Palo Alto Research Center

Secure and Efficient Implementation of Abstract Data Types for Databases

Robert B. Hagmann

XEROX

Secure and Efficient Implementation of Abstract Data Types for Databases

Robert B. Hagmann
Xerox Palo Alto Research Center

CSL-91-5 June 1991 [P91-00092]

Abstract: One of the challenges for Object-Oriented Databases, Extensible Databases, and Third Generation Data Base systems is the secure and efficient implementation of Abstract Data Types (ADT). In the past, ADT's could either be implemented efficiently by binding the code into the address space of the database, which provided no security against such problems as wild stores or arbitrary branches. Or ADT's could be run in a separate process that provided good security, but poor efficiency. Security is used here in the sense of database integrity. Such activities as wild stores, looping, and arbitrary system calls are prevented, but not the full range of security (e.g, Trojan horses and compartments). The proposal in this paper is to combine the concepts of Lightweight Remote Procedure Call (LRPC) with limited address space sharing. Using LRPC, the combined call and return time between protected domains is 100-200 instructions, depending on the CPU architecture.

This paper concludes with a discussion of the implications of this design on Computer Architecture, Operating Systems, and Databases.

CR Categories and Subject Descriptors: D.4.2 [Memory Structures]: Design Styles — Virtual memory; D.4.2 [Operating Systems]: Storage Management — Virtual memory; H.2.4 [Database Management]: Systems — Query processing

Additional Keywords and Phrases: Abstract Data Types, Object-Oriented Databases, Third Generation Data Base system, Extensible Databases

XEROX

Xerox Corporation
Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, California 94304

1. Introduction

Performance and security: what could be more important to a database? This paper will show how databases can achieve good performance and good security when implementing Object – Oriented programming [Atki90], Abstract Data Types (ADT), Extensible Databases [Lind87], or a Third Generation Data Base system [Ston90]. The focus of the discussion in this paper is on Abstract Data Types (ADTs), since the abstract data type implementation is the key technology enabling the larger abstractions such as Object – Oriented programming.

All of these databases have a need for a good ADT/Object – Oriented implementation. A direction that is common to all of these systems is to allow user supplied code to run as part of the database. While the systems may differ in how to select what code to run for an operation (e.g., multiple inheritance), once the code has been selected it has to be run efficiently and securely.

Throughout this paper the term security is used. It is used in the sense that the system is not compromised by wild stores, arbitrary branches, looping forever, huge memory allocations, and other such problems. Security is used in the sense of maintaining integrity. The techniques here do not prevent Trojan Horses, statistical attack, or other means of acquiring data that should be protected.

Some kinds of user code has been run as part of the database for many years. SQL queries are usually run in the main database. Security was not a real problem because SQL is a restrictive programming language. Using SQL, it is not possible to crash the database, to put the database into an infinite loop, or to snoop in the buffer pool. Many systems wish to provide a more complete and hence more dangerous database programming language. Often this is an existing language such as C, C + +, or Ada.

An early ADT system for relational databases [Ong84] implemented ADT's written in C by binding their code into the address space of the database. This method was flawed since it had no security. Alternatively, ADT's could be run in a separate process that provided good security, but this method had poor efficiency. It was not possible to get both security and good performance.

Two ways of having both security and performance have been proposed. One is the IBM Enterprise Systems Architecture/370 "program call" [Scal89]. Another is the Protected Invocation of Psyche [Scot89, Scot90]. This paper proposes a third way to achieve similar security and performance, and, in addition, this proposal is closer to the structure of current operating systems (as opposed to Psyche's more radical approach) and without special processor instructions (as in ESA/370). However, all three methods share the basic idea of lots of address spaces, each holding an ADT implementation, and fast call/return between the address spaces. One of the purposes of this paper is to show the similarity of these concepts so that database systems can be constructed to use the similarities, and so that the databases can be simply ported between the various inter – domain invocation mechanism when changing machines and operating systems.

The proposal in this paper is to combine the concepts of Lightweight Remote Procedure Call (LRPC) [Bers89] with limited address space sharing. The basic outline for how this is done follows. The main database spawns process(es) for each protection domain (see below for a discussion of protection domains). Each process has restricted access to parts of the address space of the main database process. A process can read some of the data of the main database and it can execute the code for the methods. To invoke a method (or procedure), an inter – process procedure call is done (an LRPC) to a process in the proper protection domain. LRPC eliminates nearly all the overhead of a traditional local RPC call (see Section 3). The combined call and return time for a null call using LRPC between protected domains is 100 – 200 instructions (depending on the CPU architecture). By using limited address space sharing, much or all of the copy time can be eliminated. Hence, the null call/return time is the true overhead for doing a method invocation or procedure call. The method runs in the protected environment of the process at full speed, except that further message sends to different protection domains require additional LRPC's. The process is incapable of doing any damage to the main database since its memory protection prevents this. In many operating systems, it is also possible to control child processes so that they do not loop forever or issue unexpected system calls.

This proposal has three prime requirements for the operating system: flexible memory sharing, system call encapsulation and resource limits, and LRPC. Many operating systems have flexible memory sharing, encapsulation, and resource limits. However, LRPC is not available commercially. Hence, the proposal in this paper cannot be used immediately in commercial databases. Several research operating systems can or will have facilities that admit an efficient LRPC – like implementation [Bers89, Mass89, Scot89]. In addition, databases are a major customer of operating systems. While it is inaccurate to say that databases control the features provided by operating systems, databases do exhibit a strong influence. Since the technology is available, the customer is large, and the need is strong, operating systems will eventually provide LRPC – like implementations. The database community has to recognize the importance of these issues and translate this into pressure on the operating systems vendors.

In addition, a database that runs in the manner proposed here needs lots of address spaces. The Memory Management Unit and/or Translation Look aside Buffer (TLB), or their equivalents, must be constructed to allow efficient and low cost switching between lots of address spaces. In particular, there are too few user "contexts" of the SUN SparcStation – 1. TLB's and caches must be constructed that are not flushed on context switch.

2. Problem Motivation

In the previous section, it was shown that many database systems need ADTs. In this section, some of the problems with ADT implementation will be discussed. Database systems must have good performance, with strong security and excellent reliability.

One way to implement ADT's is to bind the object code into the database. For an unsafe language like C, this means that arbitrary code can be loaded into the database. Providing the database with source code and having it do the compile does not solve the problem: the code can still do arbitrary things. Nor does restricting the procedures that the code can call solve the problem. This code can snoop in memory, possibly issue arbitrary system calls or procedure calls (e.g., send electronic mail), rewrite itself, or insert a trojan horse. Security of the system is easy to compromise. Also, the coherence and reliability of the system are easy to destroy. This can be done maliciously or simply by having a bug.

Loading code destroys security for all users of the database. The common shared database processes are compromised. For databases that are private or shared among only one security domain, then security (as being defined in this paper) is not an issue. However in this case, no sharing is possible with other security domains, and coherence and reliability are still compromised.

Putting code into a separate process has been an inefficient way to fix these problems. A full heavyweight context switch with data transmission is necessary. This can easily take thousands of instructions. The cost of the context switch can dominate the cost of running the ADT code.

Another solution is to trust the ADT writers. This is a common in Extensible Database systems. This is undesirable since ideally any programmer or user should be able to build ADT's for systems that they work on or use. While some trusted and fast

ADT's can be provided in the system, performance and security for untrusted code is needed too.

3. LRPC

Lightweight Remote Procedure Call (LRPC) is described in [Bers89]. This technology is the key to this paper. To make this paper more self contained, a very brief description of it is provided here. The reader is encouraged to read the original paper for a more complete understanding of LRPC.

Remote Procedure Call (RPC) allows the programming primitive of procedure call to be performed remotely, with several constraints. This has proven to be one of the key building blocks of distributed systems programming. However, only a small fraction of RPC calls actually go to a different machine. For three systems measured in [Bers89], the highest percentage of RPC's that went remote was 5.3%. A RPC protocol that operates very efficiently in the local case is a clear optimization candidate.

The overheads in doing a local RPC are in stub overhead, message buffer overhead, access validation, message transfer, scheduling, context switch, and dispatch. The theoretical minimum round trip is a pair of traps and context switches. LRPC attacked these overheads. Arguments and results are exchanged in shared memory between the caller and callee threads. Virtual memory sharing is constructed at bind time, so the calls go fast. Using shared memory saves copying, allocation, deallocation, flow control, and concurrency control of arguments and results that are normally problems for the operating system kernel. The arguments are built into something that looks like an argument vector, so no manipulation on the callee side is needed, except for some validation checking. LRPC does a call to a particular thread. The general scheduler and dispatch of the operating system is avoided. Registers must be saved and the virtual memory adjusted, but the main code of the scheduler and dispatcher does not run.

LRPC nearly achieves its goal of a local RPC in the time of a pair of traps and context switchs. The theoretical minimum time is 109 microseconds on a C-VAX, while LRPC achieves 157 microseconds. The additional "overhead" goes mostly to validation, linkages, TLB flushes, and stack allocation.

4. Protection Domains

A protection domain is a set of access rights. Individuals are granted access to a protection domain. An example of a protection domain is "coding for all of the CAD

ADT's in the database". All CAD programmers would be granted access to this protection domain. This means that they can add, modify, and delete methods for the CAD ADT's.

The database has to model protection in some way. For example, SQL has the GRANT primitive. The protection domain may be explicit or implicit (as in SQL). The host operating system also must have a model of protection domains. These must mesh together.

5. Implementation

If security can be checked at compile time, then the best performance is obtained by loading the code into the database. This is true for SQL and other restricted programming languages. Here we are concerned with general purpose programming languages.

The way ADT's work is that only the ADT implementation is permitted to access the internals of an instance of the data type (e.g., to the fields or slots of the instance). This is a general property of ADT implementations, not just for databases. What it is important to understand is that the implementor of the abstract data type has full and complete knowledge of the internals of an object, while other abstract data types and other implementors have no knowledge whatsoever of the internals.

Thus, there is a protection domain associated with an implementor. Implementors would belong to several domains possibly consisting of multiple individuals. Note that protection is being enforced on the implementors, not on the clients! The implementors have full and complete knowledge of the internals of the objects anyway, so there is no way that the system could enforce protecting the objects from the implementors!

The feature of modern hardware and operating systems that enforces protection is the virtual memory system. The challenge is to configure the use of memory to conform to the protection and sharing needs of the database. An outline for how this works is presented below.

The main database process (MDP) has full read-write-execute access to its address space. For each protection domain needed at run-time, the MDP spawns a process (or processes) that has restricted access to the address space of the main database. While this process may be multi-threaded or multiple processes may be created per protection domain, for simplicity this discussion will assume that there is a single process per protection domain. The process for the protection domain has "execute" enabled for the virtual memory pages that corresponds to the code for its protection domain. A process may also read – only share the buffer pool and/or read – write share pages for a volatile object pool for the data types in the protection domain. See below for a further discussion of memory sharing.

So, what can an ADT process address? It can execute the code for any ADT in the protection domain. The system may either allow direct reference to the buffer pool, or it may convert between stable, disk based objects and volatile, memory based objects. With the buffer pool option, this assumes that all objects on a page are in the same protection domain and that the buffer pool is partitioned by protection domain. The pages a process can read correspond to the processes protection domain.

The process can also access a pool of volatile instances of objects from its corresponding protection domain. The pool is shared with the MDP. When objects are converted from a stable, disk based form into a volatile, memory resident form, the objects are placed in a volatile object pool that contains only objects from one protection domain by the MDP.

Many operating systems provide address space sharing. In Mach, the MDP would create a new task inheriting memory as appropriate. In UNIX System V, the MDP would use shared memory (e.g., using shmat, shmget, and shmctl) with appropriate protections on the shared memory segments. UNIX groups would be used as protection domains.

The UNIX Operating System has are the notion users and groups. The database system must be provided with a large number of user accounts for the use of the database. One account is the main database account. The rest of the accounts have nearly no privileges. Their "login shell" returns an error. They own no files and belong to no interesting groups. To share memory, a System V shared memory segment is created for each protection domain with the owner as one of the accounts. It is created in the group that is made up of only the main database account. The access bits are set to allow the proper access by both the main database account and the other account. Note that these can be different since one is using the owner bits and the other is using the group bits.

The process has limited rights. For example, it should not be able to spawn processes. All of its system calls should be trapped to the MDP (e.g., by using ptrace or /proc in the UNIX Operating System).

To do a method send (an object – oriented procedure call is called a send), the proper method is selected in the MDP by the database via its inheritance mechanism. If the send is to an untrusted protection domain, then a LRPC to a process in the proper protection domain is needed. If necessary, a process is constructed. LRPC is used to the spawned process. The process can execute the method code, since it has execute enabled for the code. The parameters, located in a shared read/write segment, are passed by reference. Parameters to objects that match the protection domain reside in the shared read – only part of the address space. Parameters to objects that do not match the protection domain are opaque: a reference through these pointers will give invalid address exceptions. Object Identifiers (OID's) and Tuple Identifiers (TID's) are mapped to buffer pool or volatile instance pool pointers when the parameters match the type of the ADT.

Update cannot be done directly, but has to be done via a reverse direction LRPC. Compiler support is probably needed to efficiently convert stores into LRPC's. Direct update probably is impossible due to crash recovery requirements: all writes have to be logged and the untrusted ADT's are not trusted to do proper logging. OID's, TID's, or pointers to objects not in the protection domain can be passed as part of the call. But these arguments are not directly used by the ADT code since the corresponding data is opaque and inaccessible.

Method sends to other methods inside the protection domain do not require the intervention of the MDP nor do they require a LRPC. To do a method send to a different protection domain from the process, a reverse direction LRPC is performed to the MDP. OID's and TID's are mapped for the new ADT. The send then proceeds as before. As an optimization, the return can bypass the MDP if the returned OID's and TID's (if any) don't have to be mapped.

The combined call and return time for a null call using LRPC between protected domains is 100 – 200 instructions (depending on the CPU architecture). By using limited address space sharing, much or all of the copy time can be eliminated. Hence, the null call/return time is the true overhead for doing a method invocation or procedure call. This is about an order of magnitude more efficient than previous database ADT proposals with security.

6. Different levels of security

The thing that is important to understand about ADT implementations is that the protection domains are built around the programmer, not the user. Recall that the implementor of the abstract data type has full and complete knowledge of the internals.

Processes are constructed to provide the virtual memory firewalls. What can fit in a single process? If it is just one type, then the number of processes needed would be exorbitant. Instead, protection domains corresponding to the protection domains used in the organization should be established. Provided that all methods can be considered safe from each other in a protection domain, then process(es) can be spawned for the entire protection domain. Hence the number of processes should be modest. Note that the processes should be created on demand with additional processes created for the same domain if necessary.

Trusted programmers still can write trusted code that runs in the MDP. The overhead to invoke these methods is nearly zero inside of the MDP.

There are three ways to pass arguments and do sharing. The first is the way LRPC works. LRPC constructs a shared segment between the caller and callee. Arguments are marshalled (copied) into this segment.

The second way to pass arguments is to read – only share the volatile object pool between the MDP and the processes. A set of pool pages are dedicated to a protection domain. A protection domain can have multiple pool pages, but the number of shared pages should change slowly since it will require remapping the address spaces of the processes. Volatile copies of objects are made in the pool pages. Each process only shares those pages with the proper protection domain. A method send then only has to send a pointer to the process for those arguments whose types match the ADT. All other types should be opaque and are passed by OID or TID. Note that copying is done to create the volatile object pool, but no copying is done during a send.

The third and final way to pass arguments is to pass them directly in the buffer pool. If the database can operate directly on data coming from the buffer pool, then this does no copying whatsoever. However, it may not be secure. ADT's are free to snoop in the buffer pool. If the database is constructed so that only objects of a given type or protection domain are kept on a secondary storage page, then the buffer pool can be sub – divided by protection domain. Threads can only access pages with instances of data types from their protection domain. If any object can be on any page, then the whole buffer pool has to be read – only shared. The processes cannot corrupt the buffer pool, but they can snoop on data outside of their protection domain.

The database may wish to protect itself against synthetic identifiers (OID's or TID's). By using encryption or check bits on identifiers, synthetic values cannot be created easily. Hence, it is safe to pass identifiers to methods of ADT's from different protection domains.

7. Performance

LRPC on a C – VAX FireFly takes 120 instructions round trip for null RPC. This is 157 microseconds. However, the C – VAX FireFly flushes the TLB on context switch. This costs 25% of the CPU time, but costs no instructions. For a reasonable machine that did not flush, this 25% could be saved.

For round numbers in this paper, the number of instructions necessary for a LRPC round trip is estimated as 100 – 200. The variability here is meant to reflect the variability in the instruction power of various architectures. LRPC optimizes the local RPC overhead, but still allows for remote calls. With the sharing of virtual memory, remote calls are almost impractical (although there is ongoing research in this area). Only local calls are considered here. Hence, some modest savings in call/return time can be obtained.

All of the calls and returns pass very little data. The marshalling time for pointers and OID's or TID's should be miniscule.

8. Conclusions

8.1 Conclusions for Operating Systems

An operating system should provide for fast inter – protection domain call and return. Either LRPC, or the techniques of ESA/370, Psyche's protected invocation, or something similar should be supported. LRPC is a better "fit" for most current operating systems, but all of these facilities provide similar functionality, protection, and performance for databases. The operating system must support a large number of protection domain (address spaces).

8.2 Conclusions for Computer Architecture

The MMU, TLB, and caches of a computer must be constructed to support fast inter – protection domain call and return. Assuming just a few "processes" will be active is wrong (e.g., 8 or 16 contexts on SUN's are too few). Flushing TLB's and caches on context switch cannot occur (e.g., TLB flush on the Firefly). Many LRPC calls will be short (100's or 1000's of instructions), so the TLB and cache should be left "warm".

The size of the virtual memory page is a problem. Pages provide both protection and clustering. Databases care about both clustering and protection. While 4 KB pages or larger make sense for disk based operations and for the granularity of physical memory allocation by the kernel, the virtual memory system should provide finer grain protection than whole pages. This would make it possible for objects from multiple protection domains to be kept on the same virtual memory page, but still have the proper access rights in the running processes.

For example, if the page size is be 4 KB, then 4 KB is the minimum amount that is read or written to the disk, and physical memory is allocated in 4 KB chunks. But each 4 KB page might have four protection regions of 1 KB each. Each sub – page could have different protection domains and rights.

8.3 Conclusions for Databases

The combination of LRPC and flexible memory sharing can give good performance ADT's to Object – Oriented Databases, Extensible Databases, or Third Generation Data Base systems. Many of the security problems are solved.

Please lobby your operating system and hardware vendors to provide the necessary facilities.

References

[Atki90] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. ``The Object-Oriented Database System Manifesto,'' copies of this were available at SIGMOD 90.

[Bers89] B. Bershad, T. Anderson, E. Lazowska, and H. Levy. ``Lightweight remote Procedure Call,'' Twelfth ACM Symposium on Operating Systems Principles, Litchfield Park, Arizona, Dec 1989, 102 - 113. Printed in Operating Systems Review, 23, 5.

[Lind87] B. Lindsay, J. McPherson, and H. Pirahesh. ``A Data Management Extension Architecture,'' Proceedings of SIGMOD 1987, San Francisco, Cal., May 1987, 220 - 226. Printed in Operating Systems Review, 23, 5.

[Mass89] H. Massalin and C. Pu. ``Threads and Input/Output in the Synthesis Kernel,'' Twelfth ACM Symposium on Operating Systems Principles, Litchfield Park, Arizona, Dec 1989, 191 - 201. Printed in Operating Systems Review, 23, 5.

[Ong84] J. Ong et. al. ``Implementation of Data Abstraction in the Relation System INGRES,'' ACM SIGMOD Record, March 1984.

[Scal89] C. Scalzi, A. Ganek, and R. Schmalz. ``Enterprise Systems Architecture/370: An architecture for multiple virtual address space access and authorization,'' IBM Systems Journal, 28, 1, 1989.

[Scot89] M. Scott, T. LeBlanc, and B. Marsh. ``A Multi - User multi - Language Open Operating System,'' Proceedings of the Second Workshop on Workstation Operating Systems, IEEE Computer Society, Pacific Grove, California, Sept. 1989, 125 - 129.

[Scot90] M. Scott, T. LeBlanc, B. Marsh, T. Becker, C. Dubnicki, E. Markatos, and N. Smithline. Implementation Issues for the Psyche Multiprocessor Operating System, University of Rochester Technical Report.

[Ston90] M. Stonebraker, B. Lindsay, J. Gray, M. Carey, M. Brodie, P. Bernstein, D. Beech. Third - Generation Data Base System Manifesto, U. C. Berkeley Electronics Research Laboratory Memorandum UCB/ERL M90/28.

XEROX

Secure and Efficient Implementation of Abstract Data Types for Databases

Robert B. Hagmann

CSL-91-5