Disconnected Operation in the Thor Object-Oriented Database System

Robert Gruber Frans Kaashoek† Barbara Liskov Liuba Shrira

Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

Abstract

This paper discusses issues raised by providing disconnected operation in the Thor object-oriented database system. Disconnected operation in such a system poses new challenges because of the small size of objects, the richness and complexity of their interconnections, the huge number of them, and the fact that they are accessed within atomic transactions. We propose three techniques to address these challenges: (1) using the database query language for hoarding; (2) using dependent commits to tentatively commit transactions at the disconnected client; (3) using the high-level semantic of objects to avoid transaction aborts.

1 Introduction

Thor is a distributed object-oriented database (OODB) system that provides a persistent universe of typed, encapsulated objects. Computations take place within atomic transactions that typically make use of many objects. Applications interact with Thor by invoking object methods and requesting transaction commits.

The implementation of Thor is distributed. Persistent objects reside at server machines; applications run at client machines. Copies of persistent objects are cached on clients to reduce delay for applications and offload work from servers.

The Thor implementation exists (although it is still under development) but does not support mobile clients. This note discusses issues that must be addressed to support such clients. In particular, it looks at disconnected operation in Thor. For our discussion we will assume mobile clients are relatively powerful computers (*e.g.,* laptops). We believe a system like Thor raises new issues for disconnection because:

1. An OODB is structured differently from a file system. Objects are small compared to files: we expect an average object size of under 100 bytes. (This figure is based on analysis of object sizes in the OO7 benchmark [2].) Object interconnections are complicated and irregular, while file system name spaces are simple and highly structured. Finally, the number of objects managed by a database will be much larger than the number of files managed by a file system.

2. An OODB is accessed differently than a file system. Objects are accessed by navigation and queries; accesses will often be hard to predict, with later objects of interest determined by examination of earlier ones. In addition, computations run as atomic transactions, which provide both serializability and persistence of effect.

The remainder of this note discusses how these properties affect disconnected operation and proposes three techniques to address them: (1) when preparing for disconnection, use the database query language to hoard objects; (2) while in disconnected operation, use dependent commits

Authors' addresses: {gruber, kaashoek, liskov, liuba}@lcs.mit.edu.
MIT Laboratory of Computer Science, 545 Technology Square, Cambridge, MA 02139.

This work was supported in part by the Advanced Research Projects Agency under contracts N00014-91-J-4136 and N00014-94-1-0985. († This author is partially supported by an NSF National Young Investigator award.) The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. government.

to tentatively commit transactions; (3) use the high-level semantics of objects to increase the likelihood of commit in general and at reconnect. We argue that these ideas combined with Thor's distributed architecture will provide a viable approach to handling mobile clients in an OODB.

An important point of this paper is that existing concepts from databases can be very helpful: our three main proposals all involve a novel application of earlier database work to this new domain.

2 Thor

Thor and its implementation are described in [17]. This section gives a brief overview of those aspects of the system that are relevant for disconnected operation.

Thor provides highly-reliable and highly-available storage for persistent objects at server machines. There can be many servers; at a given time, each persistent object belongs to a particular server (objects can be migrated to a new server). Objects are clustered at servers, so that most references go to objects at the same server, but they can contain references to objects at other servers.

Users of Thor run at client machines. A portion of Thor called the FE (for "frontend") runs on each client machine. The FE maintains a cache containing copies of persistent objects; it carries out client requests and interacts with the servers only to handle cache misses (in which case the FE *fetches* the needed object from its server and prefetches a number of related objects [5]) and transaction commits.

Transactions are synchronized using optimistic concurrency control (see [1]); we provide some details here since they are relevant to disconnected operation. While a transaction is running, the FE keeps track of which persistent objects it uses (reads and writes). When the client requests a commit, the FE communicates with a server (one containing some used objects) that acts as coordinator of a 2-phase commit; servers where other used objects reside act as participants. Participants *validate* the transaction in phase 1; a transaction is validated provided (1) it has not read an old version of some object, and (2) it has not modified an object read by a transaction that would be serialized after it and that has already committed or prepared. If all participants validate the transaction, the coordinator commits it; otherwise the transaction aborts. The coordinator records its decision on stable storage and notifies the client; it informs participants of the decision in the background.

When a participant finds out about a commit, it installs new states for the modified objects (this does not require an immediate disk I/O [10, 21]) and sends *invalidation messages* to FEs that might contain copies of modified objects. An FE discards invalidated objects from its cache, aborts its current transaction if it used an invalidated object, and acknowledges the invalidation. Note that invalidation messages are just an optimization: they avoid aborts by removing invalid information from FE caches, and cause early aborts of transactions that could not commit. If invalidation messages are not delivered, the system can still run correctly, since persistent objects are modified only when transactions commit, and transactions that used invalid objects will abort.

Validation is carried out without the use of version numbers: objects do not contain any concurrency control state. This decision significantly reduces space overhead for our small objects; in essence, we record concurrency control information only for objects that might cause conflicts, while avoiding such information for the bulk of objects. Validation makes use of *FE-tables*, maintained by servers, that track objects cached at client FEs. (FE-tables are maintained at a coarse granularity and, assuming objects are well-clustered within a server, will not be very large. For example, a few thousand words should be sufficient to record information about an FE with 100,000 objects in its cache.) A server uses the FE-tables to determine what invalidation messages to send. It also marks invalidated objects in the table and records this information on stable storage (in the commit record for the committing transaction). Objects are unmarked on receipt of invalidation-acknowledgements from FEs. Validation of a transaction succeeds only if none of the objects it used are marked as invalidated in the table for its FE.

The Thor architecture is well-suited to disconnected client operation. A transaction that runs at one FE can fetch and commit without the need for servers to communicate with other FE's (as would be the case, *e.g.,* if a locking technique such as call-back locking [3] were in use). This is good since other FE's might be disconnected at the time. While it is true that lists of invalid objects will grow during disconnect, these lists will still be much smaller than the space required to keep concurrency control information for every object.

3 Disconnected Operation

We now consider the problems that would arise if the client machine were disconnected from the server machine. We assume a total disconnect in which the client is unable to communicate with the server; we discuss partial disconnect in Section 4.

3.1 Getting the Right Objects in the Cache

While a client is disconnected, it can continue to run provided needed objects are in its cache, but will be unable to make progress (at least in doing that particular task) if objects are missing. Therefore, the first issue is: how to ensure that the right objects are in the cache prior to disconnect. We propose to use "hoarding queries" for this purpose.

The Coda system augments the usual LRU cache policy with user-supplied "hoarding profiles" to ensure that the right files are in a user's client cache on disconnect [22]; without such a mechanism, users would be unable to work on anything but their most recently accessed files while disconnected. Coda's hoarding profile language is tailored to the naming structure of a file system (path names with wildcards are used to name files and/or entire subtrees to be prefetched). For an OODB, a richer language that can handle complex object interconnections is necessary. Descriptions of what to hoard should be written abstractly, in terms of object methods, rather than concretely; for example, we want to talk about the "document" of an atomic part (in the OO7 database) rather than its second pointer.

Fortunately, OODBs already have a good language for describing collections of objects, namely object-oriented query languages. These languages are similar to relational query languages, but have been extended to include object navigation via the traversal of representation-exposing *path expressions*, *e.g.,* `x.project.manager.salary`, or, preferably, the invocation of arbitrary methods or functions, *e.g.,* `x.manager_salary()`. Current commercial OODBs support efficient index-based associative access for path expressions [6, 7, 16, 18, 20], while future OODBs should support indexes over methods and functions [14]. Such query languages are a good way to describe hoarding profiles, supporting both simple requests (*e.g.,* prefetch all sections of the paper I am working on) and complex ones. It is likely that queries useful for hoarding will be formulated by users during normal interaction with Thor; these queries can be saved in the database, avoiding the need to reformulate them later.

In addition to using queries to prefetch objects into a client cache, the result of a query can be stored at the client as an independent *snapshot* of the current database state. A snapshot is a volatile *copy* of the object state returned by the query; once a snapshot is generated, objects in the snapshot have no connection to objects in the database: the snapshot does not track any further updates to the database, and furthermore updates to objects in the snapshot are not propagated back to the server, as happens with updates to cached database objects. We discuss why snapshots are important in the following sections.

Users may not know the full set of objects used by a particular application, or may not know how to formulate a query that describes these objects, making it impossible for them to write a hoarding query that ensures the application will be able to run while disconnected. To solve this problem it will probably be necessary to provide a *tracing tool* that captures the reference pattern that occurs over one or more runs of an application. The set of object references captured in this way can be stored and later used in a hoarding query. Such a tool has been proposed for object prefetching [4]; it is similar to the file tracing tool provided by Coda [22] that helps users determine which "hidden files" are used by an application so that they can be included in their hoarding profiles. However, an OODB trace can include not only object identifiers but descriptions of the queries used during the trace run. Such queries can be replayed as part of a later hoarding query, making it more likely that the right set of objects is prefetched.

3.2 Running While Disconnected

There are two main problems that arise while running disconnected: what to do if there is a cache miss, and what to do about transaction commits. We propose to make these problems visible to users via cache miss exceptions and tentative commits. Our guiding philosophy here is that a change in the semantics of an operation ought to be visible to the user, and the user ought to have control over how to proceed.

In the current Thor implementation, an FE communicates with a server when there is a miss and shuts down if communication fails. Clearly a different approach is needed for disconnected operation. Our solution is to have the method call (made by the client application) that led to the cache miss fail with an exception indicating that a cache miss was the problem. (All method calls to Thor can terminate with an exception.) The application can respond appropriately, *e.g.,* by aborting the current transaction and suspending with a message to the user indicating that it cannot proceed until a reconnect occurs. This would allow the user to start up another task in the meantime. The user should be able to request that the application be automatically resumed at reconnect time; this request could optionally include a hoarding query that should be run on reconnect before resuming the application.

Now let's consider transaction commit. During disconnected operation, a transaction that uses database objects cannot be committed, since this requires communication with Thor servers. Instead we provide a different kind of action, a "tentative commit", which records an intention to commit and allows the client to start up the next transaction. Having tentative commits leads to "dependent commits" [19]: transaction T2 depends on T1 if it uses objects modified by T1 because if T1 ultimately aborts, so must T2. While Thor avoids the possibility of such "cascaded aborts" during normal operation, it cannot do this for a sequence of tentative commits during disconnect.

To process a tentative commit, an FE must remember all objects read and written by the transaction; it uses this information to commit the transaction at reconnect, and also to compute dependencies. If the disconnect lasts a long time, the tentative commit log may grow very large. We present two approaches to controlling log size, which is important both for saving space at the client and for reducing the cost of processing the tentative commits on reconnect.

First, the log can be compacted by taking advantage of the fact that at reconnect time the tentative commits will be processed in a batch; they will all be serialized at the same point with respect to transactions that ran at other FEs. *E.g.,* if transaction T2 depends on T1, then for T2 we only log its additional reads and writes beyond what T1 did (plus a pointer to T1's log record).

Second, the use of snapshots can reduce the need for logging. Snapshots capture the state of the database at the point the snapshot was generated, and are independent of the current database state. Snapshot objects are volatile: accesses to volatile object are not tracked for concurrency control purposes and they are not logged in the tentative commit log. Transactions that use only volatile objects can be committed locally by the FE during disconnect; tentative commits are used only for transactions that read or write cached database objects. If database accesses are replaced with snapshot accesses, the size of the log can be significantly reduced.

Snapshots can only be used for cases where users can compute over data that is potentially out-of-date. For example, suppose a user snapshots the design of an automobile prior to disconnect. This snapshot can be used during disconnect to compute the total cost of the design, the number of assembly steps, *etc.* — the user, having explicitly generated the snapshot, is aware that these computations will not reflect updates to the design that occurred after the snapshot was generated. In some cases users know that an object will never be modified in the future, even though it has update methods; *e.g.,* a spreadsheet representing the last quarterly report is not updated after it has been finalized. Such objects are excellent candidates for snapshoting.

In the current Thor implementation, when an application attempts a commit, the result is either commit or abort. If tentative commits are supported, the commit operation must have an additional "tentative-commit" result. The application, on receiving the first such result, should ask the user whether it should proceed with additional transactions or should abort and suspend until reconnect (as in the cache miss case above), allowing it to execute the transaction as a normal transaction when it resumes. This is in line with our philosophy that different semantics, such as the uncertainty of a tentative commit, must be exposed to users, and users must be able to control an application's behavior for such cases. (Determining the user's instructions for the cache miss or tentative commit cases need not be interactive; there could be a user environment that applications consult to determine what action to take. Our point is that users must ultimately have control over the behavior of applications during disconnected operation.)

We should also mention here that tentative commits may be useful even when a client is fully connected. For some types of transactions, *e.g.,* where the user is very confident that a commit will occur, it makes sense to allow a transaction to do a tentative commit and proceed immediately to executing the next transaction. In this case a tentative commit record is logged as before, while a commit request is sent right away (in the background); once the commit outcome is known, the tentative commit record can be truncated from the tentative commit log. (Failed tentative commits would be handled as described for the case of failure on reconnect.)

3.3 Reconnect

When the client machine reconnects, the FE attempts to really commit all the tentatively committed transactions. This leads to two questions: how to make commit as likely as possible, and what to do if you cannot commit. We propose to make use of high-level semantics of objects to reduce aborts, and to provide replay tools when aborts happen.

There are a number of ways to increase the probability of commit. First, as described above, snapshots reduce the read and write sets of a transaction, thus making commit more likely.

Second, read-write sharing can be controlled by the application, or by users themselves, so that conflicts are avoided. For example, co-authors of a paper often agree in advance about who is working on what sections of a paper; in this case some mutual exclusion knowledge is maintained outside the system. Mutual exclusion information can also be stored within the system, *e.g.,* a co-authoring system might support the recording of authors assigned to sections, and only allow the correct author to edit a section. Such application-level locks would be held during disconnect periods, ensuring that an author's updates could be committed once the author's machine is reconnected. (Such applications often have a natural "check-out/check-in" model, where a lock is acquired/released by these actions. To avoid permanent lockup due to a check-in that never has a corresponding check-out, the application should of course include lock timeouts or a means of explicitly releasing locks; thus the client actually holds a lease on the lock [12].)

A third way to increase the likelihood of commit is to take advantage of object semantics to increase concurrency, by applying *type-specific concurrency control* [8, 23] instead of simple read-write locking . Herlihy [13] describes type-specific optimistic concurrency control, which is the approach we would use for Thor objects. Briefly, pairwise conflict checks are specified for each pair of methods of a given type. Because validation is performed after these methods have been executed, the actual arguments and results can be used in these conflict checks. For example, storing into the first element of an array does not conflict with reading the second element; as a second example, if a transaction fetched the second element of an array and obtained the result 33, this would not conflict with an earlier transaction that stored 33 in the second element.

Applying such an approach has a cost, however. Rather than logging reads and writes in the tentative commit log, one must log method invocations, possibly with argument or result values (if these are used in the specified conflict checks). As a result, tentative commit logs would be larger and validation on reconnect would take longer.

In spite of these additional costs, the type-specific approach is clearly valuable for certain data types, as it can considerably reduce the number of aborts. Users could control which data types (or even which objects of which data types) use type-specific concurrency control, limiting its use to those objects that often fail simple read-write conflict checking.

File systems actually provide a strong case for the utility of selectively applying this approach. Note that systems such as Coda actually use optimistic type-specific concurrency control, but only for a single data type: the directory type. The semantics of directory creation and deletion, and of different kinds of directory updates, are all taken into account when a Coda client reconnects. For example, adding a file to a directory does not conflict with removing another file from the same directory. To support this kind of fine-grained directory-specific conflict checking, Coda logs each directory operation separately, and sends the log to servers on reconnect, where they are compared with the logged directory updates of committed transactions. This is analogous to the work that Thor would have to perform for other data types that did this kind of conflict checking.

For an OODB, instead of just providing this kind of fine-grained conflict checking for a single built-in type, we can allow users to specify appropriate conflict checks for any of the abstract types that they define. The type system of an OODB gives us considerable flexibility compared to a typeless file system and a much more principled approach.

In spite of the above techniques, there will still be cases where a tentatively-committed transaction ends up aborting. When this happens, all transactions that depend on the aborted transaction must also abort, although non-dependent transactions may still be able to commit. Since the effects of aborted transactions are likely to be important to users, tools to help users recover are needed. To help the user recover requires saving context and allowing replay. There has been work in the database area on the recovery problem [9, 11]. (There has also been work on supporting recovery for write-write file conflicts in file systems [15].) In the best case, the system can simply re-run the transaction in the new state, but this works only if no user input is required. So the best general approach is to start up the replay at the earliest abort, with information about what happened, and let the user and/or application take it from there.

4 Partial Disconnect

In the future, clients probably won't be completely disconnected from servers but merely "less" connected, where less connected means: lower communication bandwidth, higher delay, more cost, intermittent inability to connect.

Since communication will be poorer during the less-connected time, it will still be good to prepare for such a period by preloading the client cache with the right information. When there is a cache miss, fetching the missing information is now possible, but is probably best done under user control, for two reasons: the fetch may take a long time, and the cache miss may indicate that there are many missing items, so that it is better to handle the problem at a higher level (*e.g.,* by doing a query to fetch the group).

However, the client should get better service because of the partial connection. For example, it might be possible to commit transactions, rather than tentatively committing them, or at least to commit groups of them periodically. This not only increases the probability of commit, but reduces information storage at FEs. On the other hand, committing will still have higher latency than before, so that tentative commits, along with techniques that increase the probability of commit, will still be important. In addition, FEs should reconnect periodically to keep cached information current.

5 Conclusions

This paper has discussed issues that arise in providing disconnected and partially connected operation in an object-oriented database like Thor. Our conclusions are:

1. The implementation of Thor is well-suited to this mode of operation. In particular, the use of client caching and optimistic concurrency control work well under these conditions.

2. Writing queries in an object-oriented query language is a good way to describe what to bring into a cache before disconnect.

3. Different semantics must be supported during a disconnect, and users need to be aware of this. In particular, only tentative commits are possible during a disconnect, and users should determine whether they ought to be used.

4. Techniques for increasing the probability of transaction commit will be important to make reconnect work well. These techniques are similar to those used in file systems for resolving conflicts, but can be approached more methodically when tied to abstract types.

5. Techniques used to provide good service when there is complete disconnect are also useful when there is partial disconnect, but users can expect better service in such a system because of the extra communication.

References

[1] A. Adya, R. Gruber, B. Liskov, and U. Maheshwari. Efficient Optimistic Concurrency Control Using Loosely Synchronized Clocks. *Submitted to the 1995 ACM SIGMOD Int. Conf. on Mgmt. of Data*, May 1995. Available as Programming Methodology Group Memo 85, MIT Laboratory for Computer Science.

[2] M. Carey, D. DeWitt, and J. Naughton. The OO7 Benchmark. In *Proc. 1993 ACM SIGMOD Int. Conf. on Mgmt. of Data*, pages 12–21, Washington, DC, May 1993.

[3] M. Carey, M. Franklin, M. Livny, and E. Shekita. Data Caching Tradeoffs in Client-Server DBMS Architectures. In *Proc. 1991 ACM SIGMOD Int. Conf. on Mgmt. of Data*, Denver, CO, June 1991.

[4] M. Day. Object Groups May Be Better Than Pages. In *Proc. 4th Workshop on Workstation Operating Systems*, pages 119–122, 1993.

[5] M. Day. *Client Cache Management in a Distributed Object Database*. PhD thesis, Massachusetts Institute of Technology, 1994 *in preparation*.

[6] O. Deux et al. The Story of O_2. *IEEE Trans. on Knowledge and Data Engineering*, 2(1):91–108, March 1990.

[7] O. Deux et al. The O_2 System. *Communications of the ACM*, 34(10):34–48, October 1991.

[8] A. Fekete, N. Lynch, M. Merritt, and W. Weihl. Commutativity-Based Locking for Nested Transactions. *Journal of Computer and System Sciences*, 41(1):65–156, August 1990.

[9] H. Garcia-Molina. Using Semantic Knowledge for Transaction Processing in a Distributed Database. *ACM TODS*, 8(2):186–213, June 1983.

[10] S. Ghemawat. *Disk Management for Object-Oriented Databases*. PhD thesis, Massachusetts Institute of Technology, 1995 *in preparation*.

[11] D. Gifford and J. Donahue. Coordinating Independent Atomic Actions. In *Proc. of IEEE CompCon85*, pages 92–95, February 1985.

[12] C. Gray and D. Cheriton. Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. In *Proc. 12th ACM Symposium on Operating Systems Principles*, pages 202–210, 1989.

[13] M. Herlihy. Apologizing Versus Asking Permission: Optimistic Concurrency Control for Abstract Data Types. *ACM Trans. on Database Systems*, 15(1):96–124, March 1990.

[14] D. Hwang and B. Liskov. Index Maintenance for Object-Oriented Databases Using Semantic Information. *Submitted to the 1995 ACM SIGMOD Int. Conf. on Mgmt. of Data*, May 1995. Available as Programming Methodology Group Memo 86, MIT Laboratory for Computer Science.

[15] P. Kumar and M. Satyanarayanan. Supporting Application Specific Resolution in an Optimistically Replicated File System. In *Proc. 14th Workshop on Workstation Operating Systems*, pages 66–70, October 1993.

[16] C. Lamb et al. The ObjectStore Database System. *Communications of the ACM*, 34(10):50–63, October 1991.

[17] B. Liskov, M. Day, and L. Shrira. Distributed Object Management in Thor. In T. Özsua et al., editors, *Distributed Object Management*, pages 79–91. Morgan Kaufmann, San Mateo, CA, 1994.

[18] D. Maier and J. Stein. Indexing in an Object-Oriented DBMS. In *Proc. 1986 Int. Workshop on Object-Oriented Database Systems*, pages 171–182, Pacific Grove, CA, September 1986.

[19] W. Montgomery. Robust Concurrency Control for Distributed Information System. Technical Report MIT/LCS/TR-207, MIT Laboratory for Computer Science, Cambridge MA, December 1978.

[20] J. Orenstein et al. Query Processing in The ObjectStore Database System. In *Proc. 1991 ACM SIGMOD Int. Conf. on Mgmt. of Data*, pages 171–182, San Diego, CA, June 1992.

[21] J. O'Toole and L. Shrira. Opportunistic Log: Efficient Installation Reads in a Reliable Object Server. In *1rst Symposium on Operating Systems Design and Implementation*, Monterey, CA, November 1994.

[22] M. Satyanarayanan, J. Kistler, L. Mummert, M. Ebling, P. Kumar, and Q. Lu. Experience with Disconnected Operation in a Mobile Computing Environment. In *Proc. 1993 Usenix Symposium on Mobile and Location Independent Computing*, pages 11–28, Cambridge, MA, August 1993.

[23] W. Weihl and B. Liskov. Implementation of Resilient, Atomic Data Types. *ACM Trans. on Programming Languages and Systems*, 7(2):244–269, April 1985.