Supporting Asynchronous Collaborative Editing in Mobile Computing Environments *

Marcos Bento, Nuno Preguiça
CITI/DI, FCT, Universidade Nova de Lisboa, Portugal

Keywords: Asynchronous collaborative editing systems, optimistic replication, reconciliation, operational transformation.

Abstract: Mobile computing environments have changed in recent years, with the increasing use of different types of mobile devices and wireless communication technologies. To allow users to store and share their data in this new environment, we are building the Files EveryWhere (FEW) system, that explores the multiple available storage and communication devices to provide good availability and performance. This system is based on optimistic replication and it can be used as a tool for supporting asynchronous collaborative edition, as it allows users to cooperate by accessing and modifying shared documents. Our approach is implemented using the file system interface, thus allowing users to continue using their favorite application. In this paper, we focus mainly on the FEW reconciliation mechanism. Our approach is based on operational transformation and it includes several new techniques. First, we propose a new technique for handling operations in operational transformation algorithms that supports efficient epidemic dissemination. Second, we propose a new set of transformation functions that explicitly handle line versions in text files. Finally, we propose a set of transformation functions that explicitly handle file versions for opaque files.

1 Introduction

Mobile computing environments have changed in recent years with the increasing use of different types of portable devices, ranging from mobile phones to laptops, and from MP3 players and digital cameras to portable storage devices, such as flash-disks. Although most of these devices are not general-purpose computing devices, they can be used to transport users' data, as they often have gigabytes of storage. These devices can act as sophisticated, large-capacity storage devices either attached to a computer or directly connected to a network. These new devices lead to an environment with characteristics that differ from the assumptions taken in older mobile data management systems (Satyanarayanan, 2002; Terry et al., 1995).

In this new mobile computing environment, users can transport most of their data with them all the time, thus allowing users to access and modify it at any time. Additionally, users may explore peer-to-peer ad-hoc wireless connectivity to share and synchronize data with nearby users, thus improving the potential for collaboration with other users.

To address the characteristics of the new mobile computing environment, we are developing the Files EveryWhere (FEW) system. The FEW system is a distributed file system that intends to explore the multiple available storage devices to allow users to safely store and share their data.

In FEW, users can group related files in containers and share their containers with other users. Users can create copies of containers in multiple storage devices. Additionally, temporary copies of recently used files are automatically created in portable storage devices. This approach, combined with an optimistic read any, write any model of data access, provides high data availability for data sharing. The system explores the existence of multiple replicas to improve performance and reduce energy consumption, by selecting which replica to access at each moment.

The optimistic replication approach, although im-

*This work was partially supported by FCT/MCTES with FEDER co-funding – project #POSC/59064/2004.

portant to always allow users to produce their contributions, may lead to conflicting updates. Unlike recent file systems with support for mobile computing (Tolia et al., 2004; Sobti et al., 2004), our synchronization process is based on update propagation and includes a generic reconciliation mechanism based on operational transformation (Ellis and Gibbs, 1989; Sun and Ellis, 1998). This generic approach must be customized for each file type.

For adapting operational transformation algorithms to mobile computing environments, we propose a set of new techniques. First, for supporting the typical epidemic dissemination of updates efficiently, we propose a technique that manipulates operations' dependency information at each site, thus allowing a site that receives an operation to capitalize the transformations performed in intermediate sites and requiring a single version of each operation to be stored at each site. Second, we propose a new set of transformation functions that explicitly handle line versions in text files, thus addressing the shortcomings in reconciliation solutions typically used in asynchronous settings. Finally, we propose a set of transformation functions that explicitly handle file versions for opaque files.

The remainder of this document is organized as follows. The next section briefly presents the FEW file management system. Section 3 presents the generic reconciliation and type-specific solution used in FEW. Section 4 compares our work with relevant related work and section 5 concludes the with some final remarks and future work.

2 FEW File System

The FEW system is composed of several machines, each one containing internal storage units and hosting a variable set of portable storage devices. Portable storage devices with no computing or network communication capabilities are only available to the system when they are connected and under the control of a single computer. Portable devices with limited computing and network communication capabilities (e.g. mobile phones) may act as a machine in the system or may be connected and controlled by a host computer as a passive storage device.

FEW manages groups of files, called containers. A container typically stores the data of some (cooperative) project. Containers can be replicated at multiple storage devices and be shared by multiple users. Users can explicitly create a new replica of a container. The system also automatically creates temporary replicas for recently accessed containers in its

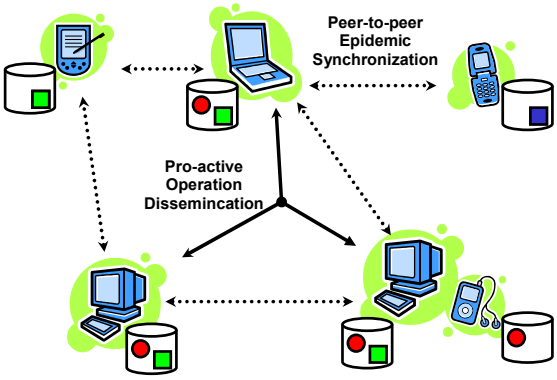

Figure 1: FEW architecture.

controlled storage devices if space, performance and energy constraints allow. Each container replica may contain only a subset of the files in the container.

The system adopts an optimistic replication approach, allowing users to modify any replica at any time. Updates to files are propagated to all replicas using two mechanisms (similar to (Birman et al., 1999)). First, to accelerate the convergence process, new updates are asynchronously propagated to all replicas using a best-effort event-dissemination system that takes into consideration portable devices' constrains, such as network conditions, energetic levels, voluntary disconnection of devices or voluntary isolation of devices (e.g. when editing a set of files containing program code, users may decide to work in isolation until they have a stable version of their changes, so that unfinished code from one user will not lead to compile and execution errors to other users). Second, to guarantee eventual convergence, replicas are synchronized in periodic pairwise epidemic propagation sessions (Demers et al., 1987). In these sessions, a pair of replicas exchanges the updates unknown to the other. This approach guarantees that each replica will receive all updates produced in all other replicas, even if it never communicates directly with some of the replicas. In Figure 1, we depict a sample FEW system, consisting of several computing devices with connected portable storage devices.

An important goal of FEW is to allow users to continue to use their preferred applications to access and modify data. To this end, we have decided to make the data stored in the FEW system accessible through the traditional file system interface, thus allowing users to use any application that stores data on files. The FEW system is built on top of FUSE system (FUSE, 2007) and in each node, the system intercepts all file system calls in the kernel and redirects them to the FEW daemon, as depicted in Figure 2.

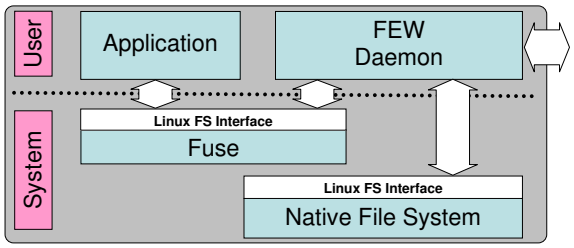

Figure 2: FEW Daemon Architecture

The FEW daemon is responsible to process each file system call, deciding which replica to access on read operations and processing writes as explained in the next section.

Besides immediately propagating recently executed updates using an event-dissemination system, as explained before, the FEW daemon also publishes information about relevant events at each node, such as access to files, user status (based on its activity), etc. In collaborative editing activities, this information can be used to provide awareness information about other users activity. For example, a user may list, for each user, the files he has recently accessed or modified. It is also possible to notify a user when he accesses some file that some other user has recently opened in other replica.

All this functionality can be provided by an awareness-provider application that listens to the events published by the FEW daemon and runs independently of the applications users are using to modify the shared files. For example, for notifying users of possible concurrent accesses to the same files, when the awareness-provider application receives an event about a local *open* file system call, it can check the recently received events from other replicas to verify if some other user has recently accessed the file. If this is the case, it can present a pop-up windows notifying the user about this fact, thus providing awareness information without modifying the application the user is running. Thus, it is possible to provide typical collaborative features while allowing users to continue using their (non-collaborative aware) applications. It is also possible to further improve support for collaborative teams by combining this simple awareness strategy with other tools (e.g. (Fitzpatrick et al., 2006)).

3 Synchronization Process

The synchronization process, which enables all replicas in the system to achieve the same final state, is divided into three different steps, as shown in Figure 3.

The first step is to infer the set of operations executed in a file. FEW infers operations comparing two versions of a file. To this end, when a user executes a "open-read/write-close" session, the system automatically stores the original and new version of the file. This is executed transparently by the FEW daemon. After the file is closed, a type-specific program is executed to infer semantic-rich operations by comparing the original and the new version of the file.

In the second step, operations are propagated among replicas. As explained before, an operation may be directly or indirectly propagated to all other replicas using an event-dissemination system or in peer-to-peer epidemic propagation sessions.

Finally, when an operation is received in a given replica, it is stored and integrated in the replica state using an operational transformation approach. This approach guarantees eventual convergence even when operations are received by different orders in different replicas.

3.1 Reconciliation

For reconciliation, the integration process uses the GOTO operational transformation algorithm proposed in (Sun and Ellis, 1998), but we could use any other operational transformation algorithm that requires TP1 and TP2 (e.g. (Shen and Sun, 2002; Li and Li, 2005; Sun and Sun, 2006)). This algorithm (as all most OT algorithms) transforms the received operation against concurrent operations so that the effect of executing the transformed operations in the current replica state is identical to the effect of executing the operation in the original data state. The algorithm also maintains a log of executed operations for each replica.

This algorithm was originally designed for synchronous environments, where each replica propagates its updates to all other replicas. Thus, this algorithm assumes that each replica receives the original version of the operation.

In our setting, where updates are propagated using an epidemic model, one replica may propagate to another replica an operation received from a third replica. However, as an operation is transformed upon reception and the original algorithm expects to receive the original operation, propagation of third-party operations must be considered carefully. A possible solution would be to store, at each replica, the original and the transformed version for each operation. However, this approach requires additional storage for storing both versions of the operation.

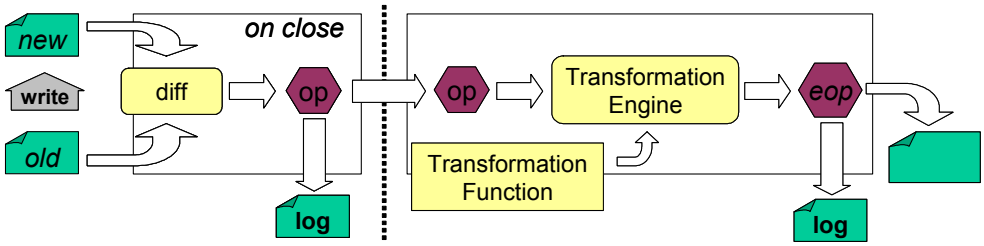

Figure 3: Synchronization process.

We propose an alternative solution, where a replica propagates the transformed version of the operation. To guarantee the correctness of the algorithm, upon reception of the operation in the other replica, the transformation process must take into consideration the transformations already executed. Achieving this is rather simple, as we explain next.

In our solution (as in most OT based solutions), we use version vectors (Parker et al., 1983) to trace causal dependencies for each operation. Additionally, each operation is uniquely identified by the replica identifier and the number of the operation in that replica (this information can be partially included in the version vector, but for simplicity we omit this optimization).

When the OT algorithm transforms the operation A to include the effects of concurrent operation B, the resulting operation A' can be executed after executing B – thus, it is as if the A' had been originally executed after B. Thus, we modify the value of the version vector for A' to include B. When A' is propagated to a third replica, upon reception, it is not considered concurrent with B (due to the new value of the version vector). Thus, as expected, the transformation process will not transform A' to include the effects of B again. It is simple to show that this solution maintains the correctness of the OT algorithm, as propagating a modified version of an operation and modifying the version vector is equivalent to perform part of the transformation process in the original site. We detail our solution in (Bento, 2007).

The proposed technique is not specific to the OT algorithm we are using and can be used to allow epidemic operation dissemination in other OT algorithms. Moreover, this technique minimizes the processing required when receiving an operation, as part of the needed transformation process has already been executed.

3.2 Text File Reconciliation

Our reconciliation solution for text files allows to maintain multiple versions for each line – when two users concurrently modify the same line, two versions of the line are created (as in CVS (Cederqvist et al., 2005)). This approach, besides being widely used in version management systems, seems suitable for asynchronous edition, where update granularity tends to be large and merging two updates performed concurrently to the same line would probably lead to a semantic inconsistency.

Unlike previous reconciliation solutions for text files (including CVS and an OT-based solution for asynchronous edition similar to CVS/RCS (Molli et al., 2003)), our solution considers line versions first-class citizens of the solution. Thus, it allows users to postpone merging multiple versions and continue to modify files with versions (including lines with versions). Additionally, it also allows new updates to be integrated according to users' intentions (unlike previous solutions).

Consider the example in figure 4, where a user at site X modifies line 2 of the file from `B` to `B1` and later to `B3`, and a user at site Y modifies line 2 from `B` to `B2`. In a CVS-like solution adapted for peer-to-peer synchronization (and in solution proposed in (Molli et al., 2003)), in reconciliation step 1, when site Y receives the first update from X, it leads to two versions of line 2 (`[B1 and B2]`). In reconciliation step 2, when Y receives the second update from site X, a new conflict is detected and the previous two versions of line 2 are marked in conflict with the new version of line 2 (leading to an hierarchy with three version of line 2 - `[B3 and [B1 and B2]]`). This shows that those solutions do not correctly consider users' intentions as it seems clear that version `B1` is only a temporary state and it should not be included in the final reconciliation solution - our solution reaches the expected resulted, including only line versions `B2` and `B3` in the final reconciliation result. This would have been the reconciliation result in those systems only if reconcil-

iation step 1 had not been executed.

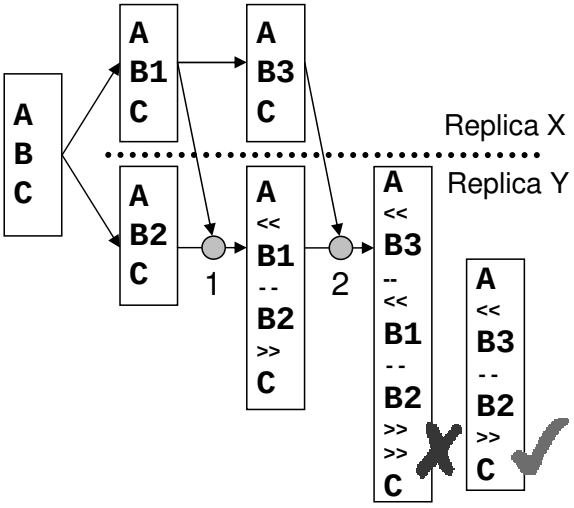

Figure 4: CVS conflict resolution.

In our solution, we model the replicated file as an object containing a sequence of line. Each logical line of text can have several versions. The following example shows a text file where the second line has two versions. Although logically the file has only 4 lines (as shown by the line number in front of each line), the actual number of lines in the file is larger due to the multiple version at line 2 and to the additional lines for marking multiple versions. Each version is identified by a unique identifier stored in the file, as shown in the example.

```
To be, or not to be:  that is the question          1
<<< version 1
Whether 'tis nobler in the mind to suffer          2
--- version 2
Whether 'tis nobler in the mind to joy             2
>>>
The slings and arrows of outrageous fortune,       3
Or to take arms against a sea of troubles,         4
```

In our solution, we have defined the following operations:

- insert(t, p) - inserts a new text line *t* at position *p*;
- delete(p) - deletes the text line at position *p*;
- update(t, p) - updates the content of line *p* with the text *t*;
- create_version(vnew, t, p, vcon) - creates a line version at position *p* with content *t* – the new version *vnew* is originated by an operation with the same identifier, concurrency executed with an operation identified by *vcon*;
- insert_version(vnew, t, p, vdel) - inserts a line version at position *p* with content *t* – the new version *vnew* is originated by an operation with the same identifier, concurrently executed with a delete operation identified by *vdel*;
- update_version(vnew, t, p, vold) - updates a line version at position *p* with content *t* – the version to update is identified by *vold*, and the execution of this operation generates version *vnew*;
- delete_version(vdel, p) - deletes a line version at position *p* – the version to delete is identified by *vdel*.

Besides the usual operations for manipulating lines, we have added operations for explicitly manipulating line versions. Users can manipulate text files as usual (inserting, removing or updating text lines). If a line containing a version is modified, an operation that manipulates versions is inferred. Usually, line version creation is originated by conflict resolution, but users can explicitly create versions by explicitly adding lines for marking line versions.

For using the OT algorithm, we had to define transformation functions for each pair of possible operations. The full set of transformations is a modified version of the functions presented in (Bento, 2007). In this modified set of functions we have adopted the solution proposed in (Oster et al., 2006) for managing line numbers. To this end, for each line in a file we keep the information about the *modified* line number when considering the deleted lines. This information is kept in a auxiliary file that is hidden from common users (by the FEW daemon). The information is this file is updated each time the file is modified, either by a local user or due to the integration of a remote operation.

In this paper we just exemplify our solution by showing how the system resolves an *update/update* and an *update/delete* conflict – see figure 5.

In *update/update* conflicts, the system generates multiple versions of the same line of text by transforming an update operation into a create_version operation. This guarantees that both changes will be presented in each replica as two different version for the given line.

The resolution of *update/delete* conflicts is executed by transforming the *update* operation into a *insert_version* operation, or the *delete* operation into a *create_version* operation. Both operations create two versions for some line, but the *create_version* restores the logical deleted line.

3.3 Opaque Content File Reconciliation

We have also defined two solutions that can be used with files for which the contents are considered

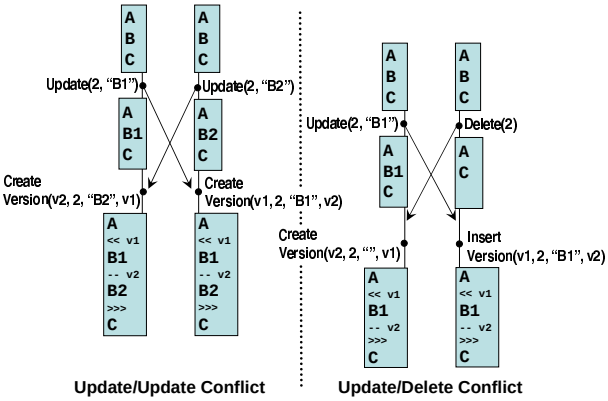

Figure 5: Example of conflict resolution.

opaque. The idea of the first solution is, when the file is concurrently modified, to coherently select a version and keep that version in all replicas. This approach has been adopted in several system, such as the original Lotus Notes, and it is appropriate for some files – e.g. executable files.

To implement this approach, for each file we keep an identifier of the current file version (based on a logical clock). When a file is updated, the corespondent (logical) operation is *update(new_vrs,state)*. The transformation function will transform an *update(v1,s1)* against an already executed *update(v0,s0)* into *update(v1,s1)* if $v0 < v1$ (according to the total order defined combining the logical clock and site identifier (Lamport, 1978)) and *id()* otherwise. This approach guarantees that the last executed *update* operation in every site is the same.

The idea of the second solution is to maintain multiple versions of the entire files. This solution will keep all versions originated by concurrent updates, so that no information is lost due to concurrent updates and that the user can access all concurrent versions when merging them. To implement this approach, for each file, the system internally maintains multiple file version with associated version identifiers. When a file is updated, the corespondent operation is *update(old_vrs,new_vrs,state)*, where *old_vrs* is the identifier of the modified version and *new_vrs* is a new unique identifier for the new version. When this operation is executed, a new file version with identifier *new_vrs* is created and if the file version with identifier *old_vrs* exists, it is deleted.

The way this operation is implemented makes it possible to achieve convergence by executing untransformed operations by causal order. Thus, the transformation function just returns the original operation. It is easy to understand why this approach is correct. The *update(old_vrs,new_vrs,state)* will always create a new file version and, as *new_vrs* is a unique identifier it is not possible for two user to create a file version with the same identifier. Thus, the same set of file versions are created, independently of the execution order. To guarantee that, after executing the same set of operations, the same set of file versions exist, we must guarantee that the same file versions are deleted. As an *update* operation deletes a file version if it exists, if, in the set of executed operations, there is an *update* operations that deletes a given version, that file version must have been deleted. The only ordering of operations that would lead to a different result would be to execute an *update* to a version *v* before executing the *update* that creates that version *v*. However, as operations are executed by causal order, this is not possible.

4 Related Work

In recent years, several systems have been developed for mobile computing environments. In (Tolia et al., 2004), the authors modify the Coda (Satyanarayanan, 2002) file system to improve availability and performance using portable storage devices as *lookaside caches*. The Blue Filesystem (Nightingale and Flinn, 2004) explores the existence of multiple storage devices to improve energy consumption in a client/server architecture similar to Coda. FEW explores the use of portable storage devices with the same objectives and advantages. However, FEW uses a peer-to-peer architecture that requires no single server and allows replicas to synchronize when ad-hoc networks are established.

Personal Raid (Sobti et al., 2002) and Footloose (Paluska et al., 2003) manage files from a single user. Although they address the problem of data stored in multiple devices, they were not designed to support data sharing among multiple users.

Segank (Sobti et al., 2004) also addresses the problem of managing file replicas stored in multiple portable devices. This system support sharing among users, but unlike FEW do not present any solution for conflict resolution – that is delegated to applications. Moreover, unlike FEW the system assumes that all portable devices are always connected, what does not seems reasonable considering the existence of portable storage devices and limited batteries.

Several generic operational transformation algorithms and specific solutions for real-time text editors have been proposed in the past (Ellis and Gibbs, 1989; Sun and Ellis, 1998; Ressel et al., 1996; Sun et al., 1998; Suleiman et al., 1998; Shen and Sun, 2002;

Molli et al., 2003; Li and Li, 2005). Our solution differs from all these solutions by proposing a technique that allows operation to be propagated efficiently using an epidemic propagation model.

Our reconciliation solution for opaque content files is similar to the one used in Coda(Satyanarayanan, 2002) (although the implementation is different) with the difference that we allow users to continue changing versions. Our solution for text files differs from typical solutions in version management system like CVS (Cederqvist et al., 2005) as it allows users to postpone merging of multiple versions and allows the evolution of line versions created during conflict resolution. Being more suited for systems that allow background peer-to-peer synchronization, our approach always allows the integration of new updates received from all users without creating bogus new line version. In this case, our implementation is also very different. An OT based solution similar with CVS has also been proposed (Molli et al., 2003) in the context of a generic file synchronizer. This solution has the same limitations of CVS, thus not guaranteeing the preservation of users' intentions when versions are created.

5 Final Remarks

In this paper we have presented FEW, a file management system for mobile computing environments with portable storage devices. FEW allows files to be shared among users and to be replicated in multiple storage devices, including portable storage devices. The system explores the multiple available replicas to improve freshness, performance and to reduce power consumption.

This system can be used to support collaborative asynchronous edition, as it allows users to asynchronously modify shared data and it includes a reconciliation mechanism for handling concurrent updates. The reconciliation mechanism implemented in the FEW system is based on operational transformation, and it proposes a new set of techniques suited for mobile computing environments, where updates are propagated among replicas using epidemic propagation. It also propose a new solution for handling concurrent updates to text files in an asynchronous setting, that allows multiple line versions to be maintained.

The FEW system is implemented using the traditional file system interface, thus allowing users to continue using their favorite applications. Additionally, as the system propagates relevant file system events, it is possible to provide awareness information while still allowing users to use legacy applications.

In the future, we intend to expand the solution proposed on this paper in several directions. First, we want to create additional type-specific solutions, in particular, for XML files. Second, we need to prove that our transformation functions for text files are correct and we want to look into alternative approach for dealing with the *false tie* problem in systems based on operational transformation. Third, we intend to extend our single line-based solution to a multi-line one, thus supporting the existence of versions of multiple lines.

REFERENCES

Bento, M. A. (2007). Contributions for the design and implementation of a file system for portable devices. Master's thesis, Dep. Informátiica, FCT, Universidade Nova de Lisboa.

Birman, K. P., Hayden, M., Ozkasap, O., Xiao, Z., Budiu, M., and Minsky, Y. (1999). Bimodal multicast. *ACM Trans. Comput. Syst.*, 17(2):41–88.

Cederqvist, P., Pesch, R., et al. (2005). Version management with CVS.

Demers, A., Greene, D., Hauser, C., Irish, W., and Larson, J. (1987). Epidemic algorithms for replicated database maintenance. In *Proc. of the 6th annual ACM PODC*, pages 1–12. ACM Press.

Ellis, C. and Gibbs, S. (1989). Concurrency control in groupware systems. In *Proc. of the 1989 ACM SIGMOD'89*, pages 399–407, NY, USA. ACM Press.

Fitzpatrick, G., Marshall, P., and Phillips, A. (2006). Cvs integration with notification and chat: lightweight software team collaboration. In *CSCW '06: Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*, pages 49–58, New York, NY, USA. ACM Press.

FUSE (2007). http://fuse.sourceforge.net/.

Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565.

Li, R. and Li, D. (2005). A landmark-based transformation approach to concurrency control in group editors. In Pendergast, M., Schmidt, K., Mark, G., and Ackerman, M., editors, *GROUP*, pages 284–293. ACM.

Molli, P., Oster, G., Skaf-Molli, H., and Imine, A. (2003). Using the transformational approach to build a safe and generic data synchronizer. In *Proc. of the ACM SIGGROUP GROUP'03*, pages 212–220, NY, USA. ACM Press.

Nightingale, E. B. and Flinn, J. (2004). Energy-efficiency and storage flexibility in the blue file system. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI '04)*, San Francisco, CA.

Oster, G., Urso, P., Molli, P., and Imine, A. (2006). Tombstone transformation functions for ensuring consistency in collaborative editing systems. In *Proc. of Second International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom 2006)*.

Paluska, J. M., Saff, D., Yeh, T., and Chen, K. (2003). Footloose: A case for physical eventual consistency and selective conflict resolution. In *Proceedings of the 5th IEEE Workshop on Mobile Computing Systems and Applications*. IEEE Computer Society.

Parker, D. S., Popek, G., Rudisin, G., Stoughton, A., Walker, B., Walton, E., Chow, J., Edwards, D., Kiser, S., and Kline, C. (1983). Detection of mutual inconsistency in distributed systems. *IEEE Transactions on Software Engineering*, SE-9(3):240–247.

Ressel, M., Nitsche-Ruhland, D., and Gunzenhäuser, R. (1996). An integrating, transformation-oriented approach to concurrency control and undo in group editors. In *Proc. of CSCW'96*, pages 288–297, NY, USA. ACM Press.

Satyanarayanan, M. (2002). The evolution of coda. *ACM Transactions on Computer Systems (TOCS)*, 20(2):85–124.

Shen, H. and Sun, C. (2002). Flexible notification for collaborative systems. In *Proceedings of the 2002 ACM conference on Computer supported cooperative work*, pages 77–86. ACM Press.

Sobti, S., Garg, N., Zhang, C., Yu, X., Krishnamurthy, A., and Wang, R. Y. (2002). PersonalRAID: Mobile storage for distributed and disconnected computers. In *Proceedings of the Conference on File and Storage Technologies (FAST'02)*, pages 159–174. USENIX Association.

Sobti, S., Garg, N., Zheng, F., Lai, J., Shao, Y., Zhang, C., Ziskind, E., Krishnamurthy, A., and Wang, R. Y. (2004). Segank: A distributed mobile storage system. In *Proc. of FAST'04*, San Francisco, CA.

Suleiman, M., Cart, M., and Ferrié, J. (1998). Concurrent operations in a distributed and mobile collaborative environment. In *Proc. of ICDE '98*, pages 36–45, Washington, DC, USA. IEEE Computer Society.

Sun, C. and Ellis, C. (1998). Operational transformation in real-time group editors: issues, algorithms, and achievements. In *Proc. of CSCW '98*, pages 59–68, NY, USA. ACM Press.

Sun, C., Jia, X., Zhang, Y., Yang, Y., and Chen, D. (1998). Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. *ACM Trans. Comput.-Hum. Interact.*, 5(1):63–108.

Sun, D. and Sun, C. (2006). Operation context and context-based operational transformation. In *CSCW '06: Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*, pages 279–288, New York, NY, USA. ACM Press.

Terry, D., Theimer, M., Petersen, K., Demers, A., Spreitzer, M., and Hauser, C. (1995). Managing update conflicts in bayou, a weakly connected replicated storage system. In *Proc. of the ACM SOSP'95*, pages 172–182, NY, USA. ACM Press.

Tolia, N., Harkes, J., Kozuch, M., and Satyanarayanan, M. (2004). Integrating portable and distributed storage. In *Proc. of FAST'04*, San Francisco, CA.