# Dealing with Tentative Data Values in Disconnected Work Groups

Marvin Theimer, Alan Demers, Karin Petersen, Mike Spreitzer, Douglas Terry, Brent Welch

Computer Science Laboratory
Xerox Palo Alto Research Center
Palo Alto, California 94304 U.S.A.
*contact: theimer@parc.xerox.com*

## Abstract

*This paper describes a problem of weakly-consistent replicated data systems used in support of disconnected groups of people. The problem concerns actions and updates derived from tentative data updates that are ultimately determined to be in conflict. While some such actions and updates can be automatically resolved, many require human intervention. Furthermore, although some file and database systems support internal conflict detection and resolution, derived actions may be external to those systems, implying that human users must ensure that proper consistency is maintained between independent components of the system. The entire problem becomes exacerbated when disconnected work groups are taken into account, where tentative data values may be seen and acted upon by multiple people.*

## 1. Conflict Detection and Resolution

In a world containing mobile computer users who can communicate with each other only part of the time, there is growing interest in disconnected operation and in weakly-consistent replicated data. This can be seen from the success of systems such as Lotus Notes[6], Coda[7][11], and Ficus[4][5]. Some of these systems ignore the issue of conflicting, concurrent updates to data; others support some form of conflict detection and even automatic conflict resolution in certain cases. However, none of them deals with the problem in its full generality.

Two kinds of inconsistency can arise when concurrent updates occur in a partitioned system: write/write conflicts and read/write conflicts. In the former case, two write operations attempt to update the same data in parallel, while in the latter a client reads data that is being updated in parallel and then performs other operations on the basis of the stale data it has read. These "derived" operations may be other write operations, but they may also be "external" operations that are not directly reflected in the replicated data of the system. If the data in a system is partitioned into multiple disjoint subsystems---for example, into a file system and a separate database system---then a write to the data in one subsystem that is derived from a read of stale data in another subsystem would also look like an external operation to each subsystem with respect to the question of conflict detection.

The basic problem is that most systems have only a poor notion of dependencies among multiple data objects, applications and user actions. In traditional file systems such dependencies are generally not recorded, and appropriate compensating actions are not explicitly specified. Systems such as Coda and Ficus originally dealt only with write/write conflicts and have only recently started to address the question of read/write conflicts[8][10].

An example of a system that deals with read/write conflicts is the work of Lu and Satyanarayanan, which introduces the notion of isolation-only transactions as an extension to Coda. By allowing applications to group their file reads and writes into transactions it becomes possible to detect when potential read/write conflicts occur. When a conflict is detected, the system may be able to resolve it automatically, in which case the users of the system never see a problem. However, in many instances a conflict cannot be automatically resolved, in which case it remains the responsibility of some user to deal with the conflicting updates to the system. In particular, external actions taken on the basis of conflicting reads must usually be compensated for by human users. More importantly, the system must rely on human beings to keep track of all such hidden dependencies, to detect chains of hidden dependencies that might exist, and to perform all appropriate compensating actions required to resolve any hidden conflicts that result from those dependencies.

## 2. Disconnected Work Groups

When only individual disconnected users are involved, the lack of explicit system-wide dependency information usually is not a major stumbling block. Conflict detection usually occurs when the (reconnecting) user is around to be made aware of it. The user presumably remembers all actions taken while disconnected, and likely still has access to all data written during the disconnection. For example, Lu and Satyanarayanan limit their isolation-only transactions to single client hosts, implying that file system conflicts cannot have propagated to other hosts that were visible to the previously disconnected host but are not accessible now. Thus, the user can determine and perform any compensating actions made necessary by the conflict resolution at one time and need not remember to “track down” conflicts on other hosts at later times.

Now consider the case of a disconnected work group, that is, a group of collaborating individuals whose computers can reach each other but cannot communicate with relevant parts of the rest of the world. In order for such a group to get work done while “disconnected”, it is necessary that members of the group be able to update data values and share the updated values within the group. However, these updates must be considered tentative since they may conflict with updates being performed concurrently outside the group.

Furthermore, tentative data does not necessarily remain local to a disconnected work group. Membership in the group may change over time, as members of the group leave it and new members join. Membership may also change because of a changing communications environment: network partitions may split the group in half as well as enable new members to join in who couldn’t previously. The result is that people may belong to several disconnected work groups over time and the tentative data they manipulate may be seen by and affect the behavior of multiple groups. For example, a meeting tentatively scheduled in one disconnected work group might end up affecting the scheduling decisions of other disconnected work groups later on that include members from the first group.

The response to a detected conflict becomes much more complicated when the conflicting updates may have been seen by many users. There is no guarantee that all users who have seen a tentative update will be available when a conflict is detected, or that all new tentative data that depends on the conflict will be available. The challenge then is to find all current or previous members of the work group who have seen and acted on a conflicting update, arrange for them to be notified of the conflict, and assume that they can remember (and arrange to compensate for) whatever actions they took that depended on the conflicting data value.

As mentioned earlier, the problem is exacerbated by the fact that disconnected work group members will interact with other people in the system, who must also be notified of the conflict. Consequently, the number of people whose actions might depend on the value of a conflicting update grows over time, and the longer the state of an update must remain tentative, the more workwill likely be needed to undo its effects.

Finally, work groups can “splinter” and later have some or all fragments merge, at which point it becomes necessary to reconcile conflicts visible within the new group. Such conflict resolutions are themselves tentative, since they are based on tentative update values, and users of the resulting data values must be careful to remember this fact. If many of these dependencies and dependencies-of-dependencies are being kept in the heads of human beings, each new permutation of a disconnected work group’s membership will cause added confusion. Figure 1 illustrates an example of this problem.

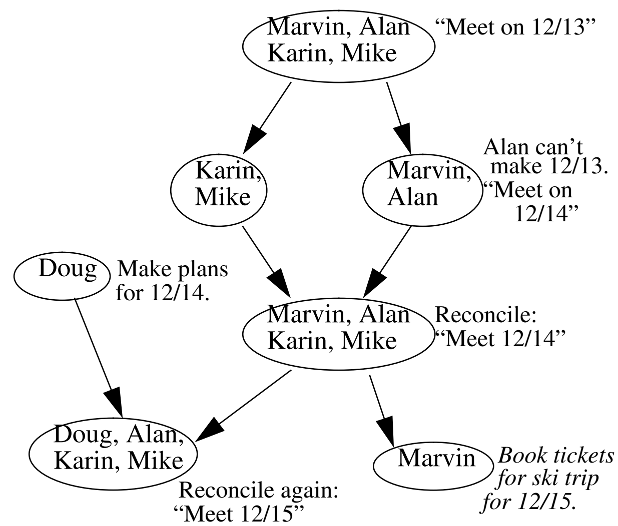


Figure 1. Trying to schedule a project meeting. Marvin forgets that the reconciliation deciding to hold the meeting on 12/14 is only tentative.

## 3. Some Examples

To illustrate the problem we have just described, we present a few examples of the kinds of dependencies that we expect to cause problems.

First, consider a document formatting system in which bibliographic references in documents are made by means of unique key strings that refer to a specific bibliographic entry in a shared bibliographic database. If someone adds a new bibliographic entry to the shared database while he is disconnected, he risks choosing a “unique” key string that will also get chosen by someone else adding a different bibliographic entry. Consequently, when bibliography database update conflicts occur, it is necessary to notify everyone who might have a document that contains a reference to the conflicting bibliographic entry; otherwise some documents risk using the wrong entry when they get reformatted.

In a single disconnected user scenario it is likely that a user’s changes to a bibliographic database will get propagated to the rest of the system at the same time as the user’s use of those changes in one or more document files. Consequently, the user is likely to remember that he must adjust his document files accordingly. In a disconnected work group scenario, various users might update and use the copy of a bibliographic database that resides on a single user’s computer, while keeping their document files on their own computers. If the group then splits up, it will be the responsibility of the lone user who was keeping the copy of the bibliographic database to notify the others about any update conflicts that might invalidate the references in their document files.

Shared calendars are another difficult-to-handle example. A shared calendar system could be structured as a shared database representing meeting rooms, together with a private database for each user containing the meetings he or she was scheduled to attend. A conflict would arise in the shared data base if two disconnected groups tried to schedule meetings in the same room at the same time; in that case, one of the conflicting meetings would be cancelled or rescheduled, and the system should find and notify all the affected users. These, in turn, must update their private calendars accordingly. Furthermore, if any of the affected users has performed other derived actions---such as booking travel plans based on the assumption that a scheduled meeting would occur---then those users will have to remember to effect alternate travel bookings.

## 4. Characteristics of a Solution

The challenge represented by the problems we have described is primarily a practical rather than a theoretical one. The database community has long understood about conflicts between serializable transactions in replicated data systems[1], optimistic concurrency control and cascading transaction aborts[2], and compensating actions for the externally-performed operations of aborted transactions[3]. What is hard is translating this understanding into a practically usable system that spans more than just a single database or file system: the solution arguably needs to encompass *everything* that people interact with during their normal work, from file systems to airline reservation systems to the paper calendars they carry around in their briefcases.

We believe that a system designed to address the problems we have described will need to support at least the following:

- It will need to keep track of which individuals have seen each tentative update value produced in a disconnected work group. This information will also have to be maintained about all users that indirectly come into contact with a tentative update value by interacting with someone who has already seen that tentative update value.
- It will need to keep track of dependencies that users either implicitly or explicitly create between a tentative update value and other data or actions in the system.
- It will need to inform the users and applications who have (directly or indirectly) seen a tentative update value whenever a conflict is discovered. Given a list of the dependencies derived from a tentative update value, the system could simply hand those to the relevant human beings and let them deal with actually invoking all necessary compensating actions and updates. However, to the extent that such actions can be automated, the system will be able to avoid having to depend on the reliability and availability of human beings to keep things consistent.
- It will need to let users know when they are dealing with tentative data and when they are dealing with “committed” data. When dealing with the former, users must be careful to not initiate any actions that can’t be undone. Irrevocable actions (such as buying a non-refundable ticket) should only be derived from data that is known not to be tentative (for example, when a meeting has somehow been “confirmed” or when it can be established that a tentative update value has reached some “primary” site that declares it to be permanent).

This implies that application user interfaces may need to be modified to make the tentativeness of data and (derived) information being displayed much more explicit. It may also be useful to display the “history” of a tentative data value, by which we mean the set of tentative data values that something was derived from. For example, it might be interesting to know if something was derived from a single tentative value or from multiple ones that perhaps even required (tentative) conflict resolution to yield the values currently being used.

## 5. Concluding Remarks

In this position paper we have described a problem of weakly-consistent replicated data systems concerning how to undo actions and updates derived from tentative data values that are ultimately determined to be in conflict. While this problem exists even for individually disconnected users of a system, we believe that it will become a significantly more serious problem when disconnected work groups are involved. This is because individual users can usually remember all the actions they took while disconnected and are typically informed of update conflicts in a timely fashion. In contrast, disconnected work groups allow multiple individuals to see tentative data values and then go their separate ways afterwards. Worse yet, those individuals may interact with other individuals they meet, allowing tentative data values to potentially be seen by an ever-expanding set of people.

The consequence we take from this problem is that systems providing weakly-consistent, replicated data semantics should be structured so that all the relevant interactions among collaborating individuals and all the dependencies among data and actions be made explicit. Only then can update conflict resolution systems hope to maintain system integrity without having to depend entirely on the availability and reliability of human beings.

## 6. Acknowledgements

The problem described in this paper is being explored as part of the Bayou project, which includes the authors of the paper as well as Carl Hauser. We have also benefited from conversations with a number of colleagues, especially Atul Adya, Helen Davis, Terri Watson, and Mark Weiser.

## 7. References

[1] P. A. Bernstein and N. Goodman. An algorithm for concurrency control and recovery in replicated distributed databases. *ACM Transactions on Database Systems* 9(4):596-615, December 1984.

[2] S. Davidson, H. Garcia-Molina, and D. Skeen. Consistency in a partitioned network: A survey. *ACM Computing Surveys* 17(3):341-370, September 1985.

[3] J. Gray and A. Reuter. Transaction processing: concepts and techniques, Morgan Kaufmann, 1993.

[4] R.G. Guy, J.S. Heidemann, W. Mak, T.W. Page, Jr., G.J. Popek, and D. Rothmeier. Implementation of the Ficus replicated file system. *USENIX Conference Proceedings*, pages 63-71, USENIX, June 1990.

[5] J. S. Heidemann, T. W. Page, R. G. Guy, and G. J. Popek. Primarily disconnected operation: Experiences with Ficus. *Proceedings Second Workshop on the Management of Replicated Data*, Monterey, California, November 1992, pages 2-5.

[6] L. Kalwell Jr., S. Beckhardt, T. Halvorsen, R. Ozzie, and I. Greif. Replicated document management in a group communication system. *Proceedings Conference on Computer-Supported Cooperative Work*, Portland, Oregon, September 1988.

[7] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems* 10(1): 3-25, February 1992.

[8] P. Kumar and M. Satyanarayanan. Supporting application-specific resolution in an optimistically replicated file system. *Proceedings IEEE Workshop on Workstation Operating Systems*, Napa, California, October 1993, pages 66-70.

[9] Q. Lu and M. Satyanarayanan. Isolation-only transactions for mobile computing. *ACM Operating Systems Review* 28(2):81-87, April 1994.

[10] P. Reiher, J. Heidemann, D. Ratner, G. Skinner, and G. Popek. Resolving file conflicts in the Ficus file system. *Proceedings Summer USENIX Conference*, June 1994, pages 183-195.

[11] M. Satyanarayanan, J.J. Kistler, P. Kumar, M.E. Okasaki, E.H. Siegel, and D.C. Steere. Coda: a highly available file system for a distributed workstation environment. *IEEE Transactions on Computers* 39(4):447-459, April 1990.