Disconnected Objects: Reconciliation in a Nested Object Transaction Environment*

Peter Graham, Ken Barker and Ahmad Reza-Hadaegh
Advanced Database Systems Laboratory
Department of Computer Science, University of Manitoba
Winnipeg, Manitoba, Canada R3T 2N2
{pgraham,barker,hadaegh}@cs.umanitoba.ca

Abstract

A mechanism for handling inconsistencies created when replicated objects are updated during disconnected operations on a mobile host is described. By exploiting the encapsulation property of objects, analysis of object behaviours is employed to derive "reconciliation procedures" so conflicting concurrent updates made on different copies of objects may be rendered mutually consistent. The process of generating and applying reconciliation procedures is discussed.

Keywords: Mobile Computing, Disconnected Operation, Reconciliation, Nested Transactions.

1 Introduction

When mobile machines are forced to operate in disconnected mode (i.e. without radio or other connection to a base network [8, 9]) it is necessary to make copies of the objects required by the mobile host (MH). This permits processing to continue on the MH despite disconnection. Unfortunately, the availability of multiple copies of an object introduces the potential for inconsistency. In connected systems, this is managed by a distributed consistency protocol. When *replicas* exist on unconnected machines, however, such protocols are not possible. Either concurrent access must be precluded (which is not acceptable) or any inconsistencies that arise must be reconciled or rolled-back. Roll-backs are undesirable because the re-execution costs are formidable and partial effects must be obliterated to guarantee atomicity. An alternative mechanism to resolve inconsistencies is to use *object reconciliation*.

Multi-verson concurrency control in conventional databases [1] is well understood. The basic idea has been extended to object base systems [4] to

*This research was partially supported by the Natural Sciences and Engineering Research Council (NSERC) of Canada under Operating Grant OGP-0105566.

enhance concurrency in the execution of multiple transactions. By allowing different transactions to access different *versions* of an object concurrently it is possible to achieve greater concurrency. However, as in the case of disconnected operation, consistency between versions is not ensured so reconciliation of the multiple resulting object versions must be performed.

Object reconciliation is the process of taking the results of "uncontrolled" updates on copies of an object and producing a new object which correctly reflects the combination of those updates. Reconciliation complexity ranges from simple to very complex. The complexity of reconciliation is partly determined by the object operation's execution model. We assume the closed nested transaction model [7] applied to method invocations on objects.

2 The Execution Model

A *correctness criterion* must be selected by which to measure the reconciliation procedures. The correctness criterion used in this paper is conflict-serializability [2] but we define it functionally instead of using the traditional conflict graph technique. Thus, methods on objects are state modifying functions so: given an initial object state $S[O_i]$ and methods 'f' and 'g', on object O_i, the result of reconciling the concurrent executions of 'f' and 'g' on replicas of $S[O_i]$ is correct only if the resulting state is $S^{'}(O_i) = f(g(S[O_i]))$ or $S^{'}(O_i) = g(f(S[O_i]))$.

Conflict-serializability must apply between concurrently executing transactions [2]. Since methods can invoke other methods (possibly on other objects) a *nested* transaction system [7] is utilized [1]. Therefore, methods invoked on an object require executing a transaction. Should such a method invoke another method, the resulting execution is a *sub-transaction* of the first.

Two types of objects exist in our model. First, *network-based* objects are always "on the network" so they can be managed in traditional ways. Second, *replicas* are copies of the network-based object that are acquired prior to MH disconnection. Our model's execution paradigm states that when a new object is referenced, a replica is automatically created for the requesting MH. When the MH disconnects, concurrent updates to the locally cached replicas and network-based objects is allowed. When the MH reconnects to the network, any updated replicas are checked against their network counterparts and any required reconciliation is performed.

3 Generating Reconciliation Procedures

When a conflict occurs the MH and some site on the network both execute methods on replicas of the same object. The result of one of the method executions may be wrong since it may have read stale data instead of the

[1] Moss describes two forms of nested transactions – "open" and "closed" [7]. In this paper we deal strictly with closed nested transactions.

current data written to the other replica[2]. Traditional resolutions require a "rollback" mechanism for the transaction so it can be atomically re-executed. This is undesirable because some useful work may be lost (not all computations necessarily need to be rolled back) anmd this may substantially inconvenience users in interactive environments. Reconciling the two executions *automatically* avoids these drawbacks.

The required *reconciliation procedures* may be generated automatically using techniques based on those used in optimizing and parallelizing compilers [10, 6, 3]. A reconciliation procedure for each ordered pair of methods ('f' and 'g') defined on an object is an executable sequence of code which corrects errors made in the execution of 'f' and 'g' on replicas of an object assuming that the corresponding transactions T^f and T^g are serialized by $T^f \rightarrow T^g$.

Such procedures are generated by comparing the set of attributes written by 'f' to the set of attributes read by 'g'. If there is any intersection then 'g' has read stale data rather than the data (written by 'f') that it was supposed to. Any computations by 'g', which were based on the attributes written by 'f', must be re-executed. The corresponding reconciliation procedure consists of exactly these computations[3].

4 Applying Reconciliation

Inconsistencies are detected by maintaining a record of the method invocations made by each transaction (both on the MH and networked systems) while the MH is disconnected. When the MH reconnects, each object accessed is examined. If the object has been *updated* by the networked system and an MH, then reconciliation is required to eliminate inconsistency. Determining that a method updates an object's attributes is simple using static analysis of the method code.

Several factors influence reconciliation difficulty including whether single or multiple objects were accessed by the transaction(s) on the MH and whether a single or multiple network-based transactions updated objects accessed by the MH during disconnection. Transactions on the MH and network sites which concurrently access different replicas of a object are said to *conflict*.

4.1 Single Object Reconciliation

First consider the case where inconsistency occurs at a single object and only a single network-based transaction conflicts with the MH-based transaction accessing the object in question. Clearly consistency need not be guaranteed in commitment order across multiple objects since only a single object is updated.

Given a specified serialization order for the MH and network based transactions two forms of reconciliation may be required. These are "simple" and "complex" reconciliation [4].

[2]Based on the correctness criterion.

[3]A more detailed discussion of the generation of reconciliation procedures is available in [4]. Some of the issues arising will be discussed at the workshop

Assume that the two transactions in question are T^1 and T^2 and $T^1 \rightarrow T^2$ is the serialization order selected [4]. If T^2 does not read-from [2] T^1 then *simple* reconciliation may be used to propagate the updates of T^1 into the object state produced by T^2 (except where T^2 overwrites updates made in T^1). If T^2 reads one or more attributes written by T^1 then *complex* reconciliation is required. All computations performed in T^2 which depend on attributes written by T^1 are re-executed but other non-dependent operations can be ignored. This is exactly what is done by a reconciliation procedure.

4.2 Multiple Object Reconciliation

Reconciliation must also handle the case where multiple objects have been accessed by the MH and network-connected sites concurrently. This results in conflicts at multiple objects. We begin by only considering a single network-connected transaction conflict *per object*.

Multiple object reconciliation is not as simple as applying single object reconciliation to each object because the serialization order between T^1 and T^2 must be consistent at all objects to ensure serializability. This implies that updates to all objects must be tracked so no *transitive* violations of serializability are possible. Since transaction oriented environments normally log operations for recovery purposes, the reconciliation mechanisms may obtain the needed information from the redo and undo logs [5].

4.2.1 Changing the Serialization Order

Unnecessary reconciliation may occur when dealing with inconsistencies in serialization order at multiple objects. It is desirable to try to select a serialization order which minimizes the cost of reconciliation[5]. Unfortunately, an undesirable serialization order may be introduced at one object which requires expensive reconciliation at others. Therefore, *changing* the serialization order after the fact may avoid costly reconciliation. This is a recent insight that has been developed and is the subject of another developing paper.

4.3 Multiple Object Updates

A further complication occurs when committed updates by multiple transactions are made to an object between the time the MH's replica is taken and the time reconnection occurs. In this case, the replica updated by the MH is based on the state of the object prior to the updates made by the network transactions. When this happens the likelihood of the MH transaction having read stale data (updated by the earlier committing network transactions) is high and therefore reconciliation is likely to be expensive. Effectively, what must be done is *multiple* reconciliations, one per committed network transaction.

[4] The serialization order may be selected based on order of commitment or to simplify the process of reconciliation.

[5] Coarse estimates of the cost of complex reconciliation may also be generated statically.

4.3.1 Serialization Order Revisited

The ability to change the serialization order of committed transactions is particularly desirable here. If the MH-based transaction can be inexpensively serialized prior to the already committed network-based transactions then the reconciliation overhead is avoided. Unfortunately, serializing it ahead of other transactions introduces the possibility re-reconciliation for those transactions.

5 Conclusions and Future Work

This paper discussed an approach to supporting disconnected operation in mobile object systems through the application of *automatically* generated reconciliation procedures. The application of these procedures is discussed and mechanisms sketched to limit the work required to perform reconciliation.

Future work involves changing the serialization order of committed transactions and the effects of doing so. An empirical investigation of reconciliation vs. simple rollback and re-execution is also needed. Finally, the effectiveness of reconciliation *vis a vis* the degree of object sharing between network-based and MH-based transactions during disconnected operation should be investigated.

References

[1] P.A. Bernstein and N. Goodman. Multiversion Concurrency Control – Theory and Algorithms. *ACM Transactions on Database Systems*, 8(4):465 – 483, 1983.

[2] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison Wesley, 1987.

[3] G. Goff, K. Kennedy, and C-W. Tseng. Practical Dependence Testing. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 15 – 29, June 1991.

[4] P.C.J. Graham and K.E. Barker. Effective Optimistic Concurrency Control in Multiversion Object Bases. In *International Symposium on Object Oriented Methodologies and Systems (ISOOMS) in Springer-Verlag LNCS #858*, pages 313–328, September 1994.

[5] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann Publishers, San Mateo, California, 1993.

[6] D.E. Maydan, J.L. Hennessy, and M.S. Lam. Efficient and Exact Data Dependence Analysis. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 1 – 14, June 1991.

[7] J.E.B. Moss. *Nested Transactions – An Approach to Reliable Distributed Computing.* The MIT Press, 1985.

[8] M. Satyanarayanan, J.J. Kistler, and L.B. Mummert. Experience with disconnected operation in a mobile computing environment. Technical Report CMU-CS-93-168, Dept. of Computer Science, Carnegie Mellon University, 1993.

[9] C.D. Tait and D. Duchamp. Service Interface and Replica Management Algorithm for Mobile File System Clients. In *First Intl. Conf. on Parallel and Distributed Information Systems*, pages 190–197. IEEE, 1991.

[10] Michael Wolfe and Utpal Banerjee. Data dependence and its application to parallel processing. *International Journal of Parallel Programming*, 16(2):137–178, 1987.