Isolation-Only Transactions for Mobile Computing

Qi Lu & M. Satyanaranyanan
School of Computer Science
Carnegie Mellon University

1. Motivation

The Unix File System(UFS) has historically offered a *shared-memory* consistency model. The lack of concurrency control makes this model susceptible to *read/write conflicts*, *i.e.*, unexpected read/write sharing between two different processes. For example, the update of a header file by one user while another user is performing a long-running `make` can cause inconsistencies in the compilation results. In practice, read/write conflicts are rare for two reasons. First, the window of vulnerability is relatively small because read/write conflicts only occur when the executions of two processes overlap. Second, they are often prevented via explicit user-level coordination.

However, the advent of *mobile computing* makes read/write conflicts a realistic threat to data integrity. Mobile computing is characterized by periods of *disconnection* and *intermittent connectivity*[11]. Such communication disturbances greatly widen the window of vulnerability from the life span of a process to the duration of a disconnection. They also significantly reduce the effectiveness of explicit user-level coordination, especially when disconnections are made transparent to users.

How can we preserve upward compatibility with the large body of existing Unix software, while offering improved consistency in a mobile computing environment? This position paper puts forth our solution, a new transaction model called *isolation only transaction* (IOT).

2. Background: The Coda File System

Providing portable computers convenient access to shared data in distributed Unix file systems is an important research topic in the fast growing field of mobile computing. The Coda file system addresses this issue by hiding mobility from applications and users[10]. It provides continuous file access to mobile clients even when they are disconnected from the servers. The key enabling technology is *disconnected operation*, a special form of client disk caching which employs optimistic replica control[4]. When disconnected, a Coda client services file access requests by relying solely on the contents of its local cache. Updates are performed locally, logged and later *reintegrated* to servers upon reconnection.

Coda demonstrated *optimistic replication* as a viable foundation for mobile clients to access shared files in the Unix workstation environment. However, data integrity in an optimistically replicated system becomes a serious concern because arbitrary partitioned file accesses are permitted. Two kinds of partitioned data sharing can result in inconsistency. The obvious problem is partitioned write/write sharing because it causes replicas to diverge. Fortunately, empirical evidence has shown that partitioned write sharing is very rare in a typical Unix workstation environment[4]. Furthermore, write/write conflicts can be efficiently detected[9]. In many situations, diverging replicas can be automatically reconciled using application-specific approaches[6, 5, 3]. The more subtle threat comes from partitioned read/write sharing.

Consider the following scenario of a partitioned read/write conflict. A programmer Joe caches relevant files on his Coda laptop for a weekend trip. While disconnected, he edits some source files and builds a new version of `repair`, a file resolution program. But one of the libraries `libresolve.a` that is linked in was updated on the servers during Joe's absence. Here the linking and the updating of `libresolve.a` constitute a partitioned read/write conflict, which not only leaves `repair` in a possible inconsistent state but also may cause cascading inconsistencies had Joe used this `repair` program to mutate other objects. It would be helpful if Joe is at least notified about the possible inconsistency when he reconnects the laptop to the servers.

Building upon well-known ideas in the database community, we are extending the Coda file system with an explicit isolation-only transaction service. This enables the system to only admit those partitioned read/write conflicts that satisfy certain serialization requirements. However, it is important to note that we are not trying to overhaul the current UFS semantics and replace it with a transactional one. It is simply impractical to re-write the numerous existing Unix applications. Instead, we regard upward Unix compatibility as one of our key constraints. IOT will be provided as an optional file system facility that application writers can use to selectively wrap-around applications for better consistency protection when used in mobile computing. Existing Unix applications are guaranteed to behave the same if they do not use IOT.

3. What is an IOT?

An IOT is a flat sequence of file access operations bracketed by `begin_iot` and `end_iot`. The execution of an IOT guarantees a set of properties that are specially tailored for optimistic replication and mobile Unix workstation environment. An IOT provides strong consistency guarantees depending on the system connectivity conditions. Unlike traditional transactions, it does not guarantee *failure atomicity* and only conditionally guarantees *permanence*.

The IOT execution model is inspired by Kung and Robinson's *optimistic concurrency control* model, with a client's local cache effectively serving as the private workspace for transaction[1] processing[7]. When a transaction T is invoked by a user, its entire execution is performed on the user's client machine. Remote files are accessed through the client's local disk cache; and no partial result of the execution is visible on the servers. When T's execution is completed, it enters either the *committed* or the *pending* state depending on the connectivity condition. If T's execution does not contain any partitioned file access (*i.e.*, the client machine maintains a server connection for every file T has accessed), T is committed and its result is made visible on the servers. Otherwise, T enters the *pending* state waiting to be validated later. T's result is temporarily held within the client's local cache and is visible only to subsequent processes on the same client. When the relevant partitions are healed, T is validated according to the consistency criteria to be discussed in section 5. If the validation succeeds, T's result will be immediately reintegrated and committed to the servers. Otherwise, T enters the *resolution* state. When T is automatically or manually resolved, it will commit the new result to the servers. Figure 1 shows the complete IOT execution model from the user's viewpoint.

[1] In the rest of this document we will use the term *transaction* to mean IOT when there is no ambiguity.

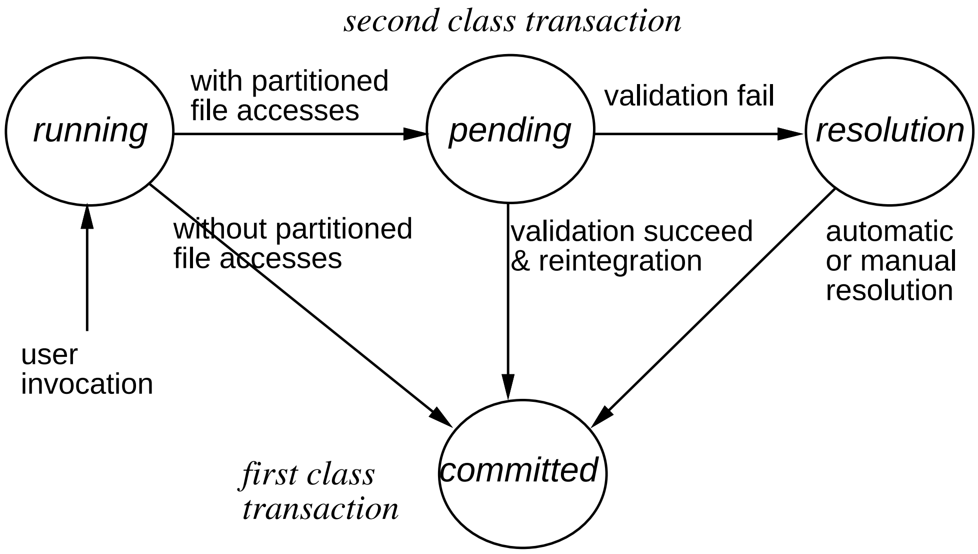

Figure 1: A State Transition Diagram for IOT Execution

4. Why Isolation Only?

Lightweight operation and high efficiency are our key design goals. As a result, the IOT model does not provide the failure atomicity and permanence guarantees present in the traditional transaction model.

Failure atomicity is not supported mainly because of high resource cost. A large amount of space is needed for undoing the effect of a transaction because it can access large objects and its execution can last long. Such cost is further magnified because space is a much more precious resource on mobile clients. Devoting too much space to the possible task of backing out transactions may cause denial of other valuable disconnected file services. Moreover, recent research has shown that the *all-or-nothing* property is not always desirable[8].

The permanence guarantee of the traditional transaction model promises that once a transaction commits, its result will stay unchanged and can survive various system failures. One of the key and often unnoticed consequences of this property is that once a transaction makes its result visible to subsequent transactions, the result must not change until it is modified by some other transactions. In the IOT model, the result of a pending transaction is visible to subsequent transactions running on the same client. But this result is subject to change upon future validation. Therefore we can only offer a conditional form of the permanence guarantee. That is, the result of a transaction is permanent only when it does not contain partitioned file access, or it is successfully reintegrated or resolved.

5. IOT Consistency Guarantees

In order to maintain data consistency, the traditional transaction model provides the isolation property to make sure that transactions are executed as if they were isolated from each other. In *serializability theory* terms, the isolation property guarantees that the results of the interleaved execution of a set of transactions are equivalent to some serial execution of the same set of transactions[2].

The IOT model offers substantially stronger consistency guarantees than the traditional transaction model. Transactions are classified into two categories: a *first class transaction* is one whose execution does not contain any partitioned file accesses. Otherwise, it is a *second class transaction* (see Figure 1).

- **Serializability(SR) for First Class Transactions**

  The execution of any first class transaction is guaranteed to be serializable with all committed transactions.

- **Local Serializability(LSR) for Second Class Transactions**

  The execution of any second class transaction is guaranteed to be serializable with other second class transactions executed on the same client.

- **Global Serializability(GSR) for Second Class Transactions**

  One of the consistency criteria for validating a pending transaction T is that T must be globally serializable(GSR) with all committed transactions. It means that if T's result in the client's local cache were reintegrated to the servers as is, T would be SR with all committed transactions.

  GSR is significantly different from SR or LSR in that GSR can not be enforced at transaction execution time. It can only be tested when the relevant partitions are healed. Therefore, as an integral part of the GSR guarantee we must specify what the system will do if the test fails. The IOT model provides the following automatic resolution options.

  - **Re-executing the transaction.**
    Successful re-execution of the transaction using the up-to-date server files is guaranteed to resolve the related inconsistencies. For example, this option can be used to automatically re-run `make` when the compilation results are inconsistent. It is our default option.
  - **Invoking the transaction's application specific resolver(ASR).**
    Sometimes it is more effective to resolve a transaction by using application-specific knowledge. The IOT model allows the transaction writer to attach an ASR to a transaction to be automatically invoked by the system. For example, non-serializable updates to an appointment calendar file can often be merged by an ASR as long as there are no time slot conflicts.
  - **Aborting the transaction.**
    Simply aborting a non-GSR transaction will suffice to restore consistency. Suppose a transaction is executed on a disconnected client to compress a large file while the same task has already been done by someone else on the servers, aborting the transaction is an appropriate action in such a situation.
  - **Notifying the users.**
    As a last resort, users can choose to manually resolve a non-GSR transaction. The IOT system will only mark its write-set as inaccessible and notify the users. If a transaction is used for editing the files of a co-authored paper on a disconnected laptop, this option is useful for coordinating possible concurrent updates.

- **Global Certification Order(GCO) for Second Class Transactions**

  In certain situations, GSR alone is not adequate for voluntarily disconnected mobile clients. In the earlier example, suppose Joe ran `make` as a second class transaction T_J to build the new version of `repair`; and the library `libresolve.a` is updated by a first class transaction T_L; and there are no other related file accesses. When Joe re-connects his Coda laptop to the servers, T_J will be admitted because it can be serialized before T_L.

To remedy this problem, we adopt a stronger consistency criterion called global certification order(GCO). GCO requires a pending transaction to be serializable not only *with* but also *after* all the committed transactions. GCO has the same set of resolution options as GSR.

If Joe wants to make sure that his work done on an isolated laptop is compatible with the most recent system state, he can select GCO as the consistency criterion for transaction validation. Now T_J will be rejected because it can not be serialized after T_L. Joe can also use the default resolution option to let the system automatically re-run `make` to build an up-to-date version of `repair`.

6. Implementation Strategy

Because of the need for partitioned transaction execution, logging is the foundation for IOT implementation in Coda. Based on the properties of the Coda mobile computing environment, we choose and extend the transaction implementation technologies that offer the best engineering trade-off. For brevity, we only highlight the following issues that are critical to the overall transaction processing performance.

- **Concurrency Control for First Class Transactions**

  We chose the *optimistic concurrency control*(OCC) method to enforce SR for first class transactions[7]. The main idea of OCC is trading transaction re-execution for global synchronization. This fits well with the scalable Coda architecture where client cycles are considered cheaper than server communication bandwidth. OCC is also capable of providing high throughput in the Coda environment because of low data contention. We extended the OCC scheme so that transaction history information can be utilized to process long-running transactions more efficiently.

- **Transaction Validation for GSR**

  We apply Davidson's *optimistic transaction model* for GSR testing[1]. This method builds a data structure called the *precedence graph* to represent the inter-dependency among transactions across partitions. GSR testing then becomes a matter of cycle detection in the corresponding precedence graph. We also extended the model so that GSR testing can be performed even when the transaction histories are truncated.

- **Transaction Logging**

  Continuous transaction logging is needed on both servers and clients. Because log space is finite, transaction service will be unavailable to a client if its log space is exhausted. However, a server will truncate its recorded transaction history to reuse log space. Based on our preliminary observations and estimation, a modest amount of server log space(*e.g.* 40MB/server) may suffice for a typical working day. Fortunately, the GCO testing can be performed without using transaction histories.

7. Evaluation Plan

When the IOT system is fully implemented, it will be put to actual daily use by a number of Coda users. Our experiments will focus on the *software development* domain where read/write sharing is believed to be frequent. Important applications such as `make`, `cc` and `latex` *etc.* will be modified to use IOT.

Quantitatively, we will measure the various aspects of transaction processing performance such as:

- What is the performance overhead for file access operations due to IOT processing?
- How much space is adequate for transaction logging on servers and mobile clients?
- How efficient are transaction validation and resolution?

With the availability of an explicit file system transaction service, we also expect to collect interesting data such as:

- How often read/write conflicts occur during a typical session of disconnected operations on a mobile client.
- What fraction of them lead to non-GSR or non-GCO transaction executions.

This new information will allow us to gain more insightful knowledge about the nature of mobile computing as well as the data sharing patterns in software development activities.

Qualitatively, our main effort will be to characterize the IOT usage properties. Based on the actual experiences of the participating Coda users, we will study important issues such as:

- Can the IOT abstraction be conveniently incorporated into existing Unix applications?
- Do GSR and GCO provide sufficient consistency support for normal mobile computing practice?
- How often are the different resolution options used and how useful are they?

8. Current Status

We are in the initial process of building a working implementation of an IOT service on the Coda file system running Mach 2.6. A detailed architectural and algorithmic design including key algorithms, protocols and data structures has been developed. Progress has been made on Mach kernel changes, Coda client/server interface extensions and the addition of a *pseudo-disconnected operation mode* which supports transaction processing. As of this writing, a simple transaction manager with primitive functionality is operational.

9. Acknowledgements

We wish to thank Jay Kistler for his valuable help in formulating the initial IOT framework. We would also like to thank members of the Coda group, especially Puneet Kumar, Maria Ebling, Lily Mummert, David Steere, Brian Noble and Josh Raiff for helpful discussions and assistance.

This research is supported by the Advanced Research Projects Agency (Hanscom Air Force Base under Contract F19628-93-C-0193, ARPA Order No. A700), the IBM Corporation, Digital Equipment Corporation, the Intel Corporation, and Bellcore. The views and conclusion expressed in this paper are those of the author, and should not be interpreted as those of the funding organizations or Carnegie Mellon University.

References

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

[2] 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.

[3] R. Guy and G. Popek. Reconciling Partially Replicated Name Spaces. Technical Report CSD-900010, University of California, Los Angeles, 1990.

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

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

[6] P. Kumar and M. Satyanarayanan. Supporting Application-Specific Resolution in an Optimistically Replicated File System. In *Proc. of the Fourth Workshop on Workstation Operating Systems*, Napa, CA, October 1993.

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

[8] B. Martin and C. Pedersen. Long-lived Concurrent Activities. Technical Report HPL-90-178, HP Laboratories, 1990.

[9] D. Parker, G. Popek, G. Rudisin, A. Stoughton, B. Walker, E. Walton, J. Chow, D. Kiser, and C. Kline. Detection of Mutual Inconsistency in Distributed Systems. *IEEE Transaction on Software Engineering*, SE-9(3), May 1983.

[10] 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.

[11] M. Satyanarayanan, J. Kistler, L. Mummert, M. Ebling, P. Kumar, and Q. Lu. Experience with Disconnected Operation in a Mobile Computing Environment. In *Proc. of the USENIX Mobile and Location-Independent Computing Symposium*, Cambridge, MA, August 1993.