have the same permissions. We do not regard this as an ideal solution. The problem is that Ode does not provide per-object permissions. We are trying to decide if per-object permissions are needed, and if so, whether they should be added to Ode itself, or provided by OdeFS as a value-added service.

6. DETAILED DESCRIPTION

This section describes the OdeFS semantics in detail: how to create extents and manipulate objects, what types of control files we use and what side effects they have, etc. If this description may make OdeFS appear hard to use, remember that this section concentrates on the ways in which OdeFS *differs* from an ordinary file system. The ways in which OdeFS behaves like a file system—manipulating objects by reading and writing ofiles—need no special description.

6.1 Extent Directories

An extent directory describes a type extent. It has the following control files and built-in directories:

`..edir`	A zero-length file identifying this as a extent directory.
`+class`	A file containing the name of the class of the objects in this extent (OdeFS needs this because `+class.h` can define more than one class).
`+class.h`	The O++ specification of the class of the objects in the extent.
`+methods.c`	The O++ specification of the member functions for this extent.
`.compile.b`	When the user creates or touches this file, OdeFS compiles the class.
`.compile.e`	Error messages from the last compilation attempt.
`+objects`	A built-in odir with an ofile for every object in this extent.
`+addobj`	A built-in odir for creating new objects.

In addition, an extent directory can have any type of sub-directory, and can have arbitrary UNIX files (additional code, comments, documentation, etc).

The `mkedir` command creates a new extent directory (the command creates an empty directory and places a zero-length `..edir` file in it). OdeFS then creates the `+addobj` and `+objects` directories. The extent directory can have any name, although it is most natural to use the class name.

The `+class`, `+class.h`, and `+methods.c` files must be created explicitly by the user. OdeFS allows class derivation. `+methods.c` should include the `+class.h` files for this extent and for all extents upon which the new class depends. At this point, OdeFS treats these as ordinary files, and does not interpret them. Thus they can be written multiple times, removed and recreated, etc.

When ready, the user asks OdeFS to compile the new extent and add it to the Ode database. The user does this by touching `.compile.b`. OdeFS compiles the `+class.h` and `+methods.c` files, and registers the class with a *catalog*. When done, OdeFS creates `.compile.e`, with the error messages, if any. As illustrated in Section 5.6, the user can use the shell procedure `ccedir` to compile an extent. Once the extent has been compiled, the user can create new objects. Once compiled, the `+class`, `+class.h`, and `+methods.c` files cannot be changed or recompiled.

When the user creates a new file in `+addobj`, OdeFS calls `ofsCreate()` to create the new object (see below). OdeFS maintains `+objects` as a read-only odir with all objects in this extent.

6.2 Required Member Functions (Methods)

Every OdeFS class must define the following member functions:

```
static persistent class* ofsCreate(const char* buff, int len);
int ofsRead(char* buff, int maxlen);
int ofsWrite(const char* buff, int len);
void ofsName(char* buff, int maxlen);
static persistent class* ofsFind(const char* name);
```

The Appendix gives example definitions of these functions. OdeFS calls `ofsCreate` to create a new object: that is, when the user creates a new file in the `+addobj` directory. The arguments specify the data written by the user. `ofsCreate` creates a new object, by calling `pnew`, initializes it with the values written by the user, and returns a persistent pointer for the object.

If `ofsCreate` cannot create an object—for example, if the user's data is in the wrong format—`ofsCreate` calls function `ofsErrMsg` to specify an error message, and returns NULL. OdeFS will make the error message available to the user. `ofsErrMsg` is a global function provided by OdeFS.

OdeFS calls `ofsRead` when the user reads an ofile for an object. `ofsRead` creates the ofile representation in the `buff` argument. The representation is limited to `maxlen` bytes. `ofsRead` returns the number of bytes in the ofile representation.

OdeFS calls `ofsWrite` when the user writes an ofile; the arguments specify the data written by the user. `ofsWrite` updates the object appropriately. If successful, `ofsWrite` returns 1. If unsuccessful, `ofsWrite` returns 0, after calling `ofsErrMsg` to specify an error message.

`ofsName` copies the default ofile name for this object into the buffer specified by the arguments. OdeFS uses this name in the `+objects` directory. The name must not start with a `+` or a period, and it must not contain a `+` or a `/`. It should be printable. If practical, the name should be unique within an extent; but if two objects have the same name, OdeFS will add a disambiguating suffix.

OdeFS calls `ofsFind` when the user specifies a file name query: that is, asks for an ofile whose name is of the form `+foo=bar`. The argument is the file name, without the leading `+`, such as `foo=bar`. `ofsFind` parses the file name, and if it specifies an object, returns a persistent pointer for it. If not, `ofsFind` returns NULL; OdeFS then returns a ''file not found'' error to the user. As mentioned earlier, `ofsFind` must be reasonably fast. Otherwise, an NFS client times out and reports an error. In case of large extents, `ofsFind` should therefore perform searches using indexed fields only.

There are some restrictions on what these functions can do. First, the functions cannot have any global variables; all the state must be contained in the objects. Only `ofsCreate` and `ofsWrite` can update objects. `ofsWrite` can interpret the data written by the user as an arbitrary update request; for example,

`ofsWrite` can leave unspecified data fields unchanged. However, `ofsWrite` must be idempotent; that is, the resulting value of the object must be the same no matter how many times `ofsWrite` is called with the same data (because of timeouts and repeats, OdeFS can see the same request several times, and hence can call `ofsWrite` several times). In practice this is not a serious limitation, but it prohibits `ofsWrite` from providing ‘‘increment’’ updates. And finally, OdeFS does guarantee that `ofsCreate` is called only once. However, OdeFS does so by transforming duplicate requests into calls to `ofsWrite`, so `ofsWrite` should accept the same data as `ofsCreate`.

6.3 Object Directories

An object directory contains an arbitrary collection of objects (ofiles), all of the same class or subtypes of that class. An object directory has two control files: `..odir`, which is a zero-length file identifying it as an object directory, and `.extent`, which is a symbolic link to an extent directory, identifying the type of the objects in the odir. Therefore the following displays the class definition for all objects in an odir:

```
$ cat .extent/+class.h
```

An optional second argument to the shell program `mkodir` can be used to specify `.extent` at odir creation time. OdeFS allows the user to remove or change `.extent`, that is, to specify the restriction of the type on all objects in the odir, whenever the odir has no ofiles.

Each extent directory has two built-in odirs: `+objects` and `+addobj`. `+objects` has an ofile for every object in the extent. The directory is read-only, in that the user cannot add ofiles to it, remove ofiles from it, or rename ofiles in it. However, the user can update objects by updating ofiles in `+objects`. Strictly speaking, the user should not be able to move an ofile from `+objects` to another odir. However, because that is a common operation, OdeFS allows it, and creates the destination ofile as a new alias for the object, but does *not* remove the source ofile from `+objects`.

`+addobj` is a writable odir, with the property that creating a new file in it creates a new object (OdeFS calls the extent’s `ofsCreate` function to create the object). Once created, the ofile becomes an alias for the object, and subsequent writes update that object. When creating a new object, the user should first ensure that the file name does not exist in `+addobj`. Otherwise, instead of a creating a new object an existing object will be updated.

Incidentally, an alternative would be to eliminate the `+addobj` odir, and allow users to create objects by creating ofiles in `+objects`. We decided not to do that because the ofile names in `+objects` are the ‘‘official’’ names given by the `ofsName` function. We could not guarantee these names if we allowed users to create ofiles in `+objects`.

6.4 Object Files And Per-Object Control Files

An object file (ofile) is an alias for an object. As described above, OdeFS translates read/write operations on ofiles into calls to member functions of the class. Those functions determine the ofile format.

6.4.1 Creating Ofiles. Creating a file in the built-in odir `+addobj` creates a new object and an alias for it. OdeFS provides an ofile for every object in the built-in odir `+objects`. Except for that, the only way to create a new ofile is to rename an existing ofile, via `mv`, or to create a duplicate of an existing ofile, via `ln`. Removing an ofile removes that alias, but does not remove the object itself.

In an edir or an odir, OdeFS interprets a file name that starts with a `+` and contains an `=` as a pseudo-ofile for a ‘‘file-name’’ query. OdeFS calls the extent’s `ofsFind` function to evaluate that query and locate the specified object, if any. A pseudo-ofile can be used just like an other ofile: the user can display it, update it, rename it, etc. However, when renaming a pseudo-ofile, the new name must not start with a `+`. Thus the first move fails, but the second succeeds:

```
$ mv +Name=Joe myodir      # fails; tries to create myodir/+Name=Joe
$ mv +Name=Joe myodir/joe  # okay
```

6.4.2 Per-Object Pseudo-Files. There are several control pseudo-cfiles associated with every ofile. To access them, append a suffix to the ofile name. These are pseudo-files because we do not want to clutter up a directory by listing all of them for every ofile. These pseudo-files include:

ofile`+err`: The error message(s) associated with the most recent attempt to update this object. The methods generate these error messages by calling `ofsErrMsg` (see Section 6.2). This file exists only if the last update failed. Thus if an attempt to update `joe` fails, the user can print `joe+err` to find out why.

ofile`+extent`: A symbolic link to this object's extent directory. While all objects in an odir must be of the same type, specified by the `.extent` file, individual objects may be of more specific (sub-) types. For instance, a `Person` odir may have an ofile in it for an object of type `Student`, a sub-type of `Person`.

ofile`+class`: The type name of this object. This is the same as *ofile*`+extent/+class`.

ofile`+object`: The ''raw bits'' of the object itself. Removing this pseudo-file removes the object itself, plus the ofile alias. The user can see the raw object by dumping this file. A privileged user can update the object by writing this file, but that is *not* the recommended update mechanism.

As we accumulate experience with OdeFS, we may add new pseudo-files types, as necessary. The conventions we have chosen make such extension easy.

6.4.3 Removing Objects. The user interface command `rmobj` deletes the objects whose ofile names are passed as arguments. For each argument of file, *name*, this command just removes *name*`+object`, after checking that *name* really is an ofile. The advantage of this command is that the user can use file name expansion to remove objects. For example, the following command removes all objects whose ofiles are in `myodir`:

```
$ rmobj myodir/*
```

The following command removes all the objects in an extent:

```
$ rmobj +objects/*
```

When the user deletes an object, OdeFS automatically removes that object's ofile from the built-in odir `+objects`. However, OdeFS does not immediately remove that object's ofiles from other odirs. Instead, any outstanding aliases to the removed object become invalid; OdeFS removes an invalidated ofile the next time an access is attempted.

6.5 Other Features

OdeFS has several features that space does not allow us to describe here in detail. For example:

- The user can ask OdeFS to populate an odir with ofiles for the objects that match a CQL++ query [7]. The query is entered into a `.query` file in an empty odir. The query is then applied by touching the `.query.b` file. OdeFS executes the query. If successful, it creates ofiles corresponding to the objects that satisfy the query and a zero-length `.query.e` file. In case of an error, OdeFS creates only a `.query.e` file containing an error message. For instance, if the type of the objects returned by the query is not the same as that specified in the `.extent` file of the odir, an error occurs.
- OdeFS supports Ode sub-extents: they appear as sub-directories of the main edir. The user can move objects between sub-extent directories as required.
- OdeFS provides a bulk-create facility for creating multiple objects by writing a single file.
- OdeFS supports multiple Ode databases; when creating an extent, the user can specify which database it is associated with.
- OdeFS supports transactions. A variant of the long query mechanism is used for this purpose. The user interface on top of this is written to provide `tbegin` and `tend` as commands at the shell level.

7. IMPLEMENTATION

OdeFS is built on top of a conventional file system; it acts as a filter between the user and the file system. OdeFS is implemented as several processes (Figure 4):

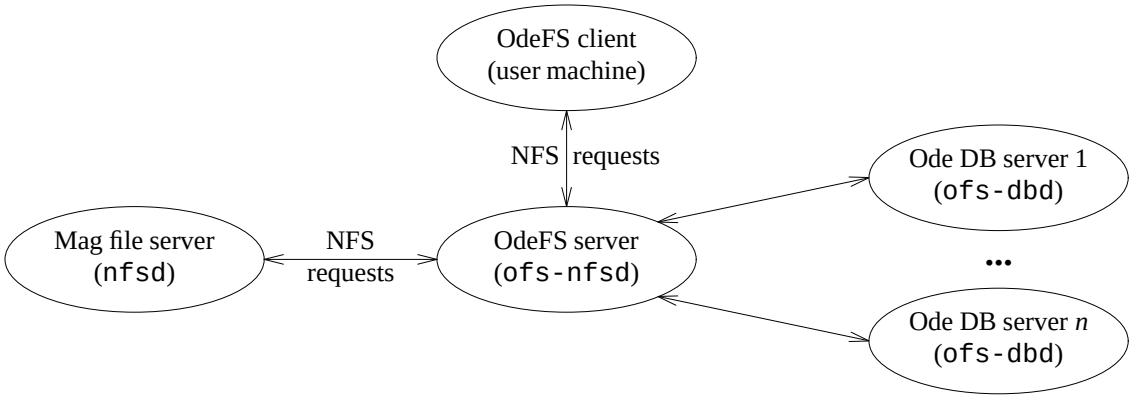

Figure 4: OdeFS Implementation

The `ofs-nfsd` process (''OdeFS NFS daemon'') is a user-level NFS server process; it accepts NFS requests from various clients and returns the appropriate replies. `ofs-nfsd` is also a client of other server processes, `nfsd`, the regular magnetic disk file server,[3] and `ofs-dbd`, a separate O++ program which handles Ode requests. There is one `ofs-dbd` process for each Ode database used by OdeFS; `ofs-nfsd` starts these processes as needed. Normally there are several `ofs-nfsd` and `nfsd` processes; they act as a worker pools, taking requests from a common queue. Currently there is only one `ofs-dbd` process per database, because Ode does not yet allow concurrent access to a database. Once that restriction is lifted, the `ofs-dbd` processes can be replicated as needed. Normally the `ofs-nfsd`, `ofs-dbd`, and `nfsd` processes run on the same computer, and once started, they run forever. We use the standard NFS interface between `ofs-nfsd` and its clients and between `nfsd` and `ofs-nfsd`. The interface between `ofs-nfsd` and `ofs-dbd` is also a request-reply mechanism, in that an `ofs-nfsd` process sends requests to a `ofs-dbd` process. The request types are similar to NFS requests, but the transport mechanism is much simpler; we assume that they are running on the same machine type, so we pass ordinary structures, and we currently use named pipes (fifos) for sending request and reply messages.

There are several reasons for this process structure. We use a separate magnetic disk file server process, `nfsd`, to simplify installation and administration, and to reuse existing file system tools. OdeFS is built on top of an ordinary file system; most administrative problems — disk partitioning, formatting, backup and recovery, checking, etc. — are handled by existing file system mechanisms.

We have separate Ode database server processes for protection. OdeFS must execute arbitrary, user-provided code (the required member functions described in Section 6.2); this could be buggy, or even be malicious. All such code is isolated to the `ofs-dbd` processes[4]. Each runs with the user id of its database, rather than as superuser. The worst that can happen is that buggy user code could damage other classes in its own database, and/or hang its `ofs-dbd` process. User code cannot damage other databases or the `ofs-nfsd` process. With suitable timeouts, the `ofs-nfsd` process can detect a hung `ofs-dbd` process and gracefully reject requests to that database.

All directories, UNIX files, and non-pseudo control files exist as directories and files in the underlying magnetic disk file system. Each of these objects is uniquely identified by the inode number assigned by the file server. For operations on ordinary files and directories, `ofs-nfsd` just forwards the requests to `nfsd`, which actually does the work. For operations on control files, `ofs-nfsd` does the appropriate actions before or after passing them on to the underlying file system. For example, when the user updates `+class.h` or `methods.c` in an uncompiled extent directory, OdeFS just passes the updates on to `nfsd`.

---

3. For simplicity, we will refer to the underlying file system as a ''magnetic disk'' file system. In reality, OdeFS will work with any type of file system.

4. Since OdeFS permits free integration of data from multiple Ode databases, partitioning of the information into small enough database ''chunks'' is desirable for protection from buggy class definitions.

When the user updates `.compile.b`, OdeFS reads and compiles these files. After the extent has been sucessfully compiled, OdeFS passes read requests for `class.h` on to `nfsd`, but rejects any write or remove requests.

Object files (ofiles) are implemented differently: they do not exist as files in the underlying file system. Instead, process `ofs-dbd` evaluates ofiles as needed. Each ofile is uniquely identified by the persistent object pointer assigned by Ode when the underlying object was created. `ofs-nfsd` forwards ofile operations to the appropriate `ofs-dbd` process.

Object directories (odirs) have two parts. Each odir exists as a directory in the magnetic disk file system, and as an Ode object, maintained by an `ofs-dbd` process. Recall that a directory is just a list of pairs of names and identifiers. The Ode odir object has the names of the ofiles and the corresponding Ode object pointers; the directory in the magnetic disk file system has the names of the non-ofiles in the odir. `nfsd` manages the names in the file system part of the directory, and `ofs-dbd` maintains the ofile names. `ofs-nfsd` logically combines them and makes them look like one directory. To link the odir directory on magnetic disk to the Ode odir object, when creating an odir, `ofs-nfsd` writes the Ode odir object's persistent pointer into a private file (`.oid`) in the magnetic disk directory for the odir.

As an example, consider what happens when we open, read, and write a ufile, cfile, or ofile. The OdeFS client translates the open request into an NFS lookup request, which it sends to `ofs-nfsd`. The arguments are a ''handle'' for a directory, and the name of a file within that directory. The reply is a handle for the requested file or directory, plus its attributes (size, type, update time, etc.). A handle is a capability for a file system object, and is opaque to the client. OdeFS handles contain either an inode number (for files and directories) or an Ode object pointer (for ofiles), plus some type information (ufile, ofile, udir, odir, etc.). In case of a file, `ofs-nfsd` forwards the lookup request to `nfsd`, and returns a ufile handle to the OdeFS client. The OdeFS client translates read and write system calls into read and write NFS requests. `ofs-nfsd` forwards these requests to `nfsd`, because they are for a ufile.

Now suppose that the user opens a cfile. In the lookup request, `ofs-nfsd` recognizes that this is a cfile, and returns a handle indicating its type. When the client sends a read or write request for that cfile handle, `ofs-nfsd` does the appropriate action. For example, for a `+class.h` file, `ofs-nfsd` always forwards a read request to `nfsd`, but rejects a write request if the extent has been compiled.

Finally, suppose the user opens an ofile in an odir (the directory handle indicates it is as an odir). `ofs-nfsd` reads the `.oid` file to get the pointer to Ode odir object, and then sends a lookup request to the appropriate `ofs-dbd` process, asking it to lookup the name in that odir. That process returns information about the ofile, which `ofs-nfsd` then returns to its client (if there was no such ofile, `ofs-nfsd` tries `nfsd`). When `ofs-nfsd` gets a read or write request for an ofile handle, it forwards the request to `ofs-dbd`.

OdeFS takes advantage of the Ode transaction mechanism. For example, one NFS request atomically moves a file from one directory to another. When moving an ofile between two odirs, `ofs-dbd` updates those two odir objects as part of one transaction. Ode then guarantees atomicity.

OdeFS itself has no storage: files and directories are handled by the underlying file system, and objects are handled by the Ode storage manager—and are in turn stored in Ode database files. Thus OdeFS avoids most file system administration problems. For example, to backup or restore an OdeFS file system, we simply back up or restore the underlying file system.

8. CONCLUSIONS

We have described OdeFS, a file system interface to an object-oriented database system. The major benefit of this interface is that it interoperates well with the UNIX operating system, and thus provides a convenient means for users to access and manipulate objects. The central idea in the implementation is to devise a layer of code, between the client operating system and the network file system, which implements OdeFS functionality, without modifying the front end operating system or the back end file system and object manager. Objects created through OdeFS can be accessed and manipulated by other interfaces to Ode, and vice versa.

The OdeFS implementation is in its final stages. We hope to be able to report experience with actual users shortly.

ACKNOWLEDGEMENTS

We are grateful to Shaul Dar for an ever so thorough reading of an early draft, and his many useful suggestions.

REFERENCES

[1] R. Agrawal and N. H. Gehani, ''Ode (Object Database and Environment): The Language and the Data Model'', *Proc. ACM-SIGMOD 1989 Int'l Conf. Management of Data*, Portland, Oregon, May-June 1989, 36-45.

[2] R. Agrawal and N. H. Gehani, ''Rationale for the Design of Persistence and Query Processing Facilities in the Database Programming Language O++'', *2nd Int'l Workshop on Database Programming Languages*, Portland, OR, June 1989.

[3] R. Agrawal, N. H. Gehani and J. Srinivasan, ''OdeView: The Graphical Interface to Ode'', *Proc. ACM-SIGMOD 1990 Int'l Conf. on Management of Data*, 1990, 34-43.

[4] R. Agrawal, S. J. Buroff, N. H. Gehani and D. Shasha, ''Object Versioning in Ode'', *Proc. IEEE 7th Int'l Conf. Data Engineering*, Tokyo, Japan, Feb. 1991.

[5] S. R. Bourne, Addison-Wesley, 1982..

[6] S. Dar, N. H. Gehani, H. V. Jagadish and J. Srinivasan, ''Queries in an Object-Oriented Graphical Interface'', AT&T Bell Labs Technical Memorandum, 1991.

[7] S. Dar, N. H. Gehani and H. V. Jagadish, ''CQL++: An SQL for a C++ Based Object-Oriented DBMS'', *Proc. of Int'l Conf. on Extending Database Technology*, Vienna, Austria, Mar. 1992.

[8] S. Dar, R. Agrawal and N. H. Gehani, ''The O++ Database Programming Language: Implementation and Experience'', *Proc. IEEE 9th Int'l Conf. Data Engineering*, Vienna, Austria, 1993.

[9] N. H. Gehani and H. V. Jagadish, ''Ode as an Active Database: Constraints and Triggers'', *Proc. 17th Int'l Conf. Very Large Data Bases*, Barcelona, Spain, 1991, 327-336.

[10] N. NFS, *Network File System: Version 2 Protocol Specification*, Sun Microsystems, Inc., Mountain View, California, 1988.

[11] B. Stroustrup, *The C++ Programming Language (2nd Ed.)*, Addison-Wesley, 1991.

APPENDIX

Here are simple versions of the extra methods needed to use the `employee` with OdeFS. We've ignored minor details such as checking for buffer overflow. OdeFS calls these functions from within a `trans` block, so that is not needed here.

```
// Return ofile representation for object.  Format:  Name=xxx, Salary=xxx
int employee::ofsRead(char* buff, int maxlen)
{
    sprintf(buff, "Name=%s, Salary=%d\n", name, salary);
    return strlen(buff);
}

// Create new object from ofile written by user.
persistent employee* employee::ofsCreate(const char* buff, int len)
{   int  xs;
    char xn[32];

    if (sscanf(buff, "Name=%s, Salary=%d", xn, &xs) != 2)
        { ofsErrMsg("Not in EMPLOYEE format"); return NULL; }
    if (xs <= 0)
        { ofsErrMsg("Salary must be >0"); return NULL; }
    return pnew employee(xn, xs);
}

// Update object from ofile written by user.
int employee::ofsWrite(const char* buff, int len)
{
    int xs;
    char xn[32];

    if (sscanf(buff, "Name=%s, Salary=%d", xn, &xs) == 2) {
        if (strcmp(xn, name) != 0)
            { ofsErrMsg("Cannot change employee name"); return 0; }
    } else if (sscanf(buff, "Salary=%d", &xs) != 1) {
        ofsErrMsg("Not in EMPLOYEE format"); return 0;
    }
    if (xs <= 0)
        { ofsErrMsg("Salary must be >0"); return 0; }
    salary = xs;
    return 1;
}

// Return default leaf name for object.
void employee::ofsName(char* buff, int maxlen)
{
    strcpy(buff, name);
}

// Given simple Name=XXX query, return object.
persistent employee* employee::ofsFind(const char* name)
{
    char xn[32];
    persistent employee* pe;

    if (sscanf(name, "Name=%s", xn) != 1)
        return NULL;
    for (pe in employee) suchthat (strcmp(pe->name, s) == 0)
        return pe;
    return NULL;
}
```