Confidential – for review only.

Dear sirs;

Enclosed please find a submission to the 21st International Conference on Distributed Computing Systems (ICDCS-21). The title of the paper is

"An examination of conflicts in a weakly-consistent, replicated application"

and the authors are

Douglas B. Terry, Marvin Theimer, Karin Petersen, and Mike Spreitzer.

Please direct all correspondence concerning this paper to Marvin Theimer at

Microsoft Research
Bldg 112/2116
One Microsoft Way
Redmond, WA 98052

425-703-2889 (office)
425-936-7329 (fax)

theimer@microsoft.com

Sincerely,

Marvin Theimer.

Confidential – for review only.

An examination of conflicts in a weakly-consistent, replicated application

Douglas B. Terry, Marvin Theimer, Karin Petersen, Mike Spreitzer[1]
Computer Science Laboratory
Xerox Palo Alto Research Center

ABSTRACT
Weakly-consistent, replicated applications, in which one or more users concurrently update shared data can often lead to conflicts that must be detected and resolved. Automatic conflict detection and resolution turns out to be a very difficult thing to provide in a systematic manner. This paper looks at a particular application, a replicated mail tool, and presents experience gathered in writing the conflict management portion of the application. Conflict management proved to be complicated and difficult to implement primarily because conflicts are defined not only by the types of concurrent operations involved but also by their parameters, the user's intentions, the application state, the database schema, and the update ordering guarantees of the replication substrate. Furthermore, automatic conflict detection and resolution must be customizable so that users' different preferences and work styles can be accomodated.

Keywords
Replicated mail tools, weak consistency, replication, concurrency control, conflict detection, conflict resolution

1 INTRODUCTION
Weakly-consistent, replicated applications provide a number of desirable properties. Replication of state among multiple machines increases its availability and the responsiveness of the application in the face of failures. An "update anywhere" model permits a user to operate effectively on machines that do not have continuous network connectivity, such as an intermittently connected laptop. Electronic mail reading is an example of an application that can readily benefit from these properties since users commonly read email from both desktop and laptop machines and commonly must deal with intermittent connectivity.

Because state is replicated and clients may independently access different replicas at various times, conflicting updates may occur. Conflicts can occur even if only a single user is involved. Consider a user who employs a replicated mail tool and is processing unread email from home and suffers from a network disconnection. The network disconnection prevents the replica on his office machine from receiving the resulting updates that mark messages as read, move messages among folders, etc. Later on, the user might continue processing email in his office and inadvertently perform an operation that conflicts with one invoked earlier on the home machine. An example is moving a message to a different folder than it was moved to on the home machine.

Conflicting updates can also occur when user actions interact with automation rules provided by the mail tool. Modern mail tools often provide rules for automatically moving newly received messages to various folders, based on the contents of the messages. If, for example, the mail tool implementation gives messages unique ids that are relative to the folder they are in then assignment of message ids can yield update conflicts if a user is concurrently moving messages between folders at another replica. Changing an automation rule can also yield conflicting actions until all replicas have been notified of the changed rule.

The possibility of conflicting updates means that unless all conflicts can be automatically detected and – to the extent possible – resolved, users will have to worry about inconsistencies, lost updates, and unintended results when using a weakly-consistent, replicated application. Hence support for automated conflict management is a crucial aspect of any weakly-consistent, replicated application.

This raises the issue of how easy or difficult it is to provide automated conflict management and the goal of the work described in this paper was to explore that question. The approach taken was to experiment with a representative application and then extrapolate from the experience gained to general conclusions. A replicated mail tool was chosen as test bed because it represents an important application in its own right and because it has a fairly rich data and operational model.

[1] Doug Terry is currently with Cogenia Corp. Marvin Theimer is with Microsoft Research, Karin Petersen is at the Stanford Business School, and Mike Spreitzer is with IBM Research.

The replicated mail tool we constructed was *BXMH*, which permits a user to read messages, delete them, move them into folders, and perform other operations even while disconnected from his mail server or from other BXMH replicas.[2] BXMH was built by taking the EXMH mail tool [11] and replacing its file-based storage layer with a Bayou replicated database management system [1, 9, 10]. Bayou manages replicas of a relational database with an update-anywhere replication scheme and supports automatic handling of conflicting updates by means of application-specific dependency checks and resolution procedures [9].

Our initial hopes and expectations were that understanding and dealing with all the possible kinds of update conflicts that can occur would be tedious but straightforward. It turned out to be far from it: not only were there many different cases to resolve, but conflicts turned out to be defined by the types of concurrent operations involved, their parameters, the user's intentions, the application state, the database schema, and the update ordering guarantees of the replication substrate. Furthermore, the very notion of what constituted a conflict depended, in part, on users' preferences and work styles, implying that conflict management must support user customization.

The rest of this paper provides an overview of differing conflict management approaches in Section 2, lessons learned about conflict management from building BXMH in Section 3, and our general conclusions in Section 4.

2 APPROACHES TO CONFLICT MANAGEMENT

Replicated systems that permit uncoordinated updates to shared data have employed a variety of schemes for automatically detecting and resolving conflicts.

In many replicated systems, a conflict is defined as concurrent updates to the same object or database record. Concurrent updates, and therefore conflicts, can be detected by associating an update timestamp with each object. Updates to an object note its previous timestamp. Before attempting to apply an update, a replica checks that the timestamp on its copy of the object is the same as that in the update. If the timestamps do not match, the conflict can be resolved by either discarding the update, applying it anyway, creating a forked version of the updated object, or requesting user intervention. Whichever choice is made, either one of the concurrent updates is lost or users must be prepared to deal with multiple conflicting data copies. Another problem is that this form of concurrency control cannot detect semantic conflicts implied by updates to two different data records. This simple approach to conflict management is employed in many commercial database replication schemes, such as Lotus Notes, as well as in virtually all commercial products for synchronizing mobile devices with office workstations, such as Starfish's TrueSync software used in Palm PDAs and Motorola cellphones.

Some research prototypes have developed more sophisticated conflict detection and resolution mechanisms. These systems generally present update conflicts in the form of a "conflict table" . Each update operation occupies a row and column of the table. Entries in the table indicate whether the row operation conflicts with the column operation when performed concurrently. A conflict is detected by looking up concurrent updates in the conflict table. Entries in the table can also indicate how to resolve detected conflicts. A "merge matrix" is an example of a conflict table that indicates operations to be performed instead of the conflicting row operation [4]. Another example is a "transformation table" that transforms the row operation into an operation that has an equivalent effect on the application state when performed after the column operation. Operational transformation has been studied extensively for collaborative editing applications [2, 3, 5, 6, 7]. Unfortunately operation table-based approaches do not easily account for conflict dependencies on operation parameters, application state, users' intentions, etc.

Bayou provides a more general approach to conflict management that takes into account both update state and the full database state at the time of an update. Updates submitted by clients to different replicas eventually propagate to all replicas by means of a peer-to-peer reconciliation process that totally orders updates at all replicas. Each replica merges updates from all replicas into a single, ordered log and then applies them sequentially to its local database state. When new updates are received that should be ordered before the latest update in the log the database is rolled back to its state prior to the earliest point at which an update will be inserted. The new updates are inserted into their proper locations in the log and then all updates past the log rollback point are reapplied to the database.

[2] We use the term *mail server* to refer to a well-known destination machine that email senders can direct email messages to. BXMH removes email messages from a user's mail server and stores them in its own (replicated) email database.

Insert (message)
Delete (message)
Move (message, fromFolder, toFolder)
Copy (message, toFolder)
Remove (message, fromFolder)
Create (folder)
Rename (folder, newName)
Destroy (folder)
Display (message)
Renumber (folder)

Figure 1: Operations that update a mail database.

Bayou-based applications implement conflict management by attaching an application-specific conflict detection predicate and conflict resolution procedure to each update that a client submits to a database replica. When an update is applied to the database state at some replica the associated conflict detection predicate is run first. This predicate can encode application-specific knowledge about such things as operations, their parameters, and their semantic intent and can also query the current state of the database. If the predicate returns TRUE then the update is applied, otherwise the associated conflict resolution procedure is run to produce a new update that will be applied instead. The resolution procedure can examine both the contents of the update as well as the current state of the database. It can produce anything from NULL to an error log update to an application-specific alternative update that resolves the conflict. More details can be found in [9].

3 LESSONS FROM BXMH

This section presents eight aspects of application-specific conflict detection and resolution that we learned from building the BXMH replicated mail tool application.

3.1 Lots of potential conflicts arise in a replicated mail tool application.

Figure 1 lists the operations on a mail tool that can affect the underlying mail database. For users of mail systems, these should be familiar with the possible exception of the Renumber operation. BXMH associates a numeric number with each message in a folder. As messages are deleted or moved from the folder, these numbers can become sparse. The Renumber operation assigns new contiguous numbers to messages in a folder. The Display operation, in addition to showing the message contents, may update the database to record that a message has been seen. The Insert operation checks the user's mail server for new messages, removes any found there, and places them in the BXMH "inbox" folder. The other operations are directly available in the BXMH user interface via buttons or menus [11]. While this interface includes lots of other operations for manipulating windows, composing messages, searching, and so on, only the operations in Figure 1 modify the database. Hence, only these 10 operations can possibly be involved in update conflicts.

In examining these operations and their effect at the database level, we discovered 46 different conflict scenarios. Some were obvious, like deleting a message conflicts with any other operation on the message and renumbering a folder conflicts with other folder operations. Others were less obvious and possibly even debatable, like operations that create two folders with different names may conflict. The other lessons in this section shed more light on the subtleties of defining and dealing with conflicts.

3.2 Conflicts depend on more than the type of operations involved.

Conflict tables, as discussed in section 2, define conflicts as occurring between two operations. In a replicated mail tool application, to determine if two operations conflict, considering their type is not sufficient. The system must also examine the parameters to the operations. Consider two move operations: Move (m1, f1, t1) and Move (m2, f2, t2). For what values of m1, m2, f1, f2, t1, and t2 do these operations conflict? These operations certainly conflict if m1=m2 and f1=f2 and t1#t2. That is, the operations conflict if they move the same message into different folders. However, if the operations involve different messages or if the target folder is the same for both operations, then one could reasonably argue that they do not conflict.

Interestingly, whether two operations conflict might also depend on the state of the mailbox. For example, suppose that a message is in exactly two folders. Suppose the user removes the message from the first folder (but leaves it in the second folder) and later, at another replica, removes the message from the second folder (but leaves it in the first). If both operations are allowed to proceed, then the message will end up in no folders, effectively deleting the message. This is probably not what the user intended, and, therefore,

these two operations should be viewed as conflicting. On the other hand, if the message was also in a third folder, then the attempts to remove it from the other folders could safely be performed and not detected as a conflict.

3.3 What users consider conflicts and acceptable resolutions is a matter of debate.

One of our biggest surprises was discovering that what constitutes a conflict is not always clear. Even within our small design team, there were numerous disagreements about which operations conflict and under what circumstances. Some operations clearly conflict. For many others, it depends on the user's intention, which is hard for the system to capture.

As an example, consider an operation that moves a message to some folder and a concurrent operation that renames this folder. One might argue that the desired result, irrespective of operation order, should be that the message ends up in the renamed folder. But is this what the user intended? Suppose a folder is being used for compiler bug reports. While reading email on an airplane the user observes that the folder named "Compiler bugs" contains two types of messages, those reporting Java compiler problems and those reporting C++ bugs. The user decides to separate these into different folders. The user creates a new folder named "Java compiler bugs", moves all of the Java-related messages to this new folder, and then renames the original folder to "C++ compiler bugs". Back at the office, the replica on the user's office machine is receiving new email messages from a mail server and moving them to various folders according to the user's mail automation rules. Included among the newly-arrived messages are Java-related compiler bug messages, which will automatically be moved into the "Compiler bugs" folder – which has concurrently been renamed "C++ compiler bugs". Thus, one might argue that conflicting Move and Rename operations should be flagged as irreconcilably conflicting.

Scenarios of this sort caused us to completely rethink the notion of conflict and to realize that conflicts may occur for various reasons. We decided to have the system detect any operations that might be considered conflicts under a reasonable scenario and then give users choices for what to do about such "conflicts", including nothing. Section 3.8 discusses BXMH's handling of user preferences in more detail.

3.4 Conflicts may depend on a user's style of interaction.

Although systems have traditionally defined conflicts based on the semantics of the operations available in an application, the intention of the users performing the operations is actually what's important. The replicated mail tool application presents an example where users' intentions are not necessarily clear from the operations that they perform. A previous section argues that two concurrent Move operations conflict if they involve the same message and different target folders. This is true since no matter in what order the Moves are performed, one of the user's intentions is not met. Does the same hold for two Copy operations? The semantics of Copy are different from Move in that the message is not removed from its current folder. Assuming that a message initially resides in a single folder, then a Move operation preserves this property while a Copy operation leaves the message in two folders. Thus, one could reason that concurrent Copy operations do not conflict; the message simply ends up in all of the folders to which it is copied.

In practice, however, we have observed that users occasionally perform the equivalent of a Move operation by first copying a message to the target folder and then removing it from its previous folder. This is an artifact of the user interface making Move a more awkward operation than Copy. More generally, it is the result of applications providing more than one way to do many things. Since concurrent Copy operations do not conflict and concurrent Remove operations of the same message do not conflict, two Moves performed as Copies followed by Removes do not conflict if the two Copy operations are ordered before the two Remove operations. The result is that the message ends up in two folders, which may not be what the user intended. This example serves to point out that, although the semantics of Copy are well-defined, a user's intention when performing a Copy is not. Users who perform Moves as Copies followed by Removes want different conflict detection rules than users who perform Moves or Copies as appropriate. To accommodate both types of user interaction, we considered adding a flag that a user could set to indicate whether he wants BXMH to maintain the invariant that a message should reside in at most one folder. If set, the system would allow a message to be copied to a second folder temporarily but not allow a concurrent Copy.

3.5 Conflicts occur at both the database level and user level.

Each user-level operation listed in Figure 1 maps down through software into one or more operations on a relational database. Adding a new message to a mailbox, for instance, updates six different database tables. Operations that conflict at the user level may or may not conflict at the database level, and vice versa. Hence, determination of which operations conflict must consider not only the user- visible semantics of the

operations but also their effect on the underlying database.

Some operations, like moving a message to a folder and destroying that same folder, conflict at both the user and database level. As an example of operations that conflict only at the user level, consider a user moving a message to a folder at one replica while later "concurrently" renaming the folder at another replica. Section 3.3 discussed why these two operations may conflict from a user's perspective. At the database level, one table, the MsgFolder table, maintains the many-to- many mapping between messages and folders. However, this table, for space efficiency reasons, refers to folders by a short unique identifier. Another table, the Folders table, maps these system-assigned unique identifiers to user- assigned folder names. A Move operation updates the MsgFolder table while a Rename operation updates the Folders table. Thus, these operations never conflict at the database level.

On the other hand, some operations that can be performed concurrently at the user level with well-defined semantics, conflict at the database level. For example, concurrently creating new folders with different names is perfectly reasonable. However, concurrent attempts to create folders may conflict at the database level because of the need to assign unique identifiers to each folder. Since in BXMH unique identifiers are obtained from a simple counter, two concurrently created folders will likely be assigned the same identifier, thereby causing havoc at the database level if a conflict is not detected. Fortunately, such a conflict can be readily detected and easily resolved by reassigning a new identifier to one of the folders. Other examples of operations that conflict only at the database level are concurrent identical operations, such as a user deleting the same message or moving a message into the same folder on two different replicas. In the serial execution of these operations at each replica, the second operation will fail because the message no longer exists or because it already exists in the target folder. In this case, again, the resolution is straightforward and does not involve the user: simply ignore the second operation.

3.6 The database schema design affects conflicts.

The schema governing how application data is stored in a database can affect which operations conflict at the database level as well as the detection and resolution of conflicts. For BXMH, we originally defined a database schema that closely modeled the data structures used in the EXMH mail tool. Later, we modified the schema to reduce unnecessary conflicts. For example, the Folders table originally included the number of messages in each folder. Every Move, Copy, and Remove operation needed to update the message counts for the folders involved. Thus, any concurrent operations on a folder would conflict at the database level even though operations such as moving different messages from or to a folder do not conflict semantically. To reduce such database conflicts, we removed the message count from the Folders table. This information does not need to be stored directly in the database since it can be computed by counting tuples in the MsgFolder table. BXMH now computes the size of a folder when the folder is displayed and caches this information in non-persistent storage, thereby eliminating a large class of conflicts.

Modifications to the schema were also made to simplify conflict detection. Some operations depend on the contents of a folder. The Bayou dependency checks for such operations need to verify that the contents of a folder are the same as they were when the user issued the operation. Originally, these dependency checks needed to record the complete contents of a folder and then compare the saved contents to its current contents. To eliminate this expensive operation, we added a timestamp to the Folders table that is updated each time a folder's contents change, that is, each time a message is added to or removed from the folder. Importantly, unlike the message counts that caused conflicts in the Folders table and hence were removed, the update timestamp does not introduce additional conflicts since it can be safely updated by concurrent folder operations.

3.7 Session guarantees prevent out-of-order conflicts.

To enable experimenting with thin client settings, BXMH is structured as a UI client part and an email storage part. The client part can interact with any local or remote BXMH storage replica. In replicated systems where client machines, which run the application code, are distinct from server machines, which manage replicas of the application's database, even operations performed sequentially by a given client can potentially be seen as conflicting.

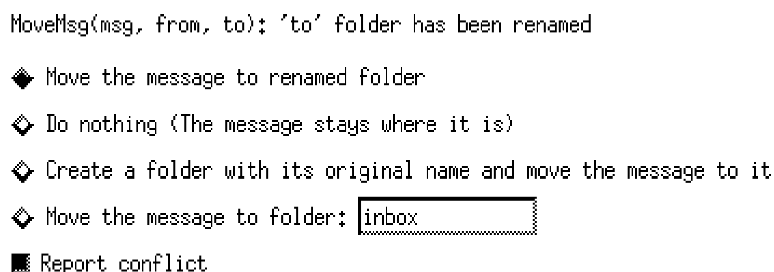

Figure 2: A portion of BXMH's conflict preferences menu

Consider the case where a user running the BXMH client code on their sub-notebook accesses replicas on mailbox servers over a wireless network. The user creates a folder and then moves a message into this new folder. Suppose the Create operation is performed at a replica that is no longer accessible when the user does the Move. The Move operation is therefore performed on a different replica. This second replica, or any other replicas, may receive the Move operation before receiving the Create operation. In this case, the Move operation fails because the target folder does not yet exist. It appears as though this Move operation conflicts with a concurrent operation that destroyed the target folder.

This is an example of a "false conflict" caused by reordering of operations. In BXMH, we avoided false conflicts of this sort by utilizing Bayou's session guarantees to enforce a causal ordering on updates made by a given client [8]. Not only did this simplify BXMH's dependency checks and merge procedures, but it also simplified the design task in that arbitrary low-level operation reorderings within a high-level mail operation did not need to be considered.

3.8 Users want a choice of conflict resolution options.

Because we observed that different users have different views of what constitutes a conflict and different preferences for how to resolve conflicts, BXMH gives users fine control over these choices. The one and only change made to the EXMH user interface was the addition of a "Conflict Control" preference menu. At the top of this menu, users can chose general preferences for dealing with conflicts. For instance, users who do not trust automatic mechanisms may perform manual resolution by selecting "Always do nothing" and "Always report conflicts". The rest of the menu contains descriptions of each possible conflict along with all possible means for resolving each conflict. Figure 2 presents a screen snapshot depicting one conflict scenario and the resolution choices. Essentially, users can customize their "merge policy" by selecting from the various options. The initial default choices are intended to provide reasonable functionality for novice users.

How best to present the staggering number of conflict scenarios and possible resolutions to users remains an open issue. Should users be presented with a small set of profiles they can choose rather than long lists of conflict scenarios? Should conflicts be described as ordered pairs of conflicting operations, unordered pairs, or operations and a database state? We decided on the latter since it reflects the way that conflicts are actually handled in Bayou. Unfortunately, we were unable to perform studies to determine if users could actually deal with BXMH's complex conflict preferences menu.

4 CONCLUSIONS

Even single-user applications, such as electronic mail tools, must deal with novel conflict management and concurrency control challenges when required to run in an environment that requires weakly-consistent replication technology in order to provide high availability. Contrary to our hopes and expectations, understanding and dealing with all the possible kinds of update conflicts that can occur in such an environment proved to be exceedingly difficult.

Despite the existence of an extensive literature in the CSCW community on conflict resolution for collaborative applications, we found that BXMH introduced a substantially greater range of update conflicts than are covered by this literature. In hindsight, this was not surprising since replicated mail databases have a richer set of operations and a more complex data model than previously studied groupware applications like editors, bibliographic databases, and spreadsheets. Many of the lessons presented in this paper derive from subtle interactions between mail tool operations at both the user level and the database level.

To make matters worse, conflicts are defined by users' intentions as much as by the semantics of concurrent operations. Previous work recognized "intention-preservation" as a correctness criteria that should be maintained in the design of concurrency control schemes for collaborative applications [5, 7]. Work on operational transformation was an attempt to transform operations in a way that preserves users' intentions. Our experience with BXMH suggests that automatically preserving a user's intention is not always possible. For one thing, the intention may not be evident in the sequence of operations a user performs. For another, two concurrent operations may conflict in such a way that preserving the intention of both is not feasible. For most conflicts in BXMH, a number of reasonable resolutions are possible. Therefore, users have to be given choices that indirectly set the policies governing conflict detection and resolution.

The key conclusion we draw from our work is that conflict management in a weakly-consistent, replicated application is, in general, a very difficult thing to provide in a systematic manner. We were surprised by the number of different ways in which conflicts can occur and by how sensitive conflict management is to both user intentions as well as details of the underlying database design.

Our corollary conclusion is that the full power of an application-specific conflict management architecture, such as is provided by Bayou, is necessary to be able to deal with all the different kinds of conflicts that can occur. BXMH made heavy use of almost every feature that was available from the underlying Bayou storage system and we are doubtful that a less general infrastructure would have been sufficient.

An interesting question we have not addressed is how adaptable people are to using applications that do not handle all possible conflict cases. BXMH was an attempt to explore how far we could automate conflict management, without asking whether some conflict cases might not be worth handling because users can be "trained" to avoid them.

Another question for further investigation is whether conflict management code could be reused across multiple applications. For example, conflict management for many PIM applications, such as calendars, address books, and to-do lists, is essentially the same. It is interesting to speculate how many application classes – and hence conflict management frameworks – would be needed in order to cover the applications that users commonly use.

5 ACKNOWLEDGMENTS

We thank Alan Demers for helping design and implement the Bayou system, Brent Welch for building the EXMH mail tool on which we based BXMH, Keith Edwards and Beth Mynatt for helping us better understand Bayou's relationship to CSCW applications, and Atul Adya for feedback on the writing of this paper.

6 REFERENCES

1. Edwards, W. K., Mynatt, E. D., Petersen, K., Spreitzer, M. J., Terry, D. B., and Theimer, M. M. Designing and implementing asynchronous collaborative applications with Bayou. Proceedings User Interface Systems and Technology (UIST) (Banff, Canada, October 1997), 119-128.

2. Ellis, C. A. and Gibbs, S. J. Concurrency control on Groupware systems. *Proceedings ACM SIGMOD* (Portland OR, June 1989), 399-407.

3. Matthias, R., Nitsche-Ruhland, D., and Gunzenhauser, R. An integrating, transformation-oriented approach to concurrency control and undo in group editors. *Proceedings CSCW '96* (Cambridge, MA, November 1996), 288-297.

4. Munson, J. P. and Dewan, P. A flexible object merging framework. *Proceedings CSCW '94* (Chapel Hill NC, October 1994), 231-242.

5. Suleiman, M., Cart, M., and Ferrie, J. Serialization of concurrent operations in a distributed collaborative environment. *Proceedings Group '97* (Phoenix AZ, November 1997), 435-445.

6. Sun, C., Jia, X., Zhang, Y., and Yang, Y. A generic operation transformation scheme for consistency maintenance in real-time cooperative editing systems. *Proceedings Group '97* (Phoenix AZ, November 1997), 425-434.

7. Sun, C. and Ellis, C. Operational transformation in real- time group editors: Issues, algorithms, and achievements. *Proceedings CSCW '98* (Seattle WA, November 1998), 59-68.

8. Terry, D., Demers, A., Petersen, K., Spreitzer, M., Theimer, M., and Welch, B. Session guarantees for weakly consistent replicated data. Proceedings International Conference on Parallel and Distributed Information Systems (PDIS) (Austin TX, September 1994), 140-149.

9. Terry, D. B., Theimer, M. M., Petersen, K., Demers, A. J., Spreitzer, M. J., and Hauser, C. Managing update conflicts in Bayou, a Weakly Connected Replicated Storage System,. Proceedings 15th Symposium on Operating Systems Principles (SOSP) (Cooper Mountain CO, December 1995), 172-183.

10. Terry, D. B., Petersen, K., Spreitzer, M. J., and Theimer, M. M. The case for non-transparent replication: Examples from Bayou. IEEE Data Engineering 21(4):12-20, December 1998.

11. Welch, B. B. Customization and flexibility in the exmh mail user interface. Proceedings Tcl/Tk Workshop (Toronto, Canada, 1995), 261-268.