Weakly Consistent Transactions in *ROSS*†

Barry Brachman and Gerald Neufeld
brachman@cs.ubc.ca, *neufeld@cs.ubc.ca*

Department of Computer Science
University of British Columbia
Vancouver, B.C. V6T 1Z2

DRAFT – Do Not Distribute

Abstract

This paper describes the design of the weak consistency scheme used in *ROSS*, the EAN object store. *ROSS* supports nested atomic transactions on distributed and replicated objects. The weak consistency method falls into the family of optimistic protocols. After a partitioning, execution of transactions proceeds normally. If write-write conflicts are detected when partitions later merge, transactions may be rolled back to ensure consistency. One-copy serializability is not provided. This approach is particularly well suited to a common class of database applications where there is limited interdependency between objects. A distributed name service is one such application.

1 Introduction

The distributed, replicated object store service (*ROSS*) provides a highly available, application-independent object storage facility. Each instance of an object store, called a *replica*, has its own database of objects. *ROSS* provides the programmer with a simple interface to atomic transactions on objects, largely hiding the fact that the object store is distributed and replicated.

ROSS gives the programmer two types of object consistency. A strongly consistent transaction uses the conventional, quorum-based virtual partition algorithm [3]. To provide greater availability, however, the object store also supports a much weaker form of replica consistency. This paper discusses the design of *ROSS*'s weak consistency mechanism.

ROSS's notion of weak consistency has been designed to efficiently support applications that use objects having limited interdependencies, although general database structures are possible. These applications require atomic transactions to ensure that updates to two or more objects are done in an all-or-nothing fashion. The nature of many applications is such that the probability of conflicting transactions is low, perhaps because the frequency of updates is low or there is a one-to-one mapping between an object and the location of its administrator. Also, in many applications it is not necessary for read operations to always return the most recent version of an object. A distributed name service is an example of an application having these characteristics. Some types of applications, such as network monitoring, may wish to enforce a weak form of consistency but permit transactions to occasionally be "lost" in exchange for higher throughput. This is analogous

† This work was made possible by a grant from OSIware, Inc.

to an application choosing to use unreliable datagrams to transmit real-time voice. Weak consistency replication may be appropriate in situations where disconnected operation is possible, as with portable computers.

Protocols used to enforce strong consistency, which make the collection of replicas behave as a single copy as far as applications are concerned, can be inefficient if there are many replicas, communication links are slow or unreliable, or hosts crash frequently. The need to retain locks while a distributed commit protocol is being executed can degrade transaction throughput. Also, in the event of a partitioning, availability can be severely restricted.

The semantics of weak consistency transactions are a unique characteristic of *ROSS*. Its goal is to provide the utmost availability in situations where conflicts among transactions are unlikely and temporary inconsistencies are acceptable. *ROSS*'s weak consistency is an optimistic protocol [6]. In the event of a partitioning, transactions proceed normally; any copy of an object may be read or written. Write-write conflicts are handled automatically. A relatively simple synchronization algorithm is used to ensure that all replicas eventually converge to identical object values.

The next section outlines the design of the weak consistency mechanism. Section 3 discusses related work and Section 4 concludes the paper.

2 Weak Consistency

An atomic transaction that is weakly consistent is committed in the usual way at the *originating replica*. It may later be rolled back *asynchronously*, however, because of a conflicting transaction at another replica. That is, although the application receives an indication that the transaction has been successfully committed locally, if it is found to conflict with a transaction executed at another replica it may be rolled back. The ultimate fate of a locally committed transaction is unknown until all replicas receive the transaction and report on the outcome. A weakly consistent transaction can therefore be viewed as similar to a nested transaction where the top-level commit operation is performed asynchronously or lazily and some intermediate states are externally visible. As a result, subsequent transactions may observe temporary inconsistencies. These subsequent transactions are not rolled back, however, as only write-write conflicts and not read-write conflicts are dealt with. The locally executed portion of all transactions is based on an optimistic, single-site nested transaction model [9]. No distributed locking is performed; once a transaction is locally committed all locks associated with the transaction are released.

In the remainder of this section, the normal (error-free) operation of the protocol is described, followed by a discussion of how the object store recovers from communication errors, crashes, and partitionings, and how replicas deal with conflicting transactions. The section concludes with an example application.

2.1 Normal Operation

When an object is created by the object store, it is assigned a globally-unique object identifier (`ObjOid`) that is never reused. An application is free to create objects that contain object identifiers. The object store provides primitives to start, abort, commit, and precommit transactions.

The weak consistency mechanism is only associated with update transactions. An update transaction is one that creates, modifies, or deletes one or more objects. As part of the top-level commit operation, update transactions are unreliably transmitted

to each replica from the originating replica. Each transaction is timestamped and assigned a unique identifier that includes a (locally) consecutive sequence number. A best-effort attempt is made to distribute the update to each participating replica, using (possibly broadcast or multicast) datagrams or connection-oriented communication. The update is not acknowledged.

At the originating replica and each replica receiving an update, a local transaction log record for the transaction is created and the replica's augmented *version vector* [17] entry is updated to reflect the new transaction. As illustrated in Figure 1, each replica has an entry in the version vector. An entry consists of a unique version identifier, which enables a replica to quickly detect when the entry has changed, and a transaction list. The transaction list consists of a sequence of transaction identifier, transaction state pairs. The transaction is applied to the database and undo information is kept in case it is aborted later. Each replica maintains a version vector entry for its own transactions and tracks the version vectors entries of all other replicas. Replicas periodically broadcast their complete version vector so that others can determine which updates they have missed.

Figure 1: The version vector

Each transaction in the system whose global outcome is uncertain is in one of three states at each replica: `unknown`, `aborted`, or `precommitted`. If a replica is unaware of a transaction (i.e., it has not seen the transaction identifier yet), the transaction is implicitly in the `unknown` state at that replica. Transaction propagation in the presence of failures is discussed in the next section. If the transaction is aborted at any replica, its state becomes `aborted` in the version vector and all replicas will eventually abort the transaction. If the transaction is successfully committed locally, it is marked as `precommitted`. It is globally committed when all replicas consider the transaction to be `precommitted`. When all replicas abort or commit a transaction, the transaction list item, undo information, and other transaction log information can be deleted. This can be determined locally from the version vector itself.

2.2 Transaction Propagation

A replica can miss an update transaction because it crashed or through a communication failure. In the meantime, transactions at other replicas proceed normally. A replica can determine which transactions it has missed by comparing an updated version vector with its previous version vector. By examining the version vector it can find out which other replicas have seen the update and can contact any such replica to obtain the transaction.

2.3 Conflict Resolution

If a replica finds that the write-sets of two transactions intersect (i.e., they have updated one or more of the same objects), a conflict resolution algorithm is invoked. There are currently three conflict resolution methods: older wins, newer wins, and application-specific. Many other resolution methods are possible. In the first method, the transaction with the oldest timestamp[2] "wins" and the other transaction is aborted. This is appropriate, for example, if the transaction represents a reservation. The second method is appropriate for updates that supersede old information, and so the newest transaction should win. The

[2] To implement this fairly, clocks must be synchronized using a protocol such as NTP [14].

application-specific method is passed the conflicting updates and determines whether one transaction should be aborted or a pair of conflicting objects can be replaced by an update semantically equivalent to the result of applying both updates. For example, if the object represents an unordered queue and each update adds a new item, the conflict could be resolved by creating a new version of the object that merges both updates since the operation is commutative. In cases where one of the transactions is aborted, it can be logged so that the initiator can re-execute it if desired. The appropriate resolution method is selected based on the type of the transaction, which is explicitly stated at the time it is started.

Replicas are autonomous in deciding the outcome of a transaction; no central authority needs to be consulted. The version vector, together with local information, provides sufficient information to determine whether a transaction should be committed or aborted, whether transactions have been missed, where a missed transaction can be obtained, and when resources can be freed. All replicas will eventually agree on the final outcome of each transaction and, provided communication between replicas is restored for a sufficiently long period, all replicas will converge on identical object values.

2.4 Temporary Inconsistencies

Since a transaction may not be processed at all replicas simultaneously, users throughout the system may observe different replies to identical queries. We believe that, for many applications, users would prefer to receive old information rather than nothing at all.

With the proposed transaction distribution scheme, however, allowing objects to contain the `ObjOid`s of other objects can create problems related to consistency. Because transactions can arrive at a replica in an arbitrary order, it is possible for it to receive an object that contains references to other objects that do not yet exist locally. Another type of inconsistency arises when a reference points to an object that is expected to be more recent than the version actually stored at the replica. We do not consider this latter form of inconsistency, since we believe that the kinds of applications where only write-write conflicts are important will not encounter it. Also, existing mechanisms can work around it when necessary, albeit inefficiently.

We have considered three approaches to the problem of missing objects. The first solution simply ignores the problem and applies transactions as they are received. When a transaction tries to read the missing object it is aborted. Another solution also applies transactions as they are received, but involves handling the reference to a missing object like a page fault, trying to contact another replica that has seen a transaction that contains the object. The third and most complicated solution is to introduce strict transaction ordering (causality) when necessary; i.e., a new transaction can name one or more other transactions that must have been previously applied. A transaction depending on earlier ones may now need to be queued before it can be applied. Also, cascading aborts must be implemented, since newer, dependent transactions must be aborted if a predecessor is.

2.5 Example Application

Because *ROSS* will primarily be used in conjunction with the EAN X.500 directory service [15], we will briefly describe some of the important operational aspects of *ROSS* in that context. Within the object store, X.500 objects are of a single type and are linked together (using `ObjOids`) to form a tree structure. Each object consists of a set of attribute/value pairs. The X.500 protocol supports three kinds of update operations:

adding an entry, updating an entry, and removing an entry. Adding a directory entry involves creating a new object and modifying its parent entry to point to it. Updating an entry modifies the entry in question and sometimes the parent, while removing an entry requires deleting the entry and updating its parent. Within an X.500 transaction at most two objects will be in any write-set.

For transactions involving the X.500 object type, weak consistency is used together with an application-specific conflict resolution method. If two or more administrators update the same directory entry, those transactions doing modifications that can be merged (e.g., those modifying different attribute types of the entry) need not be aborted. Similarly, when two or more transactions add entries with the same parent entry to the directory tree, it will often be possible to merge the changes to the parent entry so that it is not necessary to abort any transactions [3]. In the case of delete-update conflicts, the deleting transaction is aborted. Two or more deletions of the same object do not conflict.

3 Related Work

Many approaches to the problem of consistency in partitioned networks have been proposed [6]. A few that are relevant either to relaxing consistency constraints in favour of increasing availability or to asynchronous propagation of transactions are mentioned here.

Allchin [1] discusses a suite of decentralized algorithms that do not achieve serial consistency but which provide high availability. Davidson's Optimistic Protocol [5] provides one copy serializability using a precedence graph to detect inconsistencies. Boaz [2] has proposed an approach that adds fault tolerance to the two phase commit protocol by having the writer send updates to some write quorum and then continue; the remaining replicas are updated "in the background". The Camelot distributed transaction facility [4] provides lazy transactions that do not provide durability until a subsequent non-lazy update transaction commits. Ladin *et al.* [12] describe a lazy approach to replication based on causal ordering.

In the context of replicated file systems, LOCUS [17], Coda [18, 11], and Ficus [10] are projects that use optimistic weak-consistency protocols.

Our approach to weak consistency is similar to that used by name services such as Xerox's Clearinghouse [16] and DEC's DNA Naming Service [13, 7], although neither support transactions or tailorable resolution policies and both are specific to a particular application.

OSCAR [8] is an architecture for a weak-consistency replication system for database management systems. It differs from the work presented here in that it operates on a per update rather than on a per transaction basis and replicas are less autonomous, depending on a fault tolerant mediation component.

4 Conclusions

ROSS features both strong and weak consistency, allowing it to be used in those situations that call for one-copy serializability as well as where availability and performance are paramount. *ROSS*'s weak consistency considers only write-write conflicts, a satisfactory approach for a common set of applications, such as name services. The object store supports atomic transactions and is application independent. The policy used to resolve conflicts between transactions can be application specific.

[3] The exception is when two or more entries with the same name are added; in this case all but one of the additions must be aborted.

A non-distributed version of *ROSS* is currently being used by the EAN X.500 name service and implementation of *ROSS*'s distributed features is underway. Future work includes an evaluation of our approach to weak consistency in the context of X.500 and other distributed applications.

About the authors

Gerald W. Neufeld is an associate professor in the Department of Computer Science at the University of British Columbia. Neufeld is the director of the Open Distributed Systems Group. His interests include computer communications, distributed applications, and distributed operating systems. He is currently working on the Raven project; an object-oriented distributed system. Neufeld received a Ph.D. in computer science from the University of Waterloo in 1987. He received a B.Sc. (Honours) and M.Sc. from the University of Manitoba.

Barry Brachman is a research associate and lecturer in the Department of Computer Science at the University of British Columbia. His research interests include distributed systems, computer communications, and operating systems. He has recently been involved in the design and implementation of the EAN X.500 Directory Service and distributed databases and object stores. Dr. Brachman received the B.Sc. Honours degree from the University of Regina in 1981, and the M.Sc. and Ph.D. degrees from the University of British Columbia in 1983 and 1989, respectively, all in Computer Science.

References

[1] J. E. Allchin. "A Suite of Robust Algorithms for Maintaining Replicated Data Using Weak Consistency Conditions", *Proc. Third IEEE Symp. on Reliability in Distributed Software and Database Systems*, Oct. 1983, pp. 47-56.

[2] Boaz Ben-Zvi. "Disconnected Actions: An Asynchronous Extension to a Nested Atomic Action System", MIT/LCS/TR-475, Jan. 1990.

[3] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. "Concurrency Control and Recovery in Database Systems", Addison-Wesley, 1987.

[4] Joshua Bloch. "Camelot and Avalon: A Distributed Transaction Facility", edited by J. Eppinger, L. Mummert, and A. Spector, Morgan Kaufmann, 1991, pp. 21-56.

[5] Susan B. Davidson. "Optimism and Consistency in Partitioned Distributed Database Systems", *ACM Trans. on Database Systems*, vol. 9, no. 3, (Sept. 1984), pp. 456-481.

[6] Susan B. Davidson, Hector Garcia-Molina, and Dale Skeen. "Consistency in Partitioned Networks", *Computing Surveys*, vol. 17, no. 3, (Sept. 1985), pp. 341-370.

[7] Digital Equipment Corporation. "DNA Naming Service Functional Specification Version 1.0.1", Nov. 1988.

[8] Alan Downing, Ira Greenberg, and Jon Peha. "OSCAR: A System for Weak-Consistency Replication", *Proc. First Workshop on the Management of Replicated Data*, Nov. 1990, pp. 26-30.

[9] Robert Gruber. "Optimistic Concurrency Control for Nested Distributed Transactions", MIT/LCS/TR-453, June 1989.

[10] Richard Guy and Gerald Popek. "Algorithms for Consistency in Optimistically Replicated File Systems", Technical Report CSD-910006, Dept. of Computer Science, UCLA, Mar. 1991.

[11] James J. Kistler and M. Satyanarayanan. "Disconnected Operation in the Coda File System", *ACM Trans. on Computer Systems*, vol. 10, no. 1, (Feb. 1992), pp. 3-25.

[12] Rivka Ladin, Barbara Liskov, Liuba Shrira, and Sanjay Ghemawat. "Providing High Availability Using Lazy Replication", *ACM Trans. on Computer Systems*, vol. 10, no. 4, (Nov. 1992), pp. 360-391.

[13] Butler Lampson. "Designing a Global Name Service", *Proc. of the 5th Annual ACM Symp. on Principles of Distributed Computing*, 1986, pp. 1-10.

[14] David Mills. "RFC 1119: Network Time Protocol (Version 2) Specification and Implementation", University of Delaware, Sept. 1989.

[15] Gerald Neufeld, Barry Brachman, Murray Goldberg, and Duncan Stickings. "The EAN X.500 Directory Service", *Journal of Internetworking Research and Experience*, Vol. 3, No. 2, (June 1992), pp. 55-82.

[16] Derek Oppen and Yogen Dalal. "The Clearinghouse: A Decentralized Agent for Locating Named Objects in a Distributed Environment", Xerox Office Products Division, Tech. Report OPD-T8103, Oct. 1981.

[17] D. Stott Parker, Jr., Gerald Popek, Gerard Rudisin, Allen Stoughton, Bruce Walker, Evelyn Walton, Johanna Chow, David Edwards, Stephen Kiser, and Charles Kline. "Detection of Mutual Inconsistency in Distributed Systems", *IEEE Trans. on Software Engineering*, vol. SE-9, no. 3, (May 1983), pp. 240-247.

[18] Mahadev Satyanarayanan, James Kistler, Puneet Kumar, Maria Okasaki, Ellen Siegel, and David Steere. "Coda: A Highly Available File System for a Distributed Workstation Environment", *IEEE Trans. on Computers*, vol. 39, no. 4, (April 1990), pp. 447-459.