Designing Persistence for Real-Time Distributed Object Systems[⋆]

Igor Nekrestyanov, Boris Novikov, and Ekaterina Pavlova

University of St.-Petersburg, Russia
`igor@meta.math.spbu.ru, borisnov@acm.org,`
`katya@meta.math.spbu.ru`

Abstract. An implementation of persistent object store for real-time systems with strict processing time constraints is a challenging task, because many traditional database techniques, e.g. transaction management schemes, are not applicable for such systems.
This paper examines technical and business requirements for one particular class of such systems and describes an architecture based on distributed shared virtual memory. The major contributions are: use of distributed dynamic hashing to achieve load balancing and tight coupling of transaction and virtual memory management, which allows local scheduling of read-only transactions.

1 Introduction and Motivation

During past two decades the object orientation evolved from a useful programming paradigm to a widespread technology which now addresses virtually all layers of software (from operating systems to application programs), all application domains, and all phases of the system life cycle.

In particular, the requirements of non-traditional database applications (such as CAD) stimulated rapid development of object-oriented databases and appropriate models.

The need to reduce software development costs together with rapid growth of global networks brought up the ideas of interoperability and software reuse as one of the major directions of research and development. Both software interoperability and reuse involve several issues related to different layers from formal compatibility of data formats (e.g. low-level network protocols), and interfaces (e.g. OMG IDL) to very deep problems of semantic interoperability.

One of very popular approaches to facilitate interoperability is creation of open systems that can communicate with other systems over network. The majority of modern systems (e.g. operating systems and DBMSs) are designed as *open* in the above sense. However, to enable efficient software reuse it is necessary, in addition, to open some of system internal functionality.

One of widely accepted ways to interoperability is through OMG CORBA architecture [10]. In this architecture, the system functionality, available via Object Request

[⋆] This work was partially supported by Russian Foundation for Basic Research under grant 98-01-00436, and UrbanSoft Ltd. under contract 35/98.

Broker (ORB), is split into several object services and common facilities, so that application may use these components as needed.

Ideally, the services should be mutually orthogonal. However, the notion of orthogonality is not precisely defined and therefore may be viewed only as a desirable but not reachable in practice. Some services, even specified as completely independent, inevitably have strong implementation dependencies. In particular, several services related to different aspects of persistence are closely related. Further, different implementations are required for various applications so that domain-specific requirements can be satisfied.

In this paper we describe an approach to implementation of the persistence and related services which can be combined into a consistent OMA-compliant architecture, providing efficient support for heterogeneous persistent real-time object environments.

Each of the desired properties is addressed by existing approaches and even commercial systems, but neither of these currently addresses the whole set of requirements:

- Distributed object environments provide for scalable, reliable, and cost-efficient computations, but, in general, do not address neither persistence nor real-time.
- Open architectures such as OMG OMA/CORBA provide basic interoperability in heterogeneous systems, but the problems of persistence and real-time become much harder than for homogeneous distributed systems.
- Persistent object environments do not address real-time constraints.
- Object-oriented DBMSs cannot efficiently support heterogeneous environments with mixed applications (e.g. transactional and non-transactional).
- Research on real-time database systems is restricted to transaction management in the presence of deadlines and do not address complex objects and queries, which may cause unpredictable response times.
- Virtual main memory databases are useful for implementation of persistence for real-time distributed object environments, but many of traditional databases techniques should be revised for this kind of database systems.

The requirements of real-time, especially performance requirements, are extremely hard to meet. We are interested in application domains were critical parts of distributed system should exhibit very high performance: most of transactions should be completed in tens of milliseconds after arrival of a transaction.

This level of performance cannot be achieved within current OMG CORBA architecture, because the best ORB implementations implementations consume at least few milliseconds per request. The standard OMG services require several requests to be issued when transaction starts or commits, locks are acquired or released, and even persistent data are accessed.

To bypass these problems, a hybrid architecture should be used, with special (streamlined) processing of requests which can be performed entirely in the time-critical part of the system.

Fortunately, due to special features of the application domain, it is possible to rely on assumptions which make the solution feasible. The most important assumption is that all time-critical transactions in the system are read-only, and a special kind of transaction scheduling can be used to meet performance requirements.

Although this assumption looks overrestrictive, it holds for applications related to embedded real-time systems with soft deadlines, e.g. telephone switches.

The paper is organized as follows.

Next section lists the requirements for the system, then the basic architecture of the proposed implementation is described and motivated. Further, use of shared virtual memory in the system, concurrency control, and the prototype implementation are described. A brief review of related work is followed by conclusions.

2 Requirements and System Architecture

2.1 Requirements and Assumptions

In this section we identify the special requirements of real-time applications to persistence and issues to be resolved.

Typical real-time requirements include:

– extremely high reliability
– transactions have firm deadlines with tight time limits
– the system is essentially distributed, and each transaction may involve more than one site.

However, applications we have in mind have special features which allows to reduce drammatically the number of remote object requests, streamlining the implementation of persistence-related services.

In this paper it is assumed that the following properties of the target application domain are valid:

– most of transactions are very short
– most of transactions are read-only, end almost all time-critical transactions are read-only ones
– the database may reside entirely in the main memory
– significant portion of data is write-protected for almost all transactions
– the size of individual data objects is relatively small
– the memory is partitioned into pages, so that objects cannot cross page boundaries.

Actually, the target system should process two different types of requests.

The major part of requests (regular service requests), should be processed with very tight time constraints. Typically, these requests should be processed in less than 100 milliseconds. Usually requests of this type come from hardware (a kind of interruptions), and almost all of them require only read access to persistently stored data (however, they need to see consistent state, so transactions are still necessary).

External requests without too strict time constraints. The of requests of this type normally come “from the network”, that is, are issued from heterogeneous environment including different hardware architectures, operating systems, and language environments.

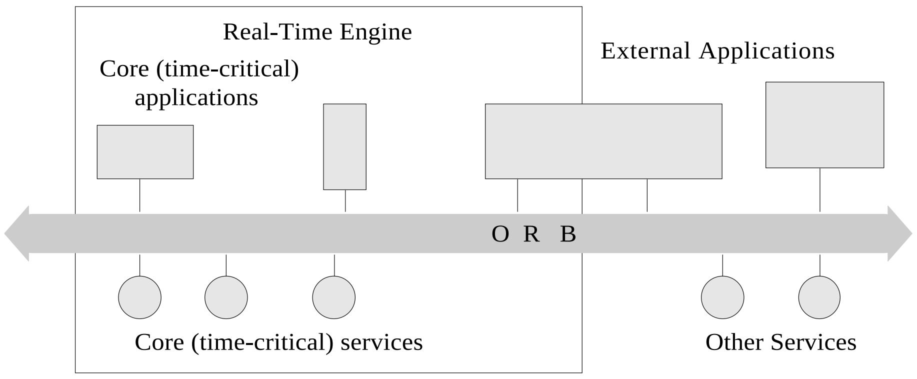
Real-Time Engine
Core (time-critical) applications
External Applications
O R B
Core (time-critical) services
Other Services

Fig. 1. The Basic Architecture

2.2 The Environment

Basically, the system consists of a *core engine* which processes all kinds of requests, and a set of *applications* which generate these requests (see Figure 1).

The external requests (including maintenance, statistics, analysis and other applications without strict real-time requirements) are issued from distributed heterogeneous environment via CORBA interfaces (this is a business requirement, rather than a designers' choice).

All time-critical requests come from hardware tightly connected with core engine, which is itself distributed (for scalability and reliability reasons).

The core stores significant amount of data permanently and thus may be considered as a database with usual requirements, such as consistency, data independence, etc.

In other words, the core engine may be considered as a real-time object-oriented DBMS with CORBA-compliant interfaces. It also contains significant amount of the application code for time-critical applications which also access data via CORBA type of interfaces.

However, as explained above, it is not possible to meet performance requirements within current CORBA implementations.

2.3 Performance Shortcuts

The architecture of the core engine follows the structure defined in CORBA, but the implementation relies on several *shortcuts* which provide fast inter-service interactions.

In order to tight time constraints, the OODB is implemented as a main memory database. The core engine is distributed, therefore, our main memory is actually a distributed shared virtual memory. Each process, however, runs in separate address space (which virtually includes the shared memory).

The following types of shortcuts are used in the core engine:

– Services may run as a single process and therefore share common address space. In this case, CORBA invocations may be replaced by direct calls.

– When both client and service reside in core engine, the shared virtual memory may be used to pass bulk amounts of data directly.
– Services may internally benefit from presence of shared virtual memory.

2.4 Objects

While the complete engine includes (relatively) high-level features and supports strongly-typed objects, only low-level storage aspects of the engine are considered in this study.

All objects in the core engine may keep (some portion of) their state in persistent memory area. In this project, we do not rely on a popular concept of *transparency of persistence* because it implies certain performance penalties, and actually significant percentage of all objects do not need it.

Instead, the (part of the) state of object which should persist is explicitly mapped to persistent memory area (implemented as shared virtual memory).

When an object needs to access it's persistent state, the appropriate portion of shared virtual memory is mapped to *the same address of* the process address space, so no pointer translation is needed. Thus the mapping of object state to persistent store does not imply performance penalties. However, this mapping is very important conceptually because it is a *forget* operator—it maps typed data objects representing persistent state to untyped *generic data objects*. The type information is stored as an attribute of generic object.

This dynamically typed (or untyped) objects layer was identified previously in other persistent object systems, e.g. Tycoon Storage Protocol [7] operates with a data structure very similar to generic data objects used here. The objects considered in [1] also possess dynamic type information and can be therefore viewed as untyped.

From this perspective, the CORBA persistence service may be described as a composition of mappings:

1. The application persistent object's (PO) transient state is mapped to a persistent state referred to by PID and represented with Data Objects.
2. Data Objects are mapped to untyped Generic Data Objects which may be handled directly in datastore.
3. The (transient) references to generic data objects are mapped to (persistent) locations in datastore.

Last mapping (usually called swizzling) was extensively studied in the literature and several options were evaluated and compared. We are considering this issue later in connection with the datastore structure 3.2.

3 Shared Virtual Memories

Significant performance gains (with respect to general purpose persistence as specified by OMG) can be obtained if the application persistent object is running on the site that can access the datastore memory. In this case the datastore memory may be shared between several clients.

Once the datastore memory is made accessible to client address space, the data can be accessed locally, either directly or via an advanced object request broker which can utilize this potential benefit.

The pre-requisite for this type of interaction is that datastore should run at the same site as application persistent object (PO). We cannot control the selection of the site for PO, hence it is necessary to relocate persistent store to gain from the potential benefits of co-location.

The straightforward solution, suggested in [1], is based on a concept of distributed shared virtual memory (DSVM). Although this solution is applicable only for homogeneous environments, which is not limiting restriction for time-critical applications.

The basic idea of the DSVM is to utilize huge virtual address space (64-bit addressable) and map it to address spaces of all sites that share this virtual memory. Some portion of the address space, should be, of course, reserved for private use of each particular site, but the size of address space provides a possibility to make this private part negligible.

With this type of environment, we may replicate data (on the virtual memory page level), so that each client can access it's persistent state locally, with additional advantages:

– The memory mapping is fixed, consequently addresses may be used as pointers, no swizzling is necessary.
– The conversions made by Type layer may be significantly simplified.

However, an attempt to implement this structure directly would face certain problems.

The first problem to mention is related to placement of persistent data in the virtual memory. If the object identifiers (represented as virtual addresses) are selected randomly, the address space will be extremely sparse, which cannot be efficient for any implementation of virtual memory. Other strategies may cause problems with load balancing. Under any allocation policy it is difficult to preserve load balancing.

Another problem is related to updates. If stored objects are claimed to be immutable (as in [1]), an additional directory is unavoidable on top of the described one (to manage references to objects representing the latest state of mutable objects), which results in extra indirection and hence performance penalties. Alternatively, if objects are relocatable, some expensive mechanizms (such as garbage collection) are un-avoidable.

Both problems mentioned above are the problems of location and identity. Next section explains how these problems can be addressed.

3.1 Object Identity and Location

The root cause of the problems listed above is that the functions of identity (provided by object identifier) and locality are often mutually exclusive. The identity should be immutable, while any dynamic environment will degrade if objects cannot be relocated for some reason.

The problem can be completely solved by additional indirection (with well-known disadvantages). To avoid extra indirection, this project relies on dynamic hashing. We are concerned with persistent object states, so the identifier in this section refers to identifier of a persistent object state in a datastore (to avoid mismatch with CORBA object reference).

The proposed structure is actually a combination of distributed shared virtual memory [1] and distributed dynamic hashing proposed in [6].

3.2 Distributed Hashing for DSVM

The shared virtual memory is segmented into pages of reasonable size. Actual locations are calculated from the values of identity using perfect hash function which yields the address of distributed shared virtual memory page where the corresponding persistent state is currently located. A small table in the header of each page is used to finally resolve the reference.

Modern hash schemes can avoid collisions, so there is no need in any overflow chains. Moreover, most of hashing schemes are dynamic, so the data will not be scattered over the whole address space. Instead, it will occupy reasonable amount of memory pages with controlled density.

An additional advantage is that the Identity space is not limited with address space size of the computer. The only requirement is that existing data should fit into available virtual memory address space.

The speed of modern connections is significantly greater than mechanical speed of disk storage. Therefore, it is reasonable to use main memory instead of disks to hold the pages. The amount of main memory in a distributed system should anyway sufficient to keep all data in the main memory.

So, instead of disk address, the hash function is used to calculate the address of a site which holds a primary copy of this page [6].

When the page is cached, the calculated pointer values may be re-used if refer to a page cached into the same address space, providing extremely fast access to persistent data.

In terms of address space, we assume that the virtually addressable space is huge, and distinguish the following ranges:

– private space of a process
– a range for cached pages of shared memory
– a space for pages hosted in this site.

Thus, the range to which a particular address belongs allows always easily decide if this address requires recalculation.

For the reasons of reliability, each page should be kept at more than one site. The modification of the hashing algorithm is also proposed in [6].

3.3 Concurrency Control for Shared Memories

The objective of any system that depends on persistent data is to preserve data consistency while providing best performance (e.g. maximal throughput or minimal response rime). The common way to improve performance, especially in distributed systems, is to increase degree of parallelism. This increase, however, is limited with requirement to preserve consistency, and concurrency control limits degree of parallelism in order to preserve consistency.

For this reason, we consider consistency as a primary objective, and the concurrency control as a tool which may be used to achieve it.

The Protocol. The concurrency control is a hard problem due to unacceptable processing overheads (e.g. locking of a single data element may require several remote procedure calls, each of which may consume few milliseconds.

On the other hand, the probability of conflicts is extremely low. For this reason, a concurrency control algorithm with low overhead is needed. Optimistic concurrency control protocol which can meet these requirements is briefly described here. More detailed discussion is available at [9].

The main idea of our protocol is to use virtual memory management to trace access to persistent data and to detect the inter-transaction conflicts.

To keep overhead minimal, we choose page-level granularity of access control. The objects are expected to be very small, hence object-level control would result in increased amount of processing necessary to keep track of accesses. This decision might lead to decrease of concurrency, but this is not for read-only transactions, which are our major concern.

The Model. In addition to system features that were outlined above we need few additional assumptions.

We assume that each transaction runs within a process and each process has it's own virtual 64-bit address space. Each address space have own page table and there is a common global page table.

The transaction manager can influence the page replacement policy of virtual memory manager through call-back functions. Finally, hardware can trap both read and write (to protected) page faults.

Data Structures. We assume that the transaction manager has access to page tables of all address spaces that are in use on the same processor. The transaction manager may either use separate process or may be invoked from regular application process (e.g. via exception handler), and in any case it can also access global page table.

The transaction manager maintains a transaction counter, which should be globally unique. This counter may be also used to define serialization order.

In this protocol, we use multiple versions of pages and allow read-only transactions to access previous states of updated pages.

To implement this feature the transaction manager should maintain additional data structures to keep track of existing versions for each (recently) updated page, and this data should be consistent for all transaction managers.

Each version of a page has associated version number, which is equal to global part of the transaction identifier (which is increased when writing transaction appears).

During the initialization of a transaction an interval of admissible version numbers for this transaction is calculated.

Transaction Identifiers. When the transaction manager registers new transaction, a globally unique transaction identifier is created.

Each transaction manager maintains local transaction counter, and the global counter is used only for writing transactions.

The transaction identifier should contain the value of global counter (advanced when a writing transaction commits), transaction manager identifier, and local sequential number (relative to the global number).

The advantage of this implementation is that the identifiers for most frequent read-only transactions can be assigned locally on each processor without any synchronization with other transaction managers.

Operations. To present the protocol, we further describe actions performed by transaction manager on certain events.

Read Operation: When a transaction attempts to read data from a page of shared persistent memory, the transaction manager looks for the page address in the global page table. If the page is found there, this page has only one version, which is made accessible to the requester.

Otherwise the transaction manager looks for appropriate version of a page in the list of versioned pages and selects a version which is suitable for the requesting transaction (the selection is based on the transaction identifier and the version number associated with each version of a page).

Write Operation: When a transaction attempts to update the data in shared persistent memory, an exception occurs, because initially each page is available in (at most) read-only mode.

If transaction attempts to write to the latest (or the only) version of a page then exception handler then creates a private copy of the page to be accessed and appropriately updates the local page table. This copy may or may not be allocated on the persistent memory. Otherwise the transaction is aborted.

In addition, the transaction is registered as writing transaction and the manager makes housekeeping records to be able to handle the private (shadow) copy properly when the transaction terminates.

Abort: In case of abort of the read-only transaction the information about this transaction is discarded. When writing transaction aborts all updated pages (that is, private copies) are simply discarded.

Commit: The read-only transaction can commit regardless of any other circumstances, because proper versions of each page it accessed were provided at the first access.

The validation of the writing transaction includes check of all updated pages. If for some pages which were updated by the transaction in question more recent versions exist the transaction should be aborted, otherwise it commits.

Rollback: In case of abort of the read-only transaction the information about this transaction is discarded. When writing transaction aborts (e.g. for it's internal reasons), all updated pages (that is, private copies) are simply discarded and all information about this transaction may be erased.

Correctness. The correctness of this protocol is obvious because every committed read-only transaction work with same state of database which was at the time of its start.

3.4 The Performance Model

Proposed concurrency control algorithm has little overhead, but may potentially result in high rate of transaction aborts (typical for all optimistic algorithms). To estimate the probabilities of successful transaction commits, a simple model was used.

For the case of read-only transaction the estimation is trivial, because they always commit, and $P_{read-only} = 1$. For updating transactions the model depends on the following parameters:

N — total number of data elements,

n — average number of data elements used by single update transaction,

l — average length of transaction,

k — average number of transactions per second.

Then the probability of successful commit is estimated as

$$P_{update} = \frac{1}{1 + (k * l - 1) * P}$$

where $P = \min(k * l * (1 - \frac{((N-n)!)^2}{(N-2n)! * N!}), 1)$.

Estimations obtained from this model show that the abort rate is very small, which is not surprising under our assumptions.

4 Implementation Environment and Prototype

The prototype was developed on a network consisting of SunSPARC 5 workstation and two Intel CPU-based computers running under Solaris 2.5, Windows NT 4.0 and Windows 95, respectively.

The development of a prototype was based on ILU - the Xerox Research implementation of [10], which implements (a subset of) CORBA 2.0.

To estimate potential performance gains, sometimes direct interaction of the objects that share common address space is used in the prototype, instead of regular ORB requests, when the placement of the objects in the same address space is reasonable.

The complete implementation of the datastore structure requires significant support from the operating system. The prototype only models some of the critical features. The implementation is based on Unix concept of mapped memory (mmap system call). This function effectively provides for persistence of main memory segments.

The limitation is that the addresses cannot be preserved in this approach, so this part of the engine is not modelled in the prototype.

The prototype includes persistence and transaction services, a virtual memory datastore implementation on UNIX system, and example clients.

The prototype exhibits expected performance and demonstrates feasibility of the reliable and efficient datastore implementation based on the concept of distributed shared virtual memory.

5 Related Work

Distributed object systems over virtual memory are described and implemented in several papers and projects, e.g. [4, 11, 12, 2]. however, neither load balancing nor object (de)clustering is considered in these projects.

Distributed persistent object directory based on shared virtual memory is described in [5, 8, 1], but the architecture proposed there does not take into account performance issues.

Distributed dynamic hashing schemes were described in series of papers, e.g. [6]. The structures described there are designed for performance and reliability, but applications to object systems are out of the scope of these papers.

Typeless representations for persistent objects in distributed heterogeneous systems are defined in [7]. This work does not consider neither real-time requirements nor performance issues.

Concurrency control protocols for real-time databases were studied in [3, 13, 14]. However, this work is mostly related to protocol enhancements that make response time predictable, and does not consider actual implementation.

6 Conclusions

The paper describes potential implementation of persistent object store for real-time systems with very tight time constraints. The implementation is based on distributed shared virtual memory combined with distributed dynamic hashing.

The major contributions of the approach are:

- The coupling of concurrency control with virtual memory management provides very efficient processing of time-critical transactions. Read-only transactions may be scheduled locally (without any remote calls).
- Use of dynamic hashing for object placement results in increased flexibility and load balancing.

The feasibility of this design was demonstrated on a prototype, but actual performance characteristics still need further investigation.

References

1. K. Barker, R. Peters, and P. Graham. Distributed Shared Memory for Interoperability of Heterogeneous Information Systems—*Position Statement*. In *OOPSLA Workshop on Interoperable Objects—Experiences and Issues*, Oct. 1995.
2. D. Hulse and A. Dearle. A Log Structured Persistent Store. In *Proceedings of the 19th Australasian Computer Science Conference*, pages 563–572, Jan. 1996.
3. L. Juhnyoung and H. S. Sang. *Performance of Concurrency Control Mechanisms in Centralized Database Systems*. Prentice-Hall, 1996.
4. E. Koldinger, J. Chase, and S. Eggers. Architectural Support for Single Address Space Operating Systems. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS*, Oct. 1992.
5. A. Lindstrom, A. Dearle, R. di Bona, A. Norris, J. Rosenberg, and F. Vaughan. Persistence in the grasshopper kernel. In *Proceedings of the 18th Australasian Computer Science Conference*, pages 329–338, Feb. 1995.
6. W. Litwin, M.-A. Neimat, and D. Schneider. RP*: A Family of Order Preserving Scalable Distributed Data Structures. In *Proceedings of the Twentieth International Conference on Very Large Databases*, pages 342–353, Santiago, Chile, 1994.
7. F. Matthes, R. Mueller, and J. W. Schmidt. Towards a unified model of untyped object stores: Experience with the tycoon store protocol. In *Proc. of the Third Intnl. Workshop on Advances in Databases and Information Systems - ADBIS'96*, pages 1–9, Moscow, Sept. 10–13 1996. MEPhI.
8. H. Moons and P. Verbaeten. Persistence in Open Distributed Systems: The COMET Approach. In *Proceedings of the Euromicro Workshop on Parallel and Distributed Processing*, pages 342–349, Jan. 1994.
9. I. Nekrestyanov, B. Novikov, E. Pavlova, and S. Pikalev. Concurrency Control Protocols for Persistent Shared Virtual Memory Systems. In *Proc. of the Workshop on Advances in Databases and Information Systems - ADBIS'97*, pages 035–039, St.-Petersburg, September 2–5 1997. Nevsky Dialect.
10. Object Management Group and X/Open. *The Common Object Request Broker: Architecture and Specification, Revision 1.1, OMG Document Number 91.12.1.* OMG, 1992.
11. R. Schmidt, J. Chase, and H. Levy. Using Shared Memory for Read-Mostly RPC Services. In *Proceedings of the 29th Hawaii International Conference on System Sciences*, Jan. 1996.
12. M. Shapiro and P. Ferreira. Larchant-RDOSS: a Distributed Shared Persistent Memory and its Garbage Collector. In *Proceedings of the 9th International Workshop on Distributed Algorithms*, Sept. 1995.
13. O. Ulusoy. Analysis of concurrency control protocols for real-time database systems. Technical Report BU-CEIS-9514, Bilkent University, 1995.
14. O. Ulusoy and G. Belford. A performance evaluation model for distributed real-time database systems. *International Journal of Modeling and Simulation*, 15(2), 1995.