Summary of Summer 1995 NSERC Project - Building a Distributed Objectbase using DSVM

Ramon Lawrence
Department of Computer Science
University of Manitoba

May 29, 1998

1 Introduction

The project's goal was to build a prototype of a peristent, distributed shared virtual memory (DSVM) objectbase system. Although the prototype allowed objects to be stored persistently and accessed through DSVM, a SQL interface was not successfully created on top of it. It is move accurate to call the prototype a persistent DSVM object server, which will be appreviated OBS, as it supports simple queries such as creating or deleting objects but not complex ones as found in a SQL.

The project can be broken down into three main phases which occurred in chronological order. Each phase will be described with an overview and in terms of the system design, operation, and limitations.

2 Motivation

An OBMS using distributed shared memory offers several advantages over a traditional OBMS. Object sharing is transparent across machines as all machines see the objects in the global memory. This may limit server communication as there is no need for servers to poll each other to find objects. The biggest win will be in the area of interoperability. In order for two different OBMSs to communicate they must establish some sort of protocol in terms of object sharing, locking, and transmission. Obviously, this is a very difficult task without some predefined standards, as an implementor of an OBMS has little insight into what will be the design trends in the future. By abstracting away the message passing, sharing, and locking into a generic global interface the designer of an OBMS no longer has to worry if the design is compatible with current or future designs but only if it conforms to the global interface accessible by all OBMSs. By providing this global interface, the user interface to objectbase may change, but the performance and power will not.

3 Phase I - Basic Object Server Implementation

3.1 Overview

The first two months were dedicated to understanding the two major programs used to create the object server. The system uses the Exodus software [1, 2] to store objects persistently on various sites and uses the Treadmarks shared memory package [4] to guarantee uniform object access and synchronization across the sites using shared memory. The integration of these two packages results in an object manager supporting persistent distributed shared memory capable of object creation and manipulation.

Exodus and Treadmarks are C++ libraries which are linked into applications to provide the desired functionality. Treadmarks, having been designed for coding parallel algorithms on a network of workstations, allows the same program to be run across a number of machines, and more importantly, provides shared memory so that all processes can see a global address space. This is achieved by providing the C++ programmer with a method of allocating global shared memory and set of synchronization primitives (locks and barriers) to guarantee that this memory is consistent across the sites. Actually, each site contains its own copy of the shared data which is only guaranteed to be consistent after synchronization operations. Treadmarks uses lazy release consistency [5, 3] and multiple-writer protocols to implement this efficiently.

Exodus allows objects, which it treats as untyped streams of bytes, to be stored in a structure it calls a volume which is either a raw disk partition or a UNIX file. The system is a client-server architecture with the application (C++ executable) being the client and a seperately running process acting as the server. By making appropriate function calls in the application program, the programmer is able to store/restore objects and create/delete indices on volumes managed by a server. The client program need not be at the same site as the server as they are able to communicate over the network using IP addressing.

3.2 System Architecture

Treadmarks is used to run an object management application capable of reading and performing requests like creating, loading, and moving objects. Treadmarks creates an object server process at each designated site and provides a DSVM between them. This is shown in Figure 1. Part of each object server process (OBS process) is a textual user interface which allows the user to enter object manipulation commands and an Exodus client which allows the OBS process to communicate with Exodus servers to store objects persistently.

The DSVM between the processes is used to store a global object table with contains object descriptors for all known objects in the system. Since this table is in DSVM, it is accessible by all the processes and must be locked during operations to avoid conflicts. The object descriptor contains vital object information including its size, resident status, unique integer id (UID), storage location (OID), and pointer to the object if it is resident. The UID is defined by the user (or system in later versions) and is a 32-bit integer value associated with the object with is guaranteed to be unique across all objects. This uniqueness is guaranteed everytime an object is created by scanning the global object table. The OID is a structure defined by Exodus which it uses to retrieve a given object from a volume. The OID is unique across all volumes so it can be used to find any objects by itself.

An object may or may not be in shared memory even though it exists persistently. Initially, no objects are resident in shared memory. As the program runs, if one process requests an object, it is loaded from an Exodus server, placed in DSVM, and the table updated so that any future accesses by any other process will not entail a disk access. There is no corresponding swapping out of objects that have not been used recently.

To maintain consistency, the table is locked as an entire object. Locking at a finer granularity (table entry level) could be done but would not be substantially more efficient as most operations do a scan through the entire table (linear search) to find the correct object entry.

The object table must be stored persistently in case of failure. This is achieved by creating an index(B-tree) in each Exodus site that stores objects. The UID (key) and OID (data) are stored in the B-tree index at the same site at which the object is stored. The index at each site is preloaded by the parent process into the object table so that any object existence query can be verified in memory. By preloading the table into shared memory, all used object ids are visible to all processes. Without preloading, everytime an object is created all indices on the Exodus sites must be checked to see if this object exists somewhere already. By bringing in the object header, not the object itself, and storing it in the table, this checking need not be done. Since this is a startup feature of the system and will be done rarely, the run-time benefits of preloading far out weight the costs.

Shared memory, provided by Treadmarks, allows the global object table, as well as any objects allocated in shared memory, to be visible across all machines. Persistence is provided by linking in the Exodus library into the program which gives it the ability to use Exodus routines and call an Exodus server to store the objects. Exodus has a client-server architecture. The Exodus client (EC) is part of the OBS process in the initial implementation, although this is changed in later versions due to Exodus/Treadmarks conflicts.

The initial system, as defined, only supports objects as a stream of bytes whereas the object model interprets the stream of bytes as attributes and allows method execution on them. The initial version was only able to create and delete such "objects." To make the system more powerful, executable objects were added. By allowing executable objects like C++ executables to be run and stored within the system, one provides a method, although rather inefficient, of executing these methods. For example, each method could be a compiled C++ program, and running this program really executes the desired method.

The next major feature added was remote execution of commands. Initially, only executable objects could be invoked remotely, but later all object server commands could be shared amoung the OBS processes. By adding a global job table in DSVM and allowing each OBS process to add jobs to it, remote execution is possible. The job table consists of a queue of jobs for each OBS process. Any OBS process can decide that the given object operation is better suited to a given OBS process (likely at a different site) and so add that job to the other OBS process' job list which will execute the task. With the job tables being in shared memory, load balancing is possible.

Remote commands are handled by an array of linked lists in shared memory. Each process has a linked list of jobs (job queue) associated with it. Each process polls its input window (asynchronous I/O (1)) and then checks the job queue in a continuous loop. If a command is entered by the user it can be run on the local site, or any remote site, by giving the correct process id. (2) If the command is executed at a remote site, it is put in the job queue in shared memory for the given process, and the calling process has no further connection with the command. (ie. There is no associated reply if the command was a success/failure.) The command may be any object server command except, terminate the process (quit). Remote object execution is permitted with the caller responsible for the command line arguments argv and envp (present, but unused).

Finally, the system architecture defined in Figure 1 is more restrictive than need be. An OBS process may or may not be sharing the site with an Exodus server. If the site does not have an Exodus server, then all operations performed on objects by this process require transmission of the object from the storage site to the process site over the network. To reduce this overhead, it would be desirable if object operations could be processed by a process at the same site as where the object is stored. Each Exodus client has access to all Exodus servers giving rise to a fully connected network of client and servers. Thus, although it is not necessary for an Exodus server to share a site with an OBS process, it makes sense in terms of efficiency. This was one of the motivations for allowing a process to pass operations onto other processes. If a request to operate on an object on a remote site handled by a given process T is received by process S, it can pass that request onto process T which can execute it more efficiently. This results in a significant improvement for most operations, especially loads, as it doesn't matter what process puts the object into shared memory.

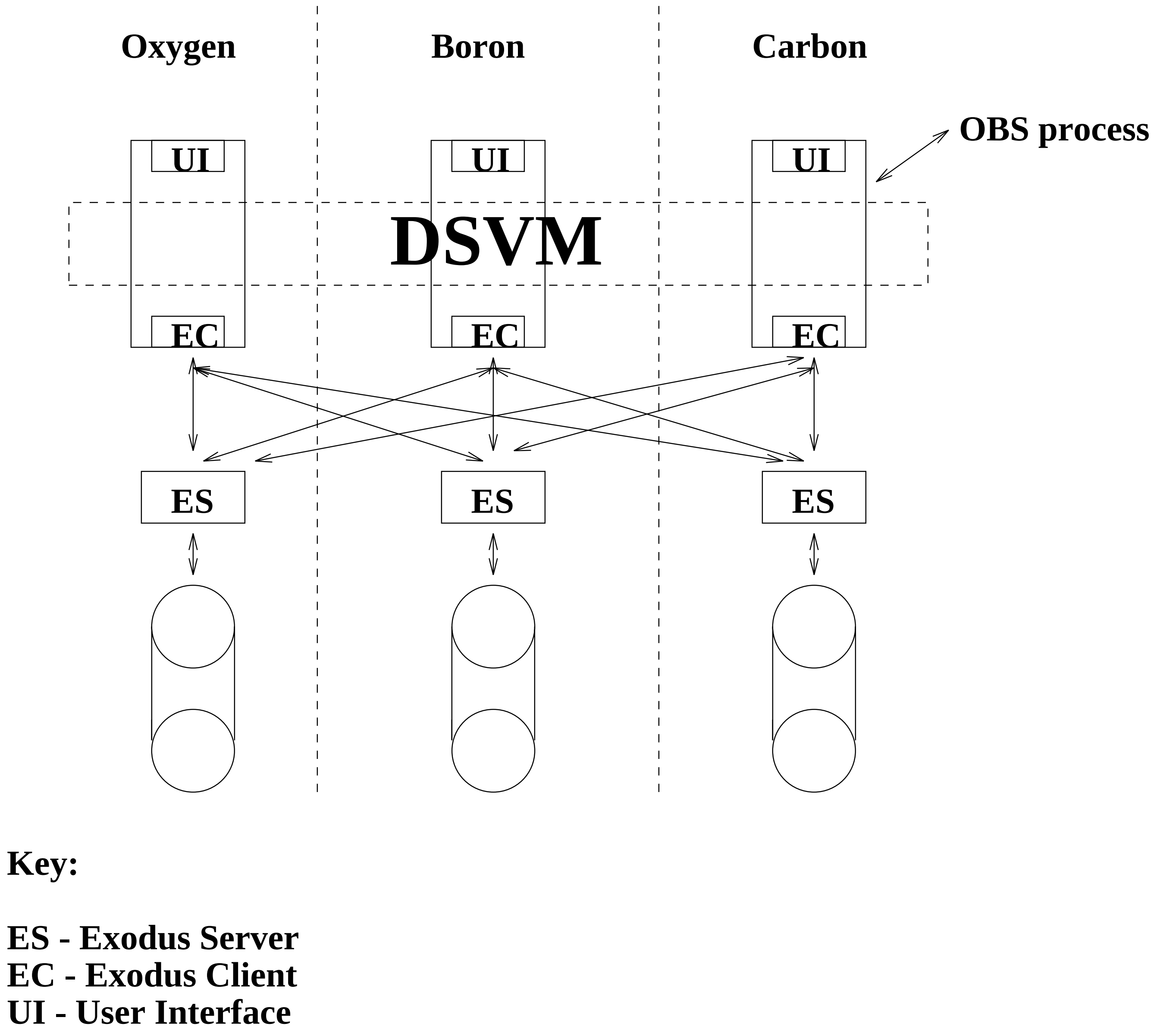


Figure 1: Diagram of phase I architecture

3.3 Operation Overview

All OBS processes are started from invokation of the parent process from the command line. This parent process is the object server (OBS) process which is then forked to the given machines by Treadmarks. The current setup is running on three machines called carbon, boron, oxygen. Each machine has an Exodus server configured under a directory in /data. Each Exodus server stores its objects in a UNIX file under /data, as each machine has a local disk. The size of each site is the same, consisting of 4 MB log space and 8 MB data space. Each server manages one volume. Each volume has a unique id across all sites. The volumes managed by carbon, boron, and oxygen servers are 4000/4001, 3000/3001, 2000/2001 respectively. Two numbers are actually provided as a data volume/log volume pair. Treadmarks is configured by listing the machine names in which the processes its forks will run. Each of the three machines carbon, boron, and oxygen should have an object server process so each are listed in the .Tmkrc file.

The OBS parent process is started. It parses the command line and forks the specified number of processes to the defined machines. It is responsible for determining all the volumes at the sites it manages, retrieving the object information from the indices in these volumes, and allocating the shared memory for the global object, job, and site tables. After the initial allocation, all processes are identical until termination. They present the same user-interface, access to objects, and all have mechanisms for handling conflicts. On termination, the parent process is responsible for any Treadmarks/Exodus specific cleanup.

A typical operation performed by the server is to create an object. Each server process polls its I/O window for any user commands. Let's say a user at boron issued a command to create an object. The object server process at boron would prompt for a unique object id, the data (a string), the site where the object will be stored persistently (oxygen,boron,carbon), and an object server process which will actually perform the operation. Let's say he chooses to store the object on carbon and have the process on oxygen do the work. When the user input is concluded, this data is copied to a command structure which contains all the above information. The command structure is then copied into the global job table for oxygen since it is performing the operation.

Meanwhile, oxygen has been busy polling its own input window and checking its job queue. It finds that a command has been issued from boron. The command is removed from the job queue and copied into a local structure. This local structure is resident on the stack for oxygen and has a size of approximately 100,000 bytes. The object server on oxygen interprets the command as a request to create an object and calls the appropriate function. This function has 5 tasks: check if the object exists, store the object on carbon, insert the unique object id in carbon's index, make the object resident in shared memory, and update the global control table.

Two of the tasks require calling the Exodus server which is called by the Exodus client imbedded in the OBS process. In this case, since oxygen is performing the operation, it is the EC in the OBS process on oxygen which is calling the Exodus server on carbon (location to store the object). Having the EC imbedded in the OBS process lead to problems which were corrected in phase II. With the EC embedded in the OBS process, a call to the Exodus server is just C++ function calls in the OBS code.

In our example, creating an object, two requests must be made of the Exodus server: store the data and insert the unique object id into the index. First the data is stored by performing an Exodus request as described above. The return value is a structure called OID which is used by Exodus to retrieve the object. If a program has an object's OID, it can retrieve the object at any time. Another Exodus request is made to insert into the index a key/data pair of the unique object id/OID. When these two requests are completed the object exists persistently in the objectbase. The object server then allocates space for the object in global shared memory and copies the data there. Finally, the global control table is updated by inserting the new object and updating the pointer to it in shared memory. During this entire time, the whole global control table was locked. This is not very efficient, but it addresses a race condition. In between the time an object server process checks to see if an object exists and the time it has a usable OID to store in the global control table, there is time for another process to create an object with the same unique id, especially as the two store operations are very slow.

3.4 Implementation Limitations

Treadmarks uses signals to synchronize DSVM when accessed by the processes. This is fine for most C++ applications, but with the EC embedded in the OBS process it also received these signals passed around by Treadmarks. The Exodus client performs many system calls, including writes, which are interrupted by signals and cannot be restarted. This lead to erratic behavior of the EC/ES during large object stores which were often preempted. There was no simple solution to this problem and lead to the major design changes of phase II. See appendices for more information.

4 Phase II - Solving Object Server Limitations

4.1 Overview

After phase I was complete, a persistent, DSVM object server was created. Unfortunately, it was limited in scope, lacking in power, and unreliable for large objects. To address this, it was decided to extract the Exodus client from the OBS processes which interfered with the normal Exodus client/server communication. This resulted in a major design change and an excursion into the IPC basics of shared memory and semaphores. This lead to two new C++ "units" which provide greatly simplified access to semaphores and shared memory which may be used in other projects. The OBS process now communicated with a seperate Exodus client process (one for each OBS process) through local shared memory (LSM) to handle its requests to the ES. Since the EC was seperate from the OBS process, Treadmarks signals no longer effected it.

The second diversion at this point in the project was to create a persistent programming language. By using the object server architecture as defined and some intelligent copying a C++ persistent programming language was created.

4.2 System Architecture

The bulk of the original implementation was still present in the phase II version with the exception of the LSM segment separating the Exodus client and OBS process. The system at this point was still on the three DEC stations.

The Exodus client is now removed from the Treadmarks application. Each process spawned by Treadmarks has an associated Exodus client which handles its requests to the Exodus server. The Exodus clients are started separately from the Treadmarks processes and run in the background. The Exodus client that was embedded in the OBS process is replaced with an stub client (SC) which passes requests and receives data from the Exodus client through LSM. A Treadmarks process communicates with its associated Exodus client through local shared memory (LSM) using a predefined command format and protocol. The Exodus client translates this command to an appropriate format for Exodus and invokes the Exodus server. Any data retrieved from the server is passed back to the Treadmarks process by the stub server through LSM. The Exodus client here is different than the Exodus client in phase I which was just function calls to the Exodus server as defined by Exodus. The Exodus client here is a seperate process and really is an interpreter for the stub client. It reads the LSM and then converts the commands within into appropriate function calls. The function calls are the same as phase I, but now the "Exodus client process" is responsible for determining what calls to make whereas before they were coded into the OBS process code. The distinction is this: the functions associated with the function calls are in the Exodus code, while this new Exodus client is a user-defined process used to select the apprropriate function calls and then perform them.

The system is much like remote procedure calls (RPC). In a RPC, there is as stub process call in the caller which invokes the OS to execute a remote command. Here the stub is in the OBS process which when it requires the Exodus server invokes the stub client to create an command. Instead of this command being passed over the network like in RPC, it is passed through shared memory. The receiver on the other end, in this case the Exodus client process, interprets the command and executes it. In this case the command is to be a client of a given Exodus server.

4.3 Operation Overview

The operation of the system is the same as described in phase I except for Exodus server calls. The term stub server and Exodus client are used interchangable in the following description although stub server is probably less confusing. Exodus client can be considered the C++ function calls to Exodus. The stub server should be thought of is the translator that translates the commands passed to it through LSM from the OBS process into the associated function calls (Exodus client operations) to Exodus servers.

Tasks which require calling the Exodus server will be described generically for any operation. Each object server process has a local shared memory (LSM) segment associated with it which it uses to communicate with a stub server serving as the Exodus client. There is a stub server on every machine that an object server process is running. There is only one shared memory segment and only the object server process and its associated stub server can use it. The shared memory segment is semaphore locked. To request an operation of the Exodus server, the object server much obtain the semaphore lock to the LSM segment. It then copies the command structure into the LSM segment and clears a done flag. The object server process "blocks" in that it waits for the operation to be completed, but it is really checking the done flag every second to determine if the stub server has completed its request. The command structure basically contains the arguments which the stub server needs to invoke the appropriate function call to the Exodus server. The whole operation is like a remote procedure call using shared memory. The LSM segment serves as a way of passing parameters and return values back and forth. Remember that this fix would not be needed if not for Treadmarks signals interrupting Exodus system calls. Meanwhile, the stub server is constantly polling the LSM segment to see if a request has been made. It finds the request, extracts the parameters from the shared memory segment and calls the Exodus server which performs the operation. Any return values are put into the LSM segment, which has been locked all this time, and sets the done flag.

The object server notices the done flag is set, extracts the parameters, and continues on.

4.4 Persistent C++ Programming Language

At this point, object server version 9, the object manipulation commands were still coming from user input through the textual user interface. As an exercise for phase III, the departure was made to change the object server interface into something more useful. The user interface was eliminated and replaced with another LSM segment (see Figure 3) in which object server commands would be placed. Thus, the old user interface (UI) of the older versions was replaced with a command interface (CI) in the newer ones.

The question arises here and especially in the phase III architecture: why do you need the LSM interface? It seems like it would be more desirable to have any clients of the object server have direct access to the DSVM. The major reason is the limiting usefulness of Treadmarks. Remember, that Treadmarks provides DSVM for multiple processes running the same program at different sites. This effectively means that the C++ application would have to be compiled with Treadmarks and the object server code. This is poor programming practice and hard to implement. Secondly, only ONE application can be run at a time, and the server would need to be restarted every time the application quits. Obviously, it is desirable if the object server has an identity of its own which was not dependent on the applications which are using it.

Thus, it was decided to communicate commands through LSM. The client application would link in a C++ header file which handles all the server requests for it, much like the Exodus and Treadmarks applications. Thus, a C++ programmer can make his object persistently simply by a few function calls. How the object server implements these function calls are none of his concern.

By adding these header files, at key sections of the code, the C++ application programmer could call object methods to make the object persistent and other methods to reload the object stored persistently. This is done without any C++ preprocessing or really fancy coding techniques. The trick relies on the fact the C++ stores its objects as a continuous stream of bytes. By determing the object size and its starting address (both easily accomplished in C++), the object can be treated as a untyped stream of bytes and saved as such. When it is loaded in, the bytes are retrieved from disk and copied over the original bytes that C++ allocated for the object. It turns out that the methods are irrelevant. Once you have changed the data for an object in memory in C++ any method innvocations performed on the new data are the same as if it was the original data. This is because the data and methods are stored differently. By using the fact that every object created has access to the methods of its class, all that must be saved/restored is the attributes of the object. In effect, the attributes are store persistently and the methods are provided by C++.

4.5 Implementation Limitations

Although not a limitation in the object server, the fact that the entire system was built on DEC stations proved a problem in phase III. The SQL-interface added is Mood. Mood uses dynamic linking for method execution, a feature not provided by Ultrix. Thus, the decision was made to move the system to SPARC workstations running Sun OS.

5 Phase III - Adding an Interface

5.1 Overview

The first few months yielded a relatively powerful object server which could provide the basic functions for a SQL interface to be built on top of it. The search went out for a OBMS which could be modified in such a way to keep its existing SQL interface and remove its storage component. Mood was chosen becuase it was based on the Exodus storage manager and seemed to present the least difficulty to change. Since Mood only ran on Sun OS, the project was moved to two SPARC stations: copper and calcium.

Compilation of Mood and Exodus was a nontrivial task involving compiler version problems and poorly segmented code. Two weeks were spent just on original compilation. The editing of Mood proved harder than anticipated and required the object server to be more powerful. The concept of a group of objects was added which could be linearly scanned.

The conversion of Mood's storage system from Exodus dependent to OBS dependent was done in stages. Unfortunately, many modules were interdependent and had to be converted simultaneously. This resulted in two versions. One used the OBS only as a database directory and Exodus for all other functions. The second, not fully debugged due to time constraints, used the OBS for most functions but was exceedingly slow.

5.2 Mood Overview

The Mood release contains two major components. The first is a generic SQL-type language called moodsql. The second is a graphical user interface for browsing, creating, and editing classes. Mood uses Exodus to manage nearly all of its important tasks. Unfortunately, a clear Exodus interface to the Mood source was never provided. Thus, the source code is a tangled mess of direct server calls, rerouted server calls, and complicated server transactions. The classes are poorly coded and the lack of indentation and comments is evidence of substandard coding techniques. Thus, the full-scale integration of Mood into a shared memory environment is difficult, bordering on impossible.

5.3 What works...

Moodsql appears to work under very basic testing like creating a class and simple select statements. The initial version has very limited interaction with the global object server using shared memory, but enough to provide database sharing.

The OBM dependent version is very slow and has only been "proven" to work for the SQL commands create a database and create a type.

5.4 What doesn't work...

Support for the graphical user interface (GUI) has been abandoned in more recent versions of the integration. The interface required special functions in order to run, so due to time constraints those functions were simply deleted. This decision was made after failed compilation of the GUI, as we are missing a necessary X library.

A compiler error involving g++2.6.3 (our current version) causes Exodus not to compiler properly. One system call involving regular expressions does not work properly when compiled under this version of g++. The code compiles normally but causes a segmentation fault while executing the system code. The parameters passed to it appear correct. Exodus, and consequently Mood, recommends compilation under g++2.4.5, a version which we do not have. Other older versions than g++2.6.3 don't work.

The same error appeared when g++2.6.3 was installed on the DECs. Consequently, we no longer have a working version of Exodus on the DECs anymore. I believe that g++2.6.3 must be changing some system header file.

Regardless, the error only effects one Exodus function, so you can get around it. Unfortunately, it is the option that reads configuration files. That means that in some places machine names are hard coded and multiple command-line parameters need to be passed. Incidently, the binaries released with Mood don't have this problem.

5.5 System Overview

The storage section of the system hasn't changed from the last version. There are still Exodus storage sites which are manipulated by Exodus clients who get their commands through local shared memory (LSM) from a particular object server process. Each object server process has access to the global shared memory provided by Treadmarks.

The interface section of the object server is new however. Using a similiar interface as os10 (the object server version which supported a persistant programming language), os11 (the latest version) communicates with a client through LSM. The client is a C++ header file which is linked into the application. Obviously, in order for the application to call the object server there must be an object server process on the same machine as the application.

Object server v11 has also extended the power of the object store system. In earlier versions, only simply object operations were provided like create, delete, and load. These operations proved inadequate for a sophisicated client like Mood. A major concept added was the idea of a group of objects. Every object now must have a group, distinguished by a character name, to which it belongs. A group can be locked and linearly searched using getfirst and getnext primitives to scan through its objects. To create, load, save, or delete an object, the client must specify a group name (character string) and the unique integer id. Locking can be done at the object level, but some conflicts with group locking still must be resolved.

5.6 Operation Overview

The first integrated version of Mood only uses the object server to store database definitions. The objects themselves are still stored on Exodus servers managed by Mood itself. More recent versions are attempting to have the object server provide more of this functionallity.

In the background, the Exodus servers, Exodus clients, and the object server are started. Then, moodsql is started. At this point, the user may create a new database, delete a database, or use an existing database. The database definitions are stored by the object server.

When a database is created in moodsql, it creates a database record which contains the file id's and object id's of critical database objects. For example, a database has a file for functions and a file for attributes. These files are managed by the local Exodus server, but their descriptors are stored in the database record. With these descriptors, an Exodus client may access information on an Exodus server regardless of machine location. Thus, by storing this database record and sharing it using the object server, you are effectively doing object sharing even though the objects themselves are not within global shared memory(GSM). Obviously, it would be desirable if the sharing was done through the GSM of the object server and not through Exodus networking protocols.

Another "cheat" is used to make this database sharing happen. When Mood creates a database and its classes, it stores their definitions in a file. Mood processes running on different machines must all see these same definitions, so I made the definitions directory in my home directory thus taking advantage of the NFS and allowing the sharing to take place. On limited testing, there has been no indication of synchronization conflicts for access to these files.

After the user creates a new database or uses an existing one, Mood operates like it did before. Since only the database lookup mechanism is changed, once the database is found and properly loaded, all regular Mood functions are available as from then on Mood communicates with the Exodus server (maybe nonlocal) which stores the actual contents of the database. Newer versions of the Mood integration are working to reduce this dependency on Exodus, but the widespread integration of Exodus calls in the Mood source code makes this a difficult task. Note that the locking is being done at the database level.

6 Future Work

There are several possibilies for future work, the most probable being adding the SQL interface as intended. This could be achieved by continued work on Mood or designing a new OBMS. Since this project was just a proof of concept and not intended for distribution, it would probably make sense just to take the lessons learned in the construction and apply them to the design of a real persistent, DSVM OBMS.

References

[1] M. Carey, D.J. DeWitt, D. Frank, G. Graefe, M. Muralikrishna, J.E. Richardson, and E.J. Shekita. The

Architecture of the EXODUS Extensible DBMS. In M. Stonebraker, editor, *Readings in Database Systems*, pages 488–501. Morgan Kaufmann Publishers, 1988.

[2] M. Carey, D.J. DeWitt, and S.L. Vandenberg. A Data Model and Query Language for EXODUS. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, pages 413–423, September 1988.

[3] Sandhya Dwarkadas, Pete Keleher, Alan L. Cox, and Willy Zwaenepoel. Evaluation of release consistent software distributed shared memory on emerging network technology. Technical report, Rice University, 1993. ftp://cs.rice.edu/public/munin/lazy-sigarch93.ps.Z.

[4] Pete Keleher, Alan L. Cox, Sandhya Dwarkada, and Willy Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. Technical report, Rice University, 1994.

[5] Pete Keleher, Alan L. Cox, and Willy Zwaenepoel. Lazy release consistency for software distributed shared memory. Technical report, Rice University, March 1992. ftp://cs.rice.edu/public/munin/lazy-sigarch92.ps.Z.

A Phase I Notes

1. Version 8 of the object manager requires at least 2 processes (windows) be active to use remote execution. Although a process is allowed to put jobs in its own job queue, Treadmarks processes do not do asynchronous I/O when only one process is active. Thus, the one process blocks for user input, and the checking for remote commands is never done.

2. Each process has a process id in the range 0 (the parent) to the number of active processes. The process ids are not bound to any machine name within the program. It is the user's responsiblity to know at what site a process corresponding to a given process id is executing. This can easily be done as the process ids are the same as the line#-1 in the .Tmkrc configuration file. eg. If the 3rd line of

the file was boron.cs.umanitoba.ca and at least 3 processes were running, then process 2 would be executing at boron.

3. Treadmarks uses signal 11 (SIGSEGV) to trap access to (global) shared memory. It also uses signals to control the asynchronous reads from the terminals associated with the forked processes. Since in the original implementation, the Exodus client and the Treadmarks process were the same process, the Exodus client would receive these signals. If they occurred during a system call (like writev), it would be interrupted causing storage level problems.

   Blocking these signals results in proper execution of the Exodus client, but then the shared memory of the Treadmarks processes is no longer consistent. Thus, system calls, especially writes, should be avoided by an application running the Treadmarks package.

   The process causing the signals is not clear. The problem has never occurred when only 1 Treadmarks process is running, so it may be the other processes which are causing the signals and not the one attempting the system call at that moment.

4. Problems arise when a Treadmarks application tries to fork processes using the fork() system call. Barriers no longer function properly for the original Treadmarks processes even if the forked process immediately calls execve(). Thus, this requires the stub server to be started separately.

B Phase II Notes

1. A generic local shared memory interface has been implemented to simplify shared memory communication. Each shared memory segment has an associated semaphore for locking, and the locking mechanism is greatly simplified. Shared memory segments are only removed when all processes detach the segment making coordination of segment destruction much simpler.

C Phase III notes

1. By adding groups to the object server, the format of the global object descriptor table changed. Each entry in the table is a group name followd by a linked list of object descriptors of objects that are in the group. Before each table entry was just an object descriptor. Thus, the group of an object is needed to find it, although the unique object id (UID) is still unique across all objects.

   A second change occurs in how the objects are stored. First, group descriptors are stored in a group file (one on each site) which is preloaded by the parent OBS process into the global table. The object descriptors of each group are stored in their own file which can be found using the group descriptor. All object descriptors of all objects at all sites are preloaded like before. Originally, object descriptors were read from an index at each site. Theses indices are still present and still store UID/OID (key/data) pairs but are no longer used. All the objects at a site are stored in one object file (one for each site) as they can be pulled out without scanning the entire file by using the OID in the object descriptor.

2. Directories of Mood conversions/features. All directories are on copper under /data.

   The Mood original compiled from source, with the before-mentioned error, is in the directory Mood. Only Moodsql compiles successfully, as a library is missing for the GUI.

   The first attempted conversion is in the directory mood_working. This is the version that uses the object server just as a database directory lookup facility and uses a NFS and Exodus networking procedures to perform the object sharing. It should be working for all SQL functions.

   The second attempted conversion is in the directory moodv2. Besides using the object server as a database directory the database sequencing object is controlled by the object server now. This results in a fairly major slowdown due to the significant performance advantage of Exodus over the object server. Also, a lot of redundant, commented out code, and unused Exodus/object calls have been eliminated from this version resulting in a lot smaller source code. All SQL function should work.

This is the version where someone trying to continue the Mood conversion should use.

The latest attempted conversion is in the directory test. It is only known to work for two simple SQL commands. It was only intended to change the linked list implementation from Exodus to OBS dependent but dependencies in the code resulted in changing 80that could never be adequatedly debugged. This version could probably be thown away if space is required in /data.

D Object store versions

This is a quick description of the object store versions and the features provided by them.

D.1 obj_store.cc (version 1)

The objects are strings identified by a unique integer id (UID). Object descriptors are stored in DSVM in a global table with key UID and data OID. The OID is Exodus dependent and allows Exodus to find the object on disk. The global table is stored persistently in a UNIX file and is preloaded into DSVM when the server starts up. The objects are stored all in the same file in Exodus and only loaded into DSVM on their first use (resident afterwards). Only the operations create and load are supported through a textual user interface. Finally, Treadmarks calls accessing DSVM are mixed with Exodus calls. (ie. no logical code seperation)

D.2 obj_store2.cc

The global object table is moved from a UNIX file into Exodus. Essentially each object has a descriptor record stored in an Exodus file containing its UID and OID. The file is preloaded into DSVM table when object server (OBS) starts.

D.3 obj_store3.cc

Instead of storing the object descriptor table in a file, it is stored in an index with the key being the UID and data the OID. Note that no performance benefit yet as whole index is preloaded when OBS starts. Actually, probably worse at this point (no delete operation) and insert into index slower than unordered file.

D.4 obj_store4.cc

The global table now has locking associated with its access which implies earlier version had a race condition in terms of table access. Unknown locking flaw still present as after find searches the table for a UID, say to create an object, then releases its lock on the table, before create finishes and inserts the new UID into the table, another OBS process could create object with same UID.

D.5 obj_store5.cc

Up to this point, all OBS processes regardless of machine location, accessed the same Exodus site. When multiple Exodus sites were added, site definitions were stored in a global table in DSVM with information retrieved from a UNIX file sites.dat. This allowed each OBS process acess to multiple Exodus volumes at (perhaps) different sites. Thus, objects may be stored at different sites. This was the first version shown.

D.6 obj_store6.cc

A major code reorganization split the DSVM table manipulation, Exodus calls, and user interface into seperate header files obj_tr.h, obj_ex.h, and obj_store6.cc respectively. The new operations move object, delete object, and create/load/delete executable object added. Problems with large objects show up.

D.7 obj_store7.cc

Idea of command or job table added. The job table is allocated in DSVM and each OBS process can poll it as well as keyboard commands. Only remote command support at this point is remote execution of executable objects. ie. One OBS process can get the user input for the object execution, put the command in another process' job queue, and let that process execute it. This requires at least 2 OBS processes to run from now on.

D.8 obj_store8.cc

User input is totally extracted from command execution allowing all commands to be executed remotely. Large objects become a major problem as Treadmarks signals interrupt Exodus system calls.

D.9 obj_store9.cc

Finally, locking race condition discovered. Solution was to lock entire DSVM object descriptor table throughout any operation involving find then write to table (create,delete). Exodus system call interruption discovered so new Exodus client created and commands passed to it through LSM. Last version shown and made on the DECs. This would be the version corresponding to phase II of the report.

D.10 obj_store10.cc

The persistent programming language version. Although it is grouped in phase II, it has a lot in common with the phase III version. The textual user interface for entering commands is removed and replace with a command interpreter in the OBS process and a OBS client header file which is to be linked into the application that generates the commands. The UID is no longer determined by the user but is generated randomly by the system and returned to the client. The OBS client is compiled into the application, and OBS commands are invoked by calling client procedures which interpret the commands, pass them through

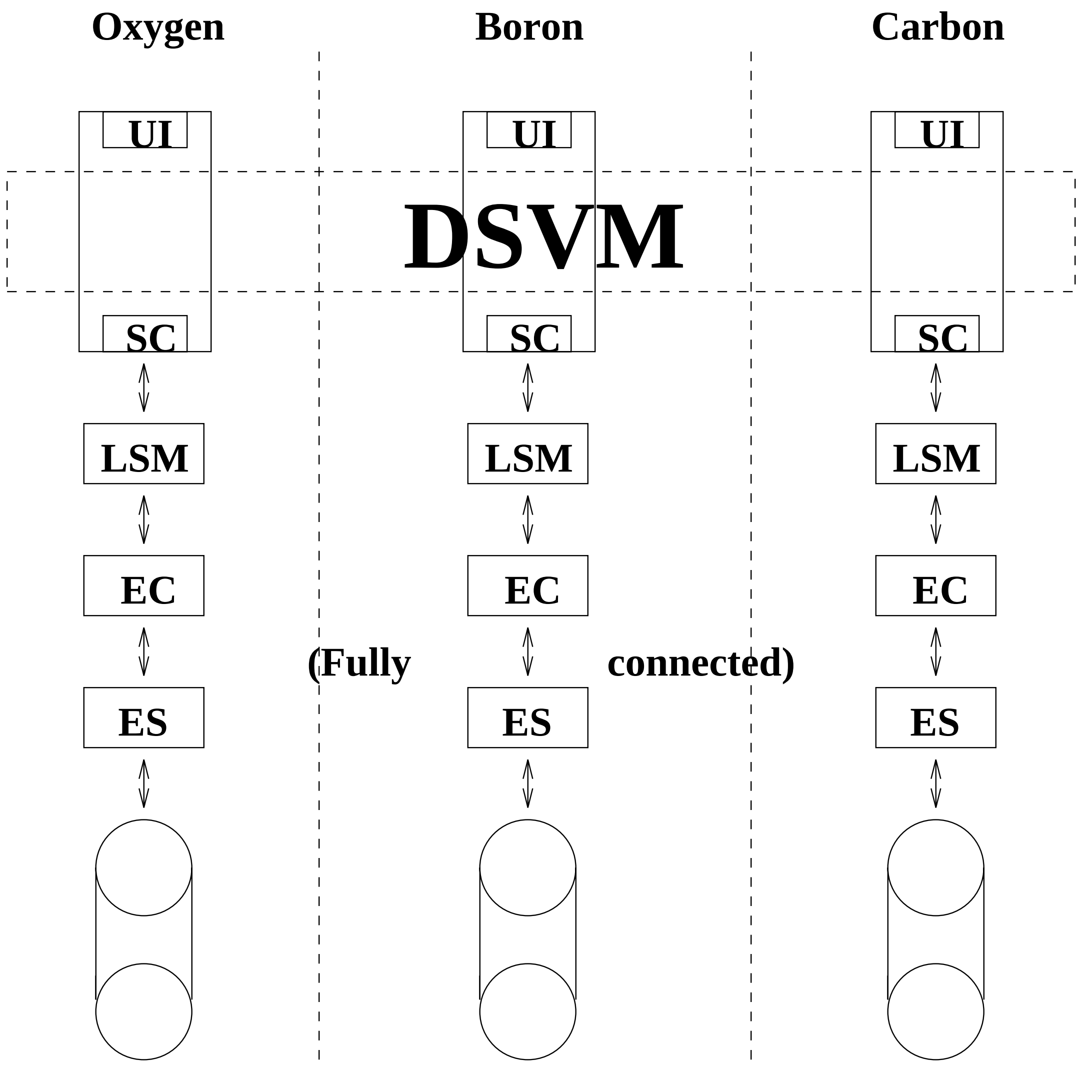
LSM to the OBS process which interprets and executes them. There are actually two LSM segments: one for incoming commands and the other returning the results. The two segments are associated with the OBS process on that given machine. Thus, all OBS clients on that machine use the same LSM segments so semaphore locking is performed. The semaphore only determines if the segment is locked or not, not whether it contains valid data. Thus, both the OBS server and OBS client still must poll the segment to determine if it can be used. The server polls the in segment looking for commands and the out sesgment to insure that it doesn't write over data not yet received. The client must insure that it doesn't write over some other process' commands in the in segment or that it gets the write data in the out segment. To minimize polling, client and server will sleep for a second if a given poll is unsuccessful. This results in the slow performance. Signals would be better, but given that Treadmarks, and consequently the OBS processes, already use signals for DSVM synchronziation, this may be hard to accomplish.

The persistent language support is easy provided by including a few header files in appropriate positions in the code. By including them in a class, it allows any object to store/load its attributes from disk by simple function calls. All that is really needed is the UID which is up to the application to remember. The methods are not stored but by using C++ scoping rules are also available to the object.

D.11 obj_store11.cc

This is the version used with Mood. The command operation is exactly the same as version 10 in that all commands come through LSM from the object server client linked into the application at compile time, in this case Mood.

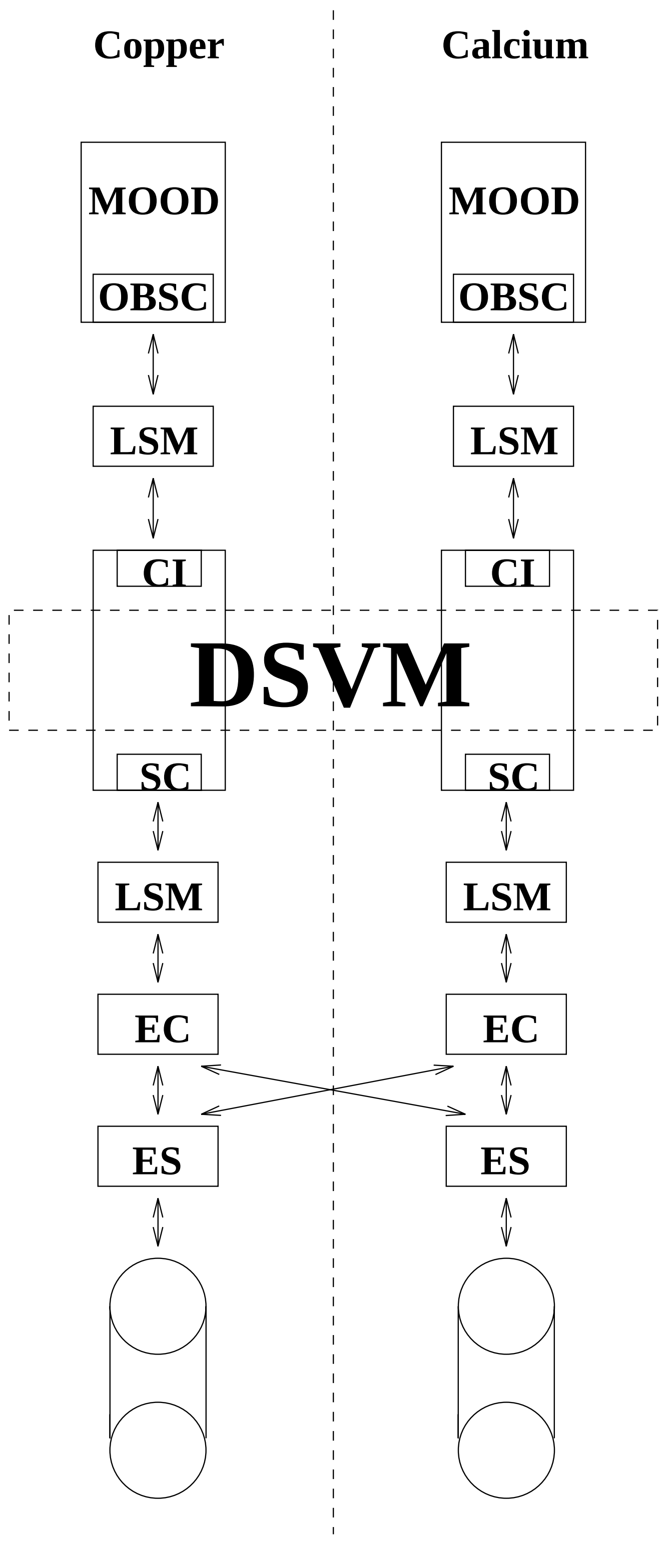
Mood performed a lot of file scans when it used Exodus, a feature that wasn't provided by the object server. So groups were added to the server which served the same role as files and could be linearly scanned. This file scanning of Mood proved bothersome as the inherent inefficiency of the object server made these file scans incredible inefficient when performed the multiple times that the Mood client perfomed them.



Key:

ES - Exodus Server
EC - Exodus Client
UI - User Interface
SC - Stub Client
LSM - Local Shared Memory

Figure 2: Diagram of phase II architecture



Key:

ES - Exodus Server	CI - Command Interface
EC - Exodus Client	OBSC - Object Server Client
UI - User Interface	LSM - Local Shared Memory
SC - Stub Client	