Simple and Efficient Transactions for a Distributed Object Store

Laszlo Boeszoermenyi and Carsten Weich
Institut fuer Informatik, Universitaet Klagenfurt,
Universitaetsstrasse 65 - 67, A-9020 Klagenfurt, Austria,
e-mail: {laszlo,carsten}@ifi.uni-klu.ac.at, tel/fax: (43)-(463)-2700-509/505
www: http://www.ifi.uni-klu.ac.at/home?{laszlo,carsten}

Abstract

Even the more or less "canonical", lower-level architecture of information systems needs to be revisited from time to time. Notions like persistence and transactions belong traditionally to the area of database management systems.

There are, however, many applications, such as CAD, VLSI design or simulation, which need persistence and could take advantage of transactions, but require especially fast implementations not provided by DBMS. In this paper we are describing a low-level transaction concept used to implement our parallel main memory object store (PPOST), to provide main memory access times combined with the safety and convenience of transactions.

Keywords: parallel systems, distributed systems, object databases and persistence.

1. Introduction

Despite of the amount of knowledge we have about object-oriented systems, some crucial points are still open. The work described in this paper has been motivated by the following considerations.

We are looking for a way to cope with storing a large number of objects in an object space that is most likely persistent in time and possibly distributed in space. What we need is first of all the proper abstraction that can hide the details of persistence and distribution and that enables fast retrieval of required objects.

As such a proper abstraction we suggested the use of polymorphic, persistent and distributed sets in some previous papers [3, 5]. Sets are on the one side associative, i.e. well-suited for queries depending on the actual content of certain object-attributes. On the other side, sets are per definition unordered, therefore operations on their elements can be easily parallelized. The proposal in [5] contains language elements for powerful, declarative set-operations (first of all a generalized selection) that can be optimized and parallelized relatively easily. All proposed set-expressions yield sets again as result, and therefore can be applied recursively.

The operations can be *composed* to still more powerful ones. Both simple and composed operations on persistent and distributed sets have to fulfill some basic requirements: they must not lead the set into an inconsistent state, they must not interfere with each other and they must not loose legal changes of the stored data. If we take a closer look at these requirements, we immediately see that they lead to the usual ACID-conditions of *transactions*.

In this paper we describe a concept for fast distributed transactions. The concept was implemented at the heart of a main-memory based parallel object store, described in [6, 4, 7].

2. Transactions for distributed object stores

Object-oriented applications build up and manipulate a data structure which consists of large numbers of objects structured in complex aggregations, as lists, sets and relations. Typical engineering applications which fall into this category are *design applications*, like CAD, VLSI design and the like. These applications have high performance requirements. They also have requirements which are traditionally served by database management systems: Their data must be kept persistent, even in presence of system failures. Usually several users access the same data; they must be protected from disturbing each other. Conventional relational database systems cannot be used for such applications: The relational data model is not expressive enough to conveniently represent their object structures [13]. In addition, they are usually too slow to meet the performance requirements [9].

Object-oriented database systems were designed for such applications. They store the application's data struc-

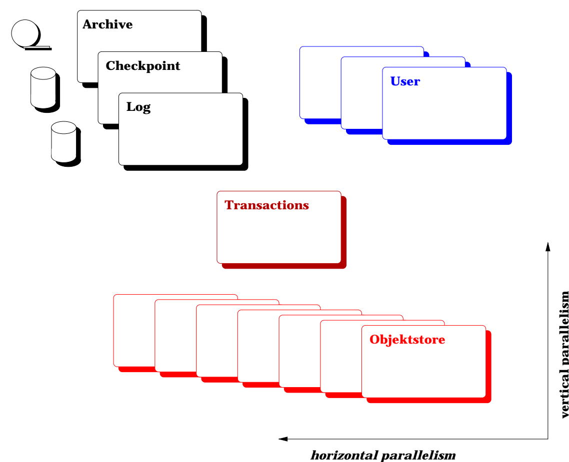

Figure 1. PPOST: A main memory distributed object store architecture

tures on disc, but they usually include features to check out data to main memory to meet the performance needs.

We propose to view the problem from a different perspective: We claim that engineering applications should store their data structures primarily in main memory. There should be no copying or conversion necessary to access the data from the application code. All that is needed is a mechanism to guarantee the persistence of the data. We propose to use the main memory as primary storage area, and to use the disc only for guaranteeing persistence in case of system crashes. The disk is used only in sequential mode (to write a log of changes), no direct access is necessary. Thus, discs operate under optimal conditions and at full speed. To guarantee consistence of safe-points and concurrent data access of several users a simple transaction mechanism shall be provided by the system. Database management systems can be built on top of this transaction mechanism, so that the application developer can choose to take advantage of additional features provided by the DBMS only if it is needed.

Main memory has the obvious disadvantage of being restricted in size. We propose to use the main memory of more than one machine to make the size of the application data scalable: If the size of the data increases we add machines to the pool of storage machines. Thus, we not only get more storage capacity but also more processing power to work with the data. We already have proposed the PPOST architecture [6, 4, 7] in which several nodes of a cluster of workstations are used to store data in main memory (Figure 1). One node acts as master node (*transaction node*). User processes access data via the transaction node. In the background - decoupled from the operation of the object store -, a checkpoint processor produces a disc copy of the main memory database with the help of the log files produced on each node.

Problems of distributed transactions

A common technique to implement transactions on central disc based systems involves locking of data which is read or written by a transaction to provide consistent access [10]. In a distributed system locking is much more difficult: If a node reads the value of an object from another node it will cache this value for repeated access (otherwise reading an object repeatedly would be extremely expensive). If an object is to be modified during a transaction, not only the original copy has to be locked but also all its replicas in the cache of other nodes.

In a distributed environment with k copies of an object we need at least $2k+3$ messages for a value update: A lock request, a lock grant, k update with k acknowledge messages and an unlock request [10]. This can only be achieved if we store the locking information on some central location (like a dedicated server – which could then do deadlock detection also – or on the owner node of the object). Using this scheme we also have to contact this central location even if we only read the object (and even if we have cached the objects value locally). If reading is more frequent than writing, which is very likely, another scheme is needed: Lock objects and replicas only for writing. Then we need at least $3k$ messages: k update/lock requests to the replicas, k acknowledgments and k unlock messages [2]. Alternative algorithms are imaginable, but they all suffer from the same problem: Before changing an object we always have to first make sure that it is “free” and after changing it we have to propagate the new value. Both parts of the operation require information exchange among different locations.

In addition to this expensive synchronization, automatic detection and resolving of deadlocks is necessary if more than one transaction is allowed running at the same time. This requires either a central location collecting the locking

states or a sophisticated distributed algorithm. Both possibilities again require additional message passing.

Possible solutions

Let us now consider some solutions to the above described problem:

1. *No caching* (i.e. avoid replicas at all)
   would prohibitively slow down data access.

2. *Reduce latency time*
   would require expensive non-standard communication hardware.

3. *Enlarge lock granularity*
   e.g. by locking whole sets of objects instead of single ones. Thus, the number of locks (and lock messages) can be greatly reduced. On the other hand, this could drastically increase the probability of deadlocks.

4. *Use optimistic transaction implementation techniques*
   Such techniques are free of deadlocks and do not require locking. But they have their own synchronization problems: The validation phase still must be done in an atomic action (instead of the whole transaction). So in the end this technique does not help a lot in distributed systems because it only delays the same distributed synchronization problem to the validation phase. In addition, the technique increases the probability of transaction roll-backs *after* they have already been processed.

5. *Prohibit inter-transaction parallelism*
   Most of the described problems do not occur if only one transaction is allowed to run at the same time.

Alternative 5 looks like a serious restriction, but in distributed systems it turns out that this restriction makes important optimizations possible. Locking of objects and immediate propagation of new values can be eliminated all together in principle: Since there is only one transaction active, which does not have to be protected against interference by others, the system's efforts to synchronize shared access can be reduced. We will take a closer look on that in the following section.

3. Distributed Transaction Consistency

If we rule out parallel execution of different transactions we still have to deal with the synchronization of data access inside a transaction (intra-parallelism). Corresponding to our basic model, objects of distributed sets maybe located on different physical nodes. Transactions consist basically of a chain of calls of methods - which can be executed on different physical nodes in parallel. Parts of a transaction, running on several nodes, might write the same object, thus causing a racing condition. This happens frequently, e.g., if we update complex objects - sharing common parts, spread over several different nodes. There are three principal possibilities to track the problem:

1. Ignore it: Let the application programmer be responsible for synchronization. This is of course easy to implement, but actually impossible to use [16].

2. Guarantee sequential consistency [14]: Let the system be responsible to provide the application a view of the data as if it were stored on a single location. This is easy to use, but leads to similarly inefficient algorithms (see, e.g., the *two-phase update* protocol in [2]), as two-phase locking, as discussed earlier.

3. Look for a compromise: Let the system provide as much consistence as necessary for as low costs as possible.

3.1. Transaction consistency

We now propose a consistency model suitable for distributed programs with short transactions. We call a system *"transaction consistent"* if it guarantees the following condition:

1. Before and after a transaction execution all read requests from every node yield the same value for all objects. Write requests are not permitted outside transactions.

2. During a transaction read requests either return

   (a) the value produced locally on the requesting node,

   (b) or the value the object had at the beginning of the transaction.

3. During transactions write requests are sequentialized, i.e., no concurrent writes are permitted. In order to submit a write request, the program first has to request an exclusive, writable copy of the object. The changes that were previously made to the object on other nodes will be visible to a writing node. Changes made on other nodes after the write request was finished are not visible (unless the exclusive copy is requested again).

4. At the end of the transaction the system is responsible to guarantee condition 1 again. I.e., read requests that come in during transaction termination are delayed until changed values are propagated. If the transaction has aborted, the state valid at its start is restored.

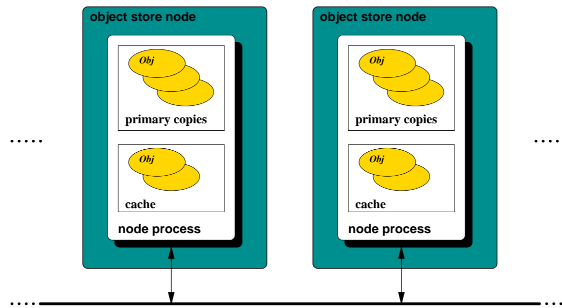

Figure 2. Nodes of the object store

This consistency model is especially suitable to support caching: Once a value has been transfered to a node to serve a read request it is not necessary to check whether it has been changed until the end of the transaction. This is also true after it has been updated: The object stays in the cache, its cached value is used for local reads.

The resulting programming model is fairly simple. The programmer knows: Read requests on a certain node either return the old value of the object or the value locally produced on the very node. It is never necessary to prepare for updates which come in asynchronously from foreign nodes. (With sequential consistency each method accessing a shared object has to deal with the fact that the object's value might change during execution of the method.)

Since write requests are sequentialized the model guarantees the memory coherence condition [1] inside transaction execution: i.e., writes to a single object are "sequentially consistent" – all changes are seen in the same order on all nodes. However, writes to several objects can be observed in different order on different nodes. A proof of the memory coherence guarantee is given in [16].

Using this model leads to short transactions. Several updates which depend on each other must be done in more than one transaction. If, e.g., you delete a number of objects in a transaction you will in general need another transaction to do consecutive updates. There is no easy way to "wait for the completion of an update" inside a transaction, if the update is done on a foreign node. (You could request the actual copy, but, without explicit synchronization, you would not know if you got it before or after the method you are waiting for has written to it.)

The model can be implemented efficiently. For a read request, only two messages are required (a request and a value message), consecutive reads of the same objects never need messages. For a write request, two or three messages are required each time the exclusive, writable copy must be allocated (a request, possible a delegation to the current location of the exclusive copy and a value message). In case of conflicts, the exclusive copy has to change its location several times, requiring message passing. If no conflict occurs, no further message passing is necessary (see Section 4). There are no locks required, apart from holding an exclusive copy. If every node is only allowed to write to a single object at the same time the system is free of deadlocks.

With the suggested transaction consistency we can expect short transactions which run very fast. Thus, the restriction that only a single transaction may be executed simultaneously seems acceptable.

4. Implementation

We have implemented a prototype of an object store using the transaction consistency model. It is described in detail in [16]. In this section we concentrate on describing the implementation of the transaction protocol. We explain how objects are stored in main memory, how they are accessed from other nodes and how they are updated during a transaction. To verify the prototype we have implemented the OO1-benchmark [9]. We conclude the section with reporting the experiences we have made.

4.1. Storing objects in main memory

The object store - implemented in Modula-3 [15] - consists of a number of node processes, a single process on each separate machine. The node processes store objects in their address space and feature a cache which also stores objects (Figure 2). Objects are ordinary Modula-3 objects, i.e. type-safe pointers to a record and a method suite. Objects always have an owner node assigned, this is the node that stores their primary copy. On other nodes, values are only stored in the cache. The primary copy is never changed until a transaction terminates successfully, all updates are done on a copy of the object in the cache, even on its owner node. To every object a reference is generated at initialization.

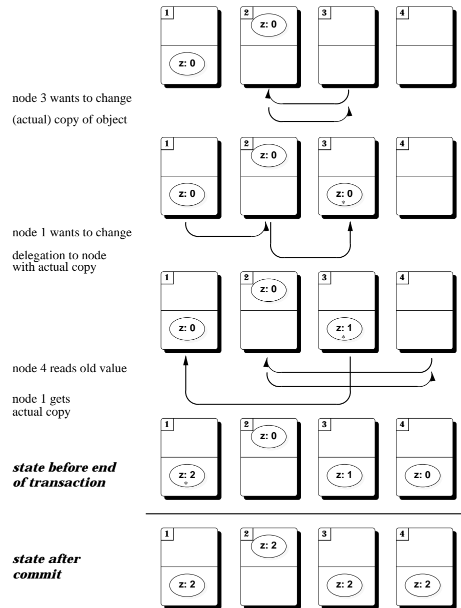

Figure 3. Write and read requests with transaction consistency

4.2. Transaction consistency protocol

To implement the transaction consistency model (3.1) read and write requests are processed as described below. An application never accesses an object directly, requests always need the reference to the object as parameter. The owner node of the object and its type can be determined from the reference. The result of the request is a readable or writable in-memory copy of the objects value:

- *Reads*
  First the cache is checked and the value there is returned if available. If not, the primary copy is returned if it is a local request. On all other nodes a message is sent to the owner node which returns the value of the primary copy of the object. This value is then put into the cache.

- *Writes*
  First the cache is checked whether the exclusive copy of the object is available. If so, the object is locked locally and made available to the caller. If not, a request for the exclusive copy is sent to the owner node of the object. The owner node immediately notes the requester as "current holder of the exclusive copy of this object". It sends a message to the preceding holder to pass the exclusive copy to the new one as soon as it is available.

  If a node receives the request to pass an exclusive copy to another node it checks that it is not locked. As soon as the exclusive copy is free (i.e. the write request on this node has already been completed), its value is passed to the future holder. The copy remains in cache but it is noted that it is no more the exclusive version of the object.

  The node waiting for the exclusive copy to process its write request puts the object in its cache when it arrives, marks it as "exclusive version", locks it and finally makes it available to the caller.

Figure 3 shows an example of how the protocol works (the exclusive copy is marked by an asterisk).

4.3. OO1-Benchmark results

We have used the OO1-benchmark [9] to verify our object store. There are newer benchmarks but OO1 was designed to measure object-oriented databases for exactly that kinds of applications we want to support. It tests the most basic operations efficiently and is especially easy to implement.

The measurements we have done with the prototype are described in detail in [16]. The OO1-benchmark results are listed in Table 1, they are compared with the best OO1 results published in [9].

The numbers show two things: On the one side, the prototype needs improvement. Some obvious optimizations have to be done yet. On the other hand, we were able to demonstrate that the transaction consistency protocol works and has the potential to exploit the processing power of a

Operation	Prototype (4 nodes)	Best OODBMS
build time	332.63	50.00
disk size	5.5MB	3.4MB
reverse traverse	2.23	13.00
lookup	2.84	13.00
traverse	3.01	13.00
insert	4.98	6.70
lookup warm	0.68	0.03
traverse warm	0.58	0.10
insert warm	2.35	3.10
sum cold	10.83	33.00
sum warm	3.61	3.20

times in sec

Table 1. OO1 benchmark results (compared with the best in [9])

distributed main memory object store. Processing of locally available objects can be done at full main memory speed, there is no buffering, copying or conversion necessary – only a function call to check whether the object is in the cache. Accessing a remote object basically needs the time to transfer the value over the network, only in case of write conflicts is some overhead needed, to look up the current holder of the actual version. Very basic operations like selection or aggregation calculation can be parallelized easily without the user having to struggle with the complexity of parallel programming.

5. Conclusion

A simple transaction concept has been presented for a distributed, main-memory based object store. The key idea lies in providing only intra-transaction but no inter-transaction parallelism. Concurrent transactions are actually sequentialized as a whole – but executed very fast, by taking advantage of a specially adopted protocol for accessing shared data and by internal parallelism. During the execution of a transaction the coherence of the distributed object store is guaranteed. At transaction commit the object store is resynchronized and thus becomes consistent. Some measurements have been shown to demonstrate that the suggested transactions can be very fast.

In the *future* we are looking for algorithms and data placement strategies which make PPOST faster. We also have to understand better, how to use the suggested transactions, especially, how could they support applications relying on *snapshot isolation*.

References

[1] M. Ahamad, R. A. Bazzi, R. John, P. Kohli, and G. Neiger. The power of processor consistency. Technical Report GIT-CC-92/34, Georgia Institute of Technology, 1992.

[2] H. Bal. *Programming Distributed Systems*. Prentice Hall, 1990.

[3] L. Boeszoermenyi. Distributed sets of objects. In *ECOOP*, Linz, 1996. position paper in workshop-9.

[4] L. Boeszoermenyi, J. Eder, and C. Weich. PPOST: A parallel database in main memory. In D. Karagiannis, editor, *Lecture Notes in Computer Science 856, Database and Expert Systems Application (DEXA)*, Athens, Greece, September 1994. Springer.

[5] L. Boeszoermenyi and K. H. Eder. M3Set – a language for handling of distributed and persistent sets of objects. *Parallel Computing*, 22 (1997) 1913 - 1925.

[6] L. Boeszoermenyi, K. H. Eder, and C. Weich. PPOST a parallel persistent object store in main memory. In *Database and Expert Systems Applications, Fifth International Conference (DEXA)*, Athens, Greece, September 1994. Springer Verlag.

[7] L. Boeszoermenyi, K. H. Eder, and C. Weich. A very fast parallel object store for very fast applications. *Simulations Practice and Theory*, 5 (1997) 605 - 622.

[8] M. J. Carey, D. J. DeWitt, and J. F. Naughton. The oo7 benchmark. Technical report, University of Wisconsin-Madison, 1994.

[9] R. G. G. Catell. An engineering database benchmark. In Gray [12]. Chapter 7.

[10] C. J. Date. *An Introduction to Database Systems*. Addision Wesley, 1990.

[11] D. J. DeWitt, J. F. Naughton, J. C. Shafer, and S. Venkataraman. Parallizing OODBMS traversals: A performance evaluation. *VLDB Journal*, 5:3–18, 1996.

[12] J. Gray, editor. *The Benchmark Handbook*. Morgan Kaufmann Publishers Inc., second edition, 1993.

[13] A. Kemper and G. Moerkotte. *Object-Oriented Database Management*. Prentice Hall, 1994.

[14] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transaction on Computers*, C-28:690–691, Sept 1979.

[15] G. Nelson, editor. *Systems Programming with Modula-3*. Prentice Hall, 1991.

[16] C. Weich. *Hauptspeicherdatenstrukturen und Transaktionskonzepte fuer eine objektorientierte Datenbank*. Dissertation, Technische Universitaet Wien, 1996.