A PROOF-OF-CONCEPT IMPLEMENTATION OF PERSISTENCE IN A HIERARCHICAL STORAGE SYSTEM

Robert Grossman and Xiao Qin
Laboratory for Advanced Computing
University of Illinois at Chicago
Chicago, Illinois

Dave Lifka
High Energy Physics Division
Argonne National Laboratory
Argonne, Illinois

This is a draft of the paper R. L. Grossman, D. Lifka, and X. Qin, A Proof-of-Concept Implementation Interfacing an Object Manager to a Hierarchical Storage System, Twelfth IEEE Symposium on Mass Storage Systems, IEEE Press, Los Alamites, 1993, pp. 209-214.

ABSTRACT

The concept of providing transparent access to a collection of *files* in a mass storage system is a familar one. The goal of this project was to investigate the feasibility of providing similar access to a collection of persistent, complex *objects*.

We describe an architecture for interfacing a persistent store of complex objects to a hierarchical storage system. Persistent object stores support the uniform creation, storage, and access of complex objects, regardless of their lifetimes. In other words, a mechanism is provided so that persistent objects outlive the processes which create them and can be accessed in a uniform manner by other processes.

We validated this architecture by implementing a proof-of-concept system and testing the system on two stores of data. These tests indicate

that this architecture supports the creation, storage and access of very large persistent object stores.

INTRODUCTION

We describe an proof-of-concept implementation of a persistent object store for complex objects distributed across a hierarchical storage system. Persistent object stores support the uniform creation, storage, and access of complex objects, regardless of their lifetimes. In other words, a mechanism is provided so that persistent objects outlive the processes which create them and can be accessed in a uniform manner by other processes.

The concept of providing transparent access to a collection of *files* in a distributed environment is a familar one. The idea is to be reference files with name and location transparency throughout a hierarchy of storage media, with warm files being cached and cool files being migrated [5]. The goal of this project was to investigate the feasibility of providing similar access to a collection of persistent, complex *objects*.

The motivation for this simple. For concreteness, consider the analysis of high energy physics data. The ease of use gained referencing files with name and location transparency is well established [9]. These files typically represent data sets used by working groups. On the other hand, physicists are not interested in the files per se, but rather, in the events contained in the files. *Events* are objects recorded by the detector representing particle collisions or putative collisions of possible interest. For example, one could query for all events containing two leptons with equal and opposite charge, whose energy is between specified bounds. This is the approach taken in [1], [3] and [2]. This approach requires that objects (in this case the events) be referenced with name and location transparency.

This is in contrast to the organization of a typical object oriented database. For example, one typically establishes a database by using a create command which specifies the size and location of the store. It is then up to the database adminstrator to juggle the databases among the available physical devices—this task becomes increasingly difficult as the number and size of the databases grow.

By an *object manager*, we mean a system which creates, stores and accesses objects in a persistent fashion. An object manager is the core of an object oriented database, but an object oriented database also provides additional features, such as transactions, back up and recovery, a data manipulation language, etc. Scientific and engineering applications typically produce large amounts of data which must be stored in an hierarchical stor-

age system. The analysis of the data is often greatly aided by using object oriented databases [2]. It is a problem of current interest to marry these two technologies. In this paper, we describe one approach—using an object manager designed to work in a hierarchical storage system. This provides a solution which works for data that is historical, that is data which is write once, read many. By layering other systems on top of this, it is possible to add functionality to work with non-historical data.

ARCHITECTURE

The system is divided into several modules: a Persistence Manager, a Persistent Volume Server, and a Persistent Volume Mover. The Persistence Manager implements persistence for complex objects. We use a modelin which persistent memory is divided into physical regions called *persistent volumes*, or simply *volumes*. If the Persistence Manager determines that the object requested is not in a volume currently loaded into persistent memory, it sends a request for the volume to the Persistent Volume Server. The Persistent Volume Server determines the bitfile containing the volume and sends a request for the bitfile to the Bitfile Server, part of the storage system. The bitfile is returned by the Bitfile Mover, also part of the storage system, to the Persistent Volume Mover, which extracts the volume from the bitfile, loads the persistent volume into virtual memory, and sends a reply to the Persistence Manager indicating that the volume is loaded. See Figure 1.

The Persistent Volume Server and Persistent Volume Mover should not be confused with the Physical Volume Repository, which is part of the Mass Storage System Reference Model [5].

The primary design consideration for the scientific applications we have in mind is performance. Our target applications contain very large amounts of experimental or simulated data, which is historical in the sense it is essentially write once, read only. For this reason, we have deliberately kept the architecture simple, and have not tried to turn it into a general purpose object oriented database [4]. For example, if transactions are required, they may be implemented by contructing appropriate layers over this architecture.

Persistence Manager

The Persistence Manager is responsible for creating, storing, and accessing complex persistent objects. Objects in a persistent object store each have a unique id, called a persistent id, or pid. A subset of the objects in the persistent store are in memory, or virtual memory, at any one time. The

Persistence Manager is also responsible for moving objects from memory to permanent storage as necessary so that objects may persist after the process which created them terminates and so that persistent objects may be accessed in essentially the same way as transient objects.

Persistent Volume Server

We assume that from the pid of an object it is possible to infer the volume which holds the object. If the Persistence Manager requests an object with a pid corresponding to a volume which is not available in (virtual) memory, it faults, and generates a request for the volume to the Persistent Volume Server. The Persistent Volume Server then determines the bitfile containing the persistent volume and sends a request to the Bitfile Server, which is part of the storage system, for the bitfile. The Persistent Volume Server also sends a message to the Persistent Volume Mover that the specified persistent volume is required.

Persistent Volume Mover

In response to a request to the Bitfile Server for a bitfile, the storage sytem responds by moving the bitfile from the Bitfile Mover, which is part of the storage system, to the Persistent Volume Mover. The Persistent Volume Mover extracts the volume from the bitfile, loads it into (virtual) memory, and sends a reply to the Persistenct Manager indicicating that the persistent volume has been loaded.

lthough the Persistent Volume Server and Persistent Volume Mover could be combined into a single module, we have found it useful to separate them. One reason is that a high speed data path may be available to move the volumes, while the request for volumes may come along another path. This reason also contributed to the decision by the Mass Storage System Reference Model to separate the Bitfile Server and the Bitfile Mover. Another reason is that we found we convenient to implement several different Persistent Volume Servers: all shared essentially the same Persistent Volume Mover.

VALIDATION STUDY

To validate this approach, we did a proof-of-concept prototype using a simplified version of the architecture. We had already implemented a Persistence Manager called PTool [7]. For this study, we implemented proof-of-concept versions of the Persistent Volume Manger and Persistent Volume Mover. We simulated hooks into a storage system complient with the Mass Storage System Reference Model [5] by using simple look-up tables which accessed bitfiles from disk and tape as appropriate.

Figure 1: An architecture for interfacing a persistent object store to mass storage system.

We tested the system using two stores: a store of trajectory segements arising in path planning problems [6] and a store of collider events from a high energy physics experiment [2]. We are currently completing benchmarks of these tests.

PTool

In this section, we follow [7]. PTool views the persistent object store as one large persistent memory: the persistent memory is divided into volumes. PTool assumes that one or more of these volumes are in virtual memory at a time. Associated to the volumes of persistent memory in virtual memory is a physical collection of disk blocks. The mapping between persistent memory and disk blocks is transparent to the PTool clients. Users do not explicity read or write to persistent memory, but rather simply indicate upon whether the object belongs to persistent or transient memory by using the standard (transient) allocation function (for example, "malloc" or "new") or a persistent allocation function (for example, "palloc" or an overloaded "new"). In both cases, accessing the persistent objects is the same as for regular (transient) data. See Figures 2 and 3. In other words, the protocal for allocating transient dynamic memory at run time or persistent memory at run time is essentially the same, but the persistent memory is available later by other processes. See [8] and [10] for a description of this approach to persistence.

In order for applications using PTool to access the persistent objects, the applications need a mechanism to iterate over collections of persistent objects. In other words, the application must access each persistent object in turn. For the proof-of-concept system, we used a simple iterator, as illustrated in Figure 3. For other applications, we have typically used a container class, such as a set or linked list. In either case, the iterator simply needs to access the entry point, or starting address, of each persistent object stored. This is done by accessing an auxillary data structure maintained by PTool.

We assume that the pid of an object is of the form (VolumeID, Location). The Location identifies the location of the object within the volume. Two different objects in different volumes may have the same Location number. The Location numbers are essentially virtual memory addresses, or, more accurately, together with an off-set, determine the virtual memory address.

VTool

We wrote a software tool called VTool to implement the Persistent Volume Server and Persistent Volume Mover, which we describe in this section. Each physical (persistent) volume corresponds to a fixed-size region of per-

sistent memory. A volume consists of four parts:

Header. The top portion is the header and contains identifier nformation about the volume.

Object Table. The second portion is the object table and contains the entry points for all the objects stored in the volume.

Object Area. The third portion is the object area and contains the persistent objects themselves.

Free Area. The fourth portion is the free area and contains available space for adding new objects to the volume.

We are currently exploring the effect of the size of the physical volumes upon the performance of the system. Volumes are divided into physical segments. The current implementation allows for only one volume to be in persistent memory at a time; future implementations will allow different segments from different volumes to be in persistent memory at the same time. Note that this design does not support objects that span across volumes. This again will be addressed in the next prototype.

FUTURE WORK

Our proof-of-concept system supports the transparent access of multiple persistent stores and persistent stores consistings of multiple volumes. An application simply interacts with the Persistence Manager; it is the task of the Persistent Volume Server to manage the necessary volumes.

As mentioned above, we have implemented only a bare minimum system. We are currently making some cosmetic changes: providing alternate iterators for objects using sets and other containing classes; and interfacing the system to a large scale storage system. We are also currently working on fundamental issues related to the system: developing caching and migration algorithms for collections of objects; extending the system by supporting large objects, or objects that extend over several volumes; and extending the system so that several volumes may be loaded into (virtual) memory at one time.

```
class Event {
public:
  int runNumber;
  int tapeNumber;
  int eventNumber;
  float vertex;
  Lepton *lepton1;
  Lepton *lepton2;
};

class Lepton {
public:
  float p[4];
  float charge;
};

main()
{
    int db = db_creat("psiEvents");

    Event *t1 = (Event*)
        palloc(db, sizeof(Event));
    t1->lepton1 = (Lepton*)
        palloc(db, sizeof(Lepton));
    t1->lepton2 = (Lepton*)
        palloc(db, sizeof(Lepton));

    t1->tapeNumber = 2984;
    t1->runNumber = 684;
    t1->eventNumber = 1849583;
    (t1->lepton1)->charge = 0.892;

    root_push(db, t1);

    db_close(db);
}
```

Figure 2: Using PTool to create a persistent Event, consisting of two persistent Leptons, together with several persistent integers and a persistent float.

```
int db = db_open("PsiSet");
root_iterator r;

Event *t1 = r.first();

...

Event *t1 = r.next();

...

db_close(db);
```

Figure 3: Iterating over the Events in the collection PsiSet.

CONCLUSION

In this note, we described an architecture for working with very large stores of persistent objects distributed over a hiearchical storage system. The architecture, in its present form, is suited for working with large amounts of historical data — data that is write once, read many. By layering other systems over this, it is possible to build the functionality required for working with data that is not historical.

The architecture is designed so that persistent, complex objects may be referenced with name and location transparency. The two main components of the architecture are a Persistence Manager and a Persistent Volume Server. We assume that the persistent store is divided into logical collections called persistent volumes. Applications requiring persistence interact with the Persistence Manager. Persistent volumes are transparent to the applications. The Persistence Manager handles persistence for volumes in the persistent store, approximately 256MB to 1GB in size. If the required volume is not loaded in (virtual) memory, a fault is generated and a request for the volume is passed to the Persistent Volume Server. The Persistent Volume Server then requests the bitfile containing the volume from the Bitfile Sever, part of the storage system.

As a final remark, note that there is nothing in this approach to preclude it from working with data stored using a relational model. We are currently investigating this.

We implemented a base line proof-of-concept system. From the results of this system, it looks like this approach is worth developing.

ACKNOWLDEDGEMENTS

This research was supported in part by NASA grant NAG2-513, DOE grant DE-FG02-92ER25133, and the Laboratory for Advanced Computing.

We are grateful to members of the PASS Project for contributing to this work.

References

[1] A. Baden and R. Grossman, "Database computing and high energy physics," *Computing in High-Energy Physics 1991,* edited by Y. Watase and F. Abe, Universal Academy Press, Inc., Tokyo, 1991, pp. 59–66.

[2] A. Baden, L. Cormell, C. T. Day, R. Grossman, P. Leibold, D. Lifka, D. Liu, S. Loken, E. Lusk, J. F. MacFarlane, E. May, U. Nixdorf, L.

E. Price, X. Qin, B. Scipioni, and T. Song, "Database Computing in HEP—Progress Report," *Computing in High Energy Physics 1992,* to appear.

[3] A. Baden, C. Day, R. Grossman, D. Lifka, E. Lusk, E. May, and L. Price, "Analyzing High Energy Physics Data Using Database Computing: Preliminary Report," *Laboratory for Advanced Computing Technical Report, Number LAC91-R17,* University of Illinois at Chicago, December, 1991.

[4] E. Bertino and L. Martino, "Object-oriented database management systems: Concepts and Issues," *Computer,* Volume 24-4, 1991, pp. 33–47.

[5] S. Coleman and S. Miller, editors, "Mass Storage System Reference Model: Version 4 (May, 1990)," to appear.

[6] R. L. Grossman, S. Mehta, X. Qin, "Path planning by querying persistent stores of trajectory segments," *Laboratory for Advanced Computing Technical Report Number 93-3,* University of Illinois at Chicago, 1993, to appear.

[7] R. Grossman and X. Qin, "PTool: A Software Tool for Working with Persistent Data", *Laboratory for Advanced Computing Technical Report Number 93-5,* University of Illinois at Chicago, 1993, to appear.

[8] E. Shekita and M. Zwilling, "Cricket: A mapped, persistent object store," in A. Dearle, G. M. Shaw, and S. B. Zdonik, *Implementing Persistent Object Bases: Principles and Practice,* Morgan Kaufmann, San Mateo, California, 1991, pp. 89–102.

[9] . D. Shiers, "Distributed storage management in high energy physics," *Eleventh IEEE Symposium on Mass Storage Systems,* IEEE Computer Society Press, Los Alamitos, California, 1991, pp. 109–112.

[10] I. Williams and M. Wolczko, "An object-based memory architecture," in A. Dearle, G. M. Shaw, and S. B. Zdonik, *Implementing Persistent Object Bases: Principles and Practice,* Morgan Kaufmann, San Mateo, California, 1991, pp. 89–102.