Improving Data Consistency in Mobile Computing Using Isolation-Only Transactions

Qi Lu M. Satyanarayanan

March 1995
CMU-CS-95-126

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

To appear in the proceedings of the 5th Workshop on Hot Topics in Operating Systems, May 4-5 1995, Orcas Island WA

Abstract

Disconnected operation is an important technique for providing mobile access to shared data in distributed file systems. However, data inconsistency resulting from partitioned sharing remains a serious concern. This paper presents a new mechanism called isolation-only transaction(IOT) that uses serializability constraints to automatically detect read/write conflicts. The IOT consistency model provides a set of options for automatic and manual conflict resolution. In addition, application specific knowledge can be incorporated to detect and resolve conflicts. To preserve upward Unix compatibility, the IOT mechanism is provided as an optional file system facility and its flexible interfaces allow any existing Unix application to be executed as an IOT. This paper describes high level system design and implementation and concludes with related work and current status.

This research was supported by the Air Force Materiel Command (AFMC) and ARPA under contract number F196828-93-C-0193. Additional support was provided by the IBM Corporation, Digital Equipment Corporation, and Intel Corporation.

Keywords: distributed file systems, mobile computing, disconnected operation, data consistency, transactions, isolation-only transactions, Coda

1 Introduction

Disconnected operation based on an optimistic replica control strategy has proved to be a viable technique for mobile file access[9]. There is now considerable evidence that *write/write conflicts,* which dominated early discussions of optimistic replication, are relatively rare and can often be transparently resolved[10, 11, 12, 18]. But actual experience indicates that *read/write conflicts,* traditionally ignored in file systems, deserve much greater attention in disconnected operation.

The possibility of read/write conflicts exists even in timesharing file systems: for example, consider the installation of a new version of a library by one user while another user is building executables that link in the library. Although such conflicts are rare in timesharing or workstation environments, the analogous scenario in mobile computing is far more likely to lead to read/write conflicts. Two factors account for this. First, long periods of disconnection significantly enlarge the window of vulnerability. Second, explicit user coordination to avoid conflicts is more difficult.

Consider the following example of a partitioned read/write conflict. A programmer Joe caches relevant files on his laptop for a weekend trip. While disconnected, he edits some source files and builds a new version of `cfs`, a file system utility program. But one of the libraries `libutil.a` that is linked in is updated on the servers during Joe's absence. Here the linking and updating of `libutil.a` constitute a read/write conflict, leaving `cfs` in a possible unsatisfactory state. The consequences of undetected read/write conflicts can be especially painful if the result `cfs` program is used for further mutations, thereby leading to cascaded read/write conflicts.

In this paper, we describe a mechanism that can detect and resolve read/write conflicts in the Coda File System[19]. The challenge in designing such a mechanism is to balance three distinct concerns. First, the mechanism has to offer improved consistency for applications in a convenient and easy to use fashion. Second, the mechanism has to be efficient and make minimal demands of resource-poor mobile clients. Third, it should be upward compatible with the large body of existing Unix software.

2 Isolation-Only Transactions

Our design extends Coda with a new transaction service called *isolation-only transaction(IOT).* IOTs are a sequence of file access operations with a set of properties specially tailored for disconnected operation in a mobile computing environment. The IOT system performs automatic read/write conflict detection based on certain serializability constraints. It supports a variety of conflict resolution mechanisms including employing application semantics for conflict detection and resolution. The name "IOT" stems from the fact that this mechanism focuses solely on the *Isolation* aspect of the ACID properties[8]. Unlike full-fledged transactions, IOTs do not guarantee *failure atomicity* and only conditionally guarantee *permanence*[15].

It is important to note that we do not replace Coda's underlying Unix file system semantics with a transactional one. Rather, to preserve upward Unix compatibility, IOTs are provided as an optional facility for helping users to maintain consistency in mobile computing. Any existing Unix application can be executed unchanged within the scope of a transaction. Note that in the rest of this document we will use the term *transaction* to mean IOT when there is no ambiguity.

2.1 IOT Interfaces

The IOT service in Coda can be easily accessed through either of the two separate interfaces we have developed. Users can dynamically specify which application to execute as a transaction using the *interactive interface* of a special C shell in a similar fashion to setting environment variables. For example the following command:

```
setiot /usr/cs/bin/make reexec
```

specifies *make* as a transaction using the default consistency guarantee and automatic re-execution as its conflict resolution option(explained later). Any execution of *make* will now obtain the specified consistency support from the IOT system. In addition, a sequence of commands bracketed by the commands of `beginiot` and `endiot` is treated as a single transaction.

Application programmers can construct a transaction using the `begin_iot(iot_spec)` and `end_iot()` library routines of the IOT *programming interface*, where `iot_spec` specifies the transaction's selection of consistency guarantee and conflict resolution option. The necessary source code adaptation is well-structured and straightforward.

2.2 Execution Model

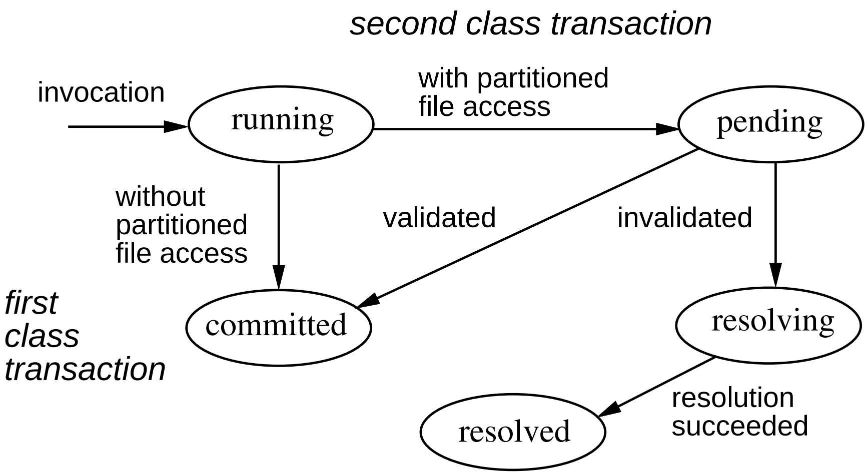


Figure 1: IOT State Transition

Transaction execution is performed entirely on the client and no partial result is visible on the servers. If a transaction, T, does not have any partitioned file access, its result is immediately *committed* to the servers and it is called a *first class* transaction. Otherwise, T is called a *second class* transaction and its result is held within the client's local cache and visible only to subsequent accesses on the same client. T stays in the *pending* state until connectivity is restored. At that time, T is validated with respect to the current server state using the consistency criteria described in the next section. T is immediately committed if it is successfully validated. Otherwise, it enters the *resolving* state to be automatically or manually resolved and then enters the *resolved* state(see Figure 1).

2.3 Consistency Guarantees

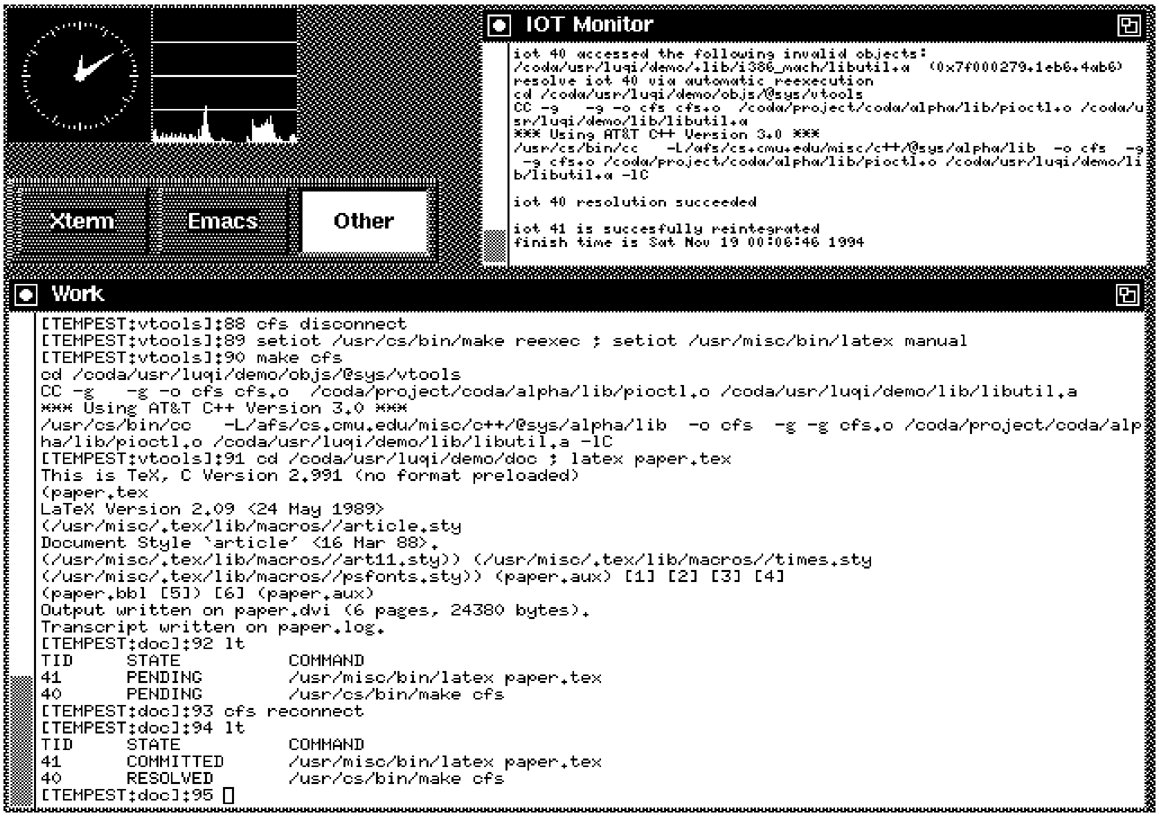
Similar to the traditional transactions, a first class transaction is guaranteed to be *serializable(SR)* with all previously committed or resolved transactions[7]. This guarantee enables IOT to serve as a general purpose file system concurrency control mechanism in fully connected environments.

Second class transactions are guaranteed to be locally serializable among themselves. Furthermore, two different serializability constraints are offered with one of them being automatically enforced as the consistency criterion for validating pending transactions based on the user's choice.

The first criterion is *global serializability*(GSR), which means that if a pending transaction's local result were written to the servers *as is*, it would be SR with all previously committed or resolved transactions[3]. However, GSR alone is not adequate for certain applications, particularly in voluntary and long lasting disconnected operation sessions.

Let us re-visit the previous example. Suppose Joe executes *make* as a second class transaction T_J to build `cfs` on his disconnected laptop and the library `libutil.a` is updated by a first class transaction T_L while Joe is away. Also suppose that there are no other related file accesses. When Joe re-connects the laptop to the servers, T_J will be admitted because it can be serialized before T_L, even though Joe might want his new `cfs` to be compatible with the latest libraries or at least be notified about the change.

Our remedy to this problem is to offer a stronger consistency criterion called *global certifiability*(GC) which requires a pending transaction be globally serializable not only *with*, but also *after*, all the previously committed or resolved transactions[15]. Intuitively, the GC criterion assures that the data accessed by a pending transaction are unchanged on the servers between the start and the validation of the transaction. We use GC as the default consistency criterion.



The work window displays the process of disconnecting the client, setting *make* and *latex* as transactions in the special IOT shell, executing a *make* transaction and a *latex* transaction, reconnecting the client, and checking transaction status using the *lt*(list transaction) command. At reconnection time the *make* transaction is invalidated because the linked library *libutil.a* was updated on the servers. The IOT monitor window shows the automatic re-execution of the *make* transaction and the committing of the *latex* transaction.

Figure 2: IOT Examples

2.4 Resolution Options

To cope with transaction validation failure, the IOT model provides the following four options.

- The first option is to automatically *re-execute* the transaction using up-to-date data from the servers. Consider our example one more time. Suppose Joe wants to make sure that his work done on a disconnected laptop is compatible with the most recent system state, he can run T_J using the default consistency guarantee and the automatic re-execution resolution option. When T_J is invalidated at the re-connection time because of T_L, the IOT system will automatically re-run *make* to build an up-to-date version of `cfs`. Part of Figure 2 shows the actual working of this example.

- The second option is to automatically *abort* the transaction by rolling back its local result. It is commonly used by traditional transactions for restoring consistency after failures. The first two options are applicable to almost any application because they are both automatic and application independent.

- The third option is to automatically invoke a user specified *application specific resolver(ASR)*[11, 12]. ASR is an important feature that distinguishes IOT from most other transaction models. It allows application semantic knowledge to be utilized for conflict resolution. More importantly, it provides support for application specific *conflict detection* as well. This is because failure to satisfy GSR or GC does not necessarily mean that the relevant objects are in an inconsistent state. An ASR can use application semantics to check whether a transaction's local result is actually consistent with respect to the current server state. If so, it can simply use the local result as the resolution outcome, effectively overcoming the limitations of syntactic validation of GC or GSR.

- As a last resort(the default option), the conflicts caused by an invalidated transaction are exposed to the user for *manual resolution*(or *repair*).

2.5 Conflict Representation

An object is *in conflict*(or *inconsistent*) if it is accessed by an invalidated transaction and its local version differs from its global version. The Coda client represents inconsistent objects in the form of a dangling symbolic link in normal operation to visually notify the users about the conflict and prevent further access and cascading inconsistency. However, in the *resolve operation mode*, where an invalidated transaction is being resolved, the client must expose both the local and global state for the relevant inconsistent objects to the resolver. To achieve this, an inconsistent object is temporarily converted into a *fake directory* with two children named "local" and "global" containing the object's local and global replica respectively. The local replica is read-only while the global replica is mutable and serves as the workspace for the resolver to build up resolution result. We also provide the *multiple view* capabilities so that the resolver can choose to view only the local or the global state of an inconsistent object in its normal form.

While an invalidated transaction T is being resolved, the local replicas of T's inconsistent objects correspond to their value at the completion of T's execution. In order to provide such a snapshot of T's resulting object state, we maintain multiple cache file versions and use dynamic binding between an object and its cache files. Notice that the resolution of T may need to access objects that were not previously accessed by T and some of them may be inconsistent due to other invalidated transactions. We employ the multiple view mechanism to ensure that only the global state of those objects is visible to the resolver so that global serializability can be obtained for the resolution process[14].

2.6 Conflict Resolution

Invalidated transactions are resolved incrementally, i.e., one by one, according to their local serialization order. Resolution computations consist of coordinated actions from the transaction system and the resolver. They are performed entirely on the client for security and scalability purposes. The system first sets up the resolution object view and the appropriate environment. Then the resolver(automatic or manual) takes over and creates the resolution result in the global replica of the corresponding objects. Finally the system atomically installs the resolution result to the servers, discards the original local result and restores the normal object view. The transaction system is also responsible for the local concurrency control necessary for the resolver's exclusive access to objects.

To facilitate the ASR programming, additional library routines are provided in the IOT programming interface to allow an ASR to adjust object views and test object membership for the corresponding transaction's readset and writeset. In addition, the Coda repair tool is extended with a set of new commands for manually repairing an individual or a group of transactions[14].

3 Implementation

Besides the IOT interfaces, the new Coda repair tool and some necessary extensions in the kernel and server to support transaction execution, most of the IOT system is implemented inside the *Venus* cache manager of Coda. Venus support for IOT resides in four main modules:

- The *execution monitor*'s main function is to record transaction readsets and writesets. The interface between the kernel and Venus is extended so that process information can be used to identify which file access operation belongs to which transaction.

- The *concurrency controller* performs two levels of concurrency control. Across clients, global concurrency control is maintained using the *optimistic concurrency control(OCC)* scheme[13]. Within a client, local concurrency control is enforced using strict two phase locking with periodic deadlock detection.

- The *replica manager*'s responsibility is maintaining and providing access to the local and global replicas for inconsistent objects. It is also in charge of maintaining multiple versions of a cache file and adjusting the binding between an object and its cache files.

- The *consistency maintainer* performs the central task of detecting and resolving read/write conflicts. Conflict detection is accomplished by checking whether a pending transaction's readset and writeset satisfy the appropriate constraints. Automatic conflict resolution support is provided by a generic invocation mechanism that can execute either the transaction itself or an associated ASR.

4 Status

As of this writing, most of the IOT system has been implemented and is operational. An early version of the replica manager together with an extended Coda repair tool has been released for public use for four months. An IOT system with all the basic functionality has been in private use for months. Examples of how transactions work in actual disconnected operation are shown in the screen image of a Coda laptop in Figure 2.

Currently, only the GC consistency guarantee is implemented because our experience indicates that it offers sufficient consistency support for the most common use of disconnected operation. Although GC may be too restrictive in certain situations compared to GSR, the ASR mechanism allows application knowledge to be utilized to compensate the limitation. Our current design is fully compatible with a future GSR implementation.

We are improving the current IOT implementation for public release to the Coda user community. Extensive experiments using IOT in application domains such as software development and document processing will be conducted. Based on quantitative measurements and qualitative usage analysis, we will evaluate IOT as a tool for improving data consistency in mobile computing.

5 Related Work

The IOT execution model is inspired by Kung and Robinson's optimistic concurrency control(OCC) model, with the client cache effectively serving as the private workspace for transaction processing. OCC is also employed as the concurrency control algorithm for first class transactions because it offers high performance and scalability in the Coda environment.

Lock based concurrency control methods[5], the main alternatives to OCC, are not used by IOT because of the difficulty of implementing distributed locking in the presence of disconnection. Timestamp based algorithms[2, 17] are not suitable for our purposes mainly because of the need to maintain timestamps for both read and write operations on the servers, which leads to significantly increased server load and reduced scalability.

The GSR consistency guarantee originates from Davidson's optimistic transaction model[3]. Application semantics have previously been used in transaction processing to improve concurrency control performance[6], whereas IOT's use of semantic knowledge focuses on application specific conflict detection and resolution.

Consistency maintenance for disconnected operation has also been the focus of some recent research. The Bayou architecture explores weak consistency in shared and replicated data repositories accessed by mobile hosts[4]. It allows individual applications to obtain a self-consistent view through *session* semantic guarantees[20], some of which can be achieved using IOT. Other recent work [16] discusses a weak consistency model for traditional transactions and [1] proposes the *causal consistency* technique for detecting mutual consistency for shared objects.

6 Conclusion

With the increasing popularity of portable computers and the use of mobile computing systems, the problem of maintaining consistency for partitioned data sharing will become more important. The key to solving this problem is to provide improved consistency within the limited resource capacity of mobile computers while maintaining usability for a wide variety of applications. We believe that the IOT mechanism is a significant step towards this goal.

References

[1] M. Ahamad, F. Torres-Rojas, R. Kordale, J. Singh, and S. Smith. Detecting Mutual Consistency of Shared Objects. In *Proceedings of the First IEEE Workshop on Mobile Computing Systems and Applications*, Santa Cruz, CA, December 1994.

[2] P. Bernstein, J. Rothnie, N. Goodman, and C. Papadimitriou. The Concurrency Control Mechanism of SDD-1: A System for Distributed Databases. *IEEE Transaction on Software Engineering*, SE-4(3):154–168, May 1978.

[3] Susan Davidson. *An Optimistic Protocol for Partitioned Distributed Database Systems*. PhD thesis, Princeton University, 1982.

[4] A. Demers, K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and B. Welch. The Bayou Architecture: Support for Data Sharing among Mobile Users. In *Proceedings of the First IEEE Workshop on Mobile Computing Systems and Applications*, Santa Cruz, CA, December 1994.

[5] K. Eswaran, J. Gray, R. Lorie, and I. Traiger. The Notion of Consistency and Predicate Locks in a Database System. *Communications of ACM*, 19(11):624–633, November 1976.

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

[7] J. Gray, R. Lorie, G. Putzulo, and I. Traiger. Granularity of Locks and Degrees of Consistency in a Shared Database. Research Report RJ1654, IBM, September 1975.

[8] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufman, 1993.

[9] J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. *ACM Transactions on Computer Systems*, 10(1):3–25, February 1992.

[10] P. Kumar and M. Satyanarayanan. Log-Based Directory Resolution in the Coda File System. In *Proceedings of the Second International Conference on Parallel and Distributed Information Systems*, January 1993.

[11] P. Kumar and M. Satyanarayanan. Flexible and Safe Resolution of File Conflicts. In *Proceedings of 1995 USENIX Conference*, New Orleans, LA, January 1995.

[12] Puneet Kumar. *Mitigating the Effects of Optimistic Replication in a Distributed File System*. PhD thesis, Carnegie Mellon University, December 1994.

[13] H.T. Kung and J. Robinson. On Optimistic Methods for Concurrency Control. *ACM Transaction on Database Systems*, 6(2):213–226, June 1981.

[14] Q. Lu and M. Satyanarayanan. Conflict Representation and Resolution for Disconnected Operation in the Coda File System. In Preparation.

[15] Q. Lu and M. Satyanarayanan. Isolation-Only Transactions for Mobile Computing. *ACM Operating Systems Review*, 28(2):81–87, April 1994.

[16] E. Pitoura and B. Bhargava. Revising Transaction Concepts for Mobile Computing. In *Proceedings of the First IEEE Workshop on Mobile Computing Systems and Applications*, Santa Cruz, CA, December 1994.

[17] D. Reed. Implementing Atomic Actions on Decentralized Data. *ACM Transaction on Computer Systems*, 1(1):3–23, February 1983.

[18] P. Reiher, J. Heidemann, D. Ratner, G. Skinner, and G. Popek. Resolving File Conflicts in the Ficus File System. In *USENIX Summer Conference Proceedings*, Boston, MA, June 1994.

[19] M. Satyanarayanan, J. Kistler, P. Kumar, M. Okasaki, E. Siegel, and D. Steere. Coda: A Highly Available File System for a Distributed Workstation Environment. *IEEE Transaction on Computers*, 20(4), April 1990.

[20] D. Terry, A. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. Welch. Session Guarantees for Weakly Consistent Replicated Data. In *Proceedings of International Conference on Parallel and Distributed Information Systems*, Austin, Texas, September 1994.