Reconciliation for Mobile Computing Environments with Portable Storage Devices [*]

Marcos Bento[1], Nuno Preguiça[1], Carlos Baquero[2], and J. Legatheaux Martins[1]

[1] DI, FCT, Universidade Nova de Lisboa
[2] DI, Universidade do Minho

Abstract. 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 to portable storage devices (e.g. flash disks). Many of these devices have large amounts of storage, allowing users to transport most of their data with them. In this paper we briefly present the FEW file management system, a system that aims to ease file management in this new mobile environment. In particular, we detail the automatic reconciliation approach used in this system based on operational transformation. We motivate our work with a study of conflicts in data managed by version control systems.

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.

The Files EveryWhere (FEW) system is a distributed file system that intends to explore the multiple available storage devices to allow users to safely store their data while providing high availability, good performance and low energy consumption. To this end, the system manages files that are automatically replicated in computing devices and portable storage devices. When a replica is modified, the update is asynchronously propagated to all file replicas using a best-effort approach. To guarantee eventual consistency, replicas are synchronized in periodic pairwise epidemic update propagation sessions [2].

An important aspect of our system is its support for data sharing among multiple users. Our optimistic approach may lead to concurrent and conflicting updates. Unlike most file systems with support for mobile computing [8, 6, 13, 10], our reconciliation approach relies on the propagation of updates to files as sequences of operations. This approach has the potential to improve performance [4] and allow better reconciliation results. For reconciliation, we have

decided to use operational transformation techniques. These techniques must be customized for each file type, but allow powerful automatic reconciliation.

In this paper we focus on the reconciliation solution implemented in the FEW system. To motivate the need for advanced reconciliation strategies, we present results from a usage study for cooperative software development projects managed by the (sourceforge) CVS (Concurrent Version System) version control systems. This study shows that unsolved conflicts are much more common in this environment that in distributed file system [6].

The remainder of this paper is organized as follows. Section 2 presents the system model, its architecture and implementation issues of the current prototype. Section 3 presents a study about conflicts in CVS repositories. Section 4 discusses FEW reconciliation approach. Section 5 discusses related work and Section 6 ends the paper with some final remarks.

2   Architecture

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 sets of files, called containers. Containers can be replicated at multiple storage devices and be shared by multiple users. Users can explicitly create new (partial or complete) replicas of a container using the system interface or by simply copying contents between storage devices. A FEW node may automatically create temporary replicas for recently accessed containers in its controlled storage devices if space, performance and energy constraints allow.

The system adopts an optimistic replication approach, allowing users to modify any replica at any time. Whenever a file is modified in a given node, the node propagates the update (or a simple invalidation report depending on the size and connectivity conditions) to other replicas using a best-effort event-dissemination system. Additionally, nodes synchronize in pair-wise background epidemic propagation sessions. These sessions guarantee that all replicas eventually converge (by receiving all updates). In Figure 1 we depict a sample FEW system, comprised of several computing devices with connected portable storage devices.

2.1   File Containers & Namespace Mapping

In FEW, the replication unit is the container, that includes a set of files. Inside containers files are stored in a tree-like hierarchy and have a symbolic name used by users and an internal unique identifier used by the system.

In a FEW node, a *virtual* directory allows access to all container replicas. For example, Figure 2 shows this directory as */few* in a system that contains an
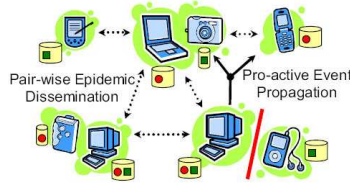
Fig. 1. FEW Architecture

internal storage device (*hda1*) and a portable storage device (*sda1*). Under the directory */few/all*, the user can access all file containers available in the storage devices controlled by the node. For each file accessed under this directory, the system chooses the best replica to retrieve data (considering metrics as replica freshness, energy requirements and access performance).

Users can also access specific file replicas, by accessing files under the appropriate device directory (e.g. */few/sda1* for replicas in the portable device) – the file system calls are redirected to the appropriate location. In either case, applications continue to access the FEW file system using the common file system interface, thus requiring no change to the applications.

2.2   Implementation Aspects

This subsection describes some important aspects of the current implementation. The FEW system is currently under development, but a preliminary prototype with some limitations is already implemented in Linux.

Each node runs a FEW daemon and an interception layer (implemented with FUSE) that redirects file system calls to the daemon – Figure 3. This daemon is organized in a set of modules responsible for the execution of the following main tasks: receive and handle file system calls; manage local replicas; manage information about remote replicas; schedule and execute peer-to-peer synchronization; publish events to remote replicas.
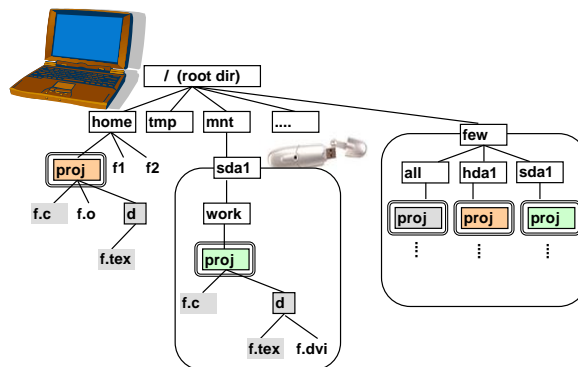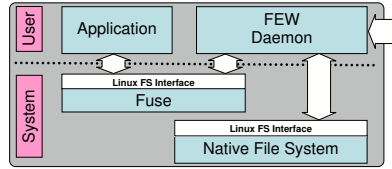

Fig. 2. Filesystem namespace for a computer with a portable storage device.

Fig. 3. FEW Daemon Architecture

Each FEW Daemon manages information about locally connected and other known replicas. This information is maintained by the *Replica Information Manager* module and it is essential in the synchronization process. The mechanism to find remote replicas is still under investigation, but it may use some event-dissemination system. For remote replicas, the following information is maintained: connection status, network latency, known freshness. For replicas controlled by the node, this modules keeps track of transfer rates, energy consumption and available energy and energy mode (when appropriate). For local replicas, the *Container Manager* sub-module additionally maintains information on how to access files in each container, used in the redirection of file system calls to the appropriate storage unit.

The *Log Manager* is responsible for the efficient storage of the logs of updates for each file in the locally controlled storage units. This module also manages the associated information to order and trace dependencies among updates.

The *Data Propagation* propagates updates among container replicas using two mechanisms. First, for the immediate propagation of updates, the system relies on an event-dissemination system. For each container, there is an associated channel that all nodes with a container replica can subscribe. All updates are published in this channel (a simple invalidation report may be delivered depending on the size and connectivity). Second, peer-to-peer epidemic synchronization sessions are established in background to guarantee convergence in all replicas. To this end, the daemon in each replica periodically selects another replica to perform synchronization, based on the information available about other replicas (including information received in invalidation reports).

The *Reconciliation* module performs two main tasks. First, when an application saves some file, this module is responsible to infer the executed updates. These updates are stored locally by the *Log Manager* and propagated to other replicas by the *Data Propagation* module. Second, the reconciliation module executes remote updates using an operational transformation approach that guarantees eventual convergence of all replicas. This process is detailed in Section 4.

3   CVS Usage Study

Usage studies in distributed file systems with support for disconnected operation show minimal file sharing and very rare concurrent updates (e.g. in [6], the authors report 0.025% for the update conflict rate and 0.004% for the unsolved conflicts). As FEW intends to support file sharing among groups of users working

on the same projects/tasks, we have decided to investigate the rate of concurrent updates in similar environments. To this end, we have used real traces from collaborative software projects managed by CVS at http://sourceforge.net.

CVS supports cooperative software development using a central server. For each project, the server maintains a file hierarchy and for each file keeps the current and all past versions. A user checks-out files, modifies his local replicas and then checks-in his new version. As local replicas can be modified concurrently by multiple users, conflicts can occur. A conflict is detected when a user tries to check in a new file version that has been modified by other users since that user has checked-out his version. In this case, the system tries to automatically merge both versions, creating a final version that includes all updates made to different lines. This process fails if there is a *real conflict* (i.e., the same line has been modified by both users). In this case, the user must resolve the conflicts manually before he can check-in his file version.

CVS logs all accesses to a project. We acquired the logs from several projects and obtained the following results: number of updates; number of conflicts (concurrent updates); number of unsolved conflicts (concurrent updates requiring manual intervention) We have selected projects in the most *active list* and other less active projects (for which we could analyze the full log).

These logs had to be processed before statistics could be obtained. First, after solving a conflict manually, a user must check-in the new version, leading to two CVS log entries. Second, we have observed that, after a conflict detection, users sometimes download the current version of the file before checking in a new version. This suggests that users prefer to re-apply their changes to the new version instead of changing the version produced by CVS. In both cases, we count the resulting pair of log entries as one unsolved conflict.

In Figure 4 we present the results for the Gnuplot Project (results are similar for other projects). The CVS log stores a total of 39843 entries, from 13 different users over 795 files, ranging from 15/04/1995 to 28/11/2005. For ease of presentation, each point in the graph presents the average of entries of 20 files. The results show that the number of concurrent updates (conflicts) is much larger than in distributed file systems and vary with the number of updates for each file. In this case, for files with more than 25 updates in the analyzed period, from 10% to 30% correspond to concurrent updates and from 5% to 15% correspond to unsolved conflicts.

These results show that, for supporting data sharing, reconciliation must be considered seriously as a large number of conflicts occur in practice. In FEW, we are addressing this problem by adapting and using automatic reconciliation techniques previously proposed in groupware systems, as we detail next.

4   Reconciliation

Operational transformation (OT) is a reconciliation approach originally designed for synchronous cooperative text editors [11, 3]. The basic idea is to allow different replicas to converge to the same state while applying transformed versions
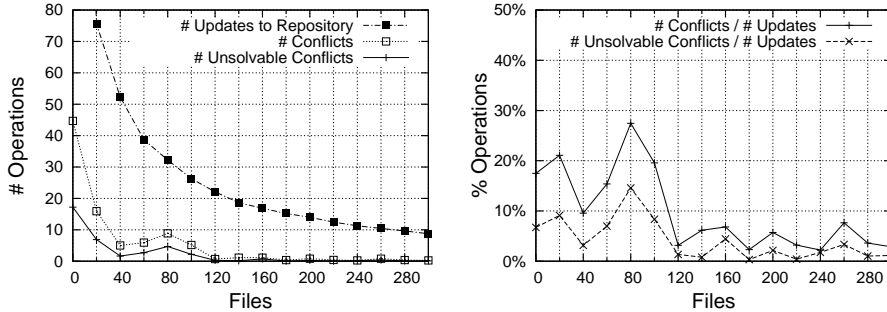
Fig. 4. Gnuplot Project CVS log statistics - absolute values on the left graph and the ratio between solved/unsolved conflicts against the number of updates on the right.

of the operations in each replica by a different orders. For example, if in a text file one user inserts a new second line and a second user concurrently changes the fourth line, both replicas can converge to the same state if in the first replica the fifth line is changed (as the fourth line is now the fifth line by the effect of the insert operation) and in the second replica the new second line is inserted.

Unlike typical solutions in log-based reconciliation (e.g. [12]), this approach allows replicas to converge to the same state while always reflecting all known updates and without requiring operations to be undone. To this end, OT transforms each operation against every other concurrent operation. This allows conflicts to be detected easily by verifying the compatibility of concurrent operations. This approach is more powerful than typical serialization-based reconciliation algorithms where conflicts must be detected only by querying the data state (this approach can also be used in our solution, as the code of operation may include such verification). Additionally, conflict resolution strategies that explore the semantics of operations can be easily implemented - conflict resolution can be executed step-by-step, when an operation is compared with a concurrent operation, instead of requiring conflict resolution to be executed considering a data state that may reflect a large set of concurrent operations.

These interesting properties lead us to explore the possibility of using this approach in our system. Our solution can be divided into two subproblems. The first is the *integration algorithm* (Subsection 4.1), that defines the algorithm to execute operations received from foreign sites. The second, is the definition of the *transformation functions*, used to transform the operations that are executed by the integration algorithm. These type-specific transformations are presented for binary files (Subsection 4.2), and for text files (Subsection 4.3).

4.1   Integration Algorithm

For the integration algorithm, we have based our solution in the GOTO (Generic Operational Transformation Optimized) algorithm [11]. This algorithm requires

the definition of two functions: the *inclusion transformation (IT)* modifies operation O_a against an operation O_b by including the impact of executing O_b before O_a; and the *exclusion transformation (ET)* modifies O_a against causally-precedent operation O_b by excluding the impact of executing O_b before O_a. For guaranteeing convergence, IT and ET functions must satisfy the following two transformation properties [11]: TP1: $O_a \circ T(O_b, O_a) \equiv O_b \circ T(O_a, O_b)$ and TP2: $T(T(T(O, O_a)), T(O_b, O_a)) \equiv T(T(T(O, O_b)), T(O_a, O_b))$.

The GOTO algorithm considers that each site receives the original submitted operation from every other site. However, in a system where replicas are synchronized asynchronously (as needed for supporting disconnected operation and for improving scalability) this is not practical as it would either forbid epidemic dissemination or it would require two versions of each operation to be stored in every site (the original and the transformed one). Thus, we have modified the GOTO algorithm to propagate operations that have already been transformed against some set of other operations. The basic idea is to include this information with the propagated operation in order to reduce the needed transformations in the receiving site [1].

4.2 Files With Unknown Type

For files that the system has no information on the internal structure, we have designed a general solution that creates multiple versions when files are changed concurrently. Unlike Coda [8], we have decided to allow users to continue to change existing versions before executing manual reconciliation (that can simple consist in deleting one of the versions).

To this end, each file version has a unique identifier and an *UpdateContent* of version operation is inferred for each open/write/close session. When this operation is executed at any site, the referred version is deleted (if it exists) and a new version is created (with a unique identifier). Thus, concurrent changes of the same version lead to two versions, as expected. Similarly, it is possible to define an operation that deletes a file version. For these operations, no transformations are needed in IT and ET ($IT(O_a, O_b) = O_a$, $ET(O_a, O_b) = O_a$), as they lead to the expected results if they are executed in causal order (as GOTO guarantees).

4.3 Text Files

For text files, we have devised a specialized solution to merge concurrent updates. Our basic approach is based on a OT implementation of the CVS reconciliation semantics that allows peer-to-peer synchronization and reconciliation. However, unlike CVS, we have decided to consider line versions (created to solve conflicting updates to the same line) first-class citizens and to allow line versions to evolve as the result of user updates without producing additional versions.

For example, considering the updates produced by two users depicted in Figure 5. The user X has produced two consecutive updates to a given file and user Y has produced one update to the same file (note that user Y updates are presented twice in the figure). Using the CVS reconciliation semantics approach,
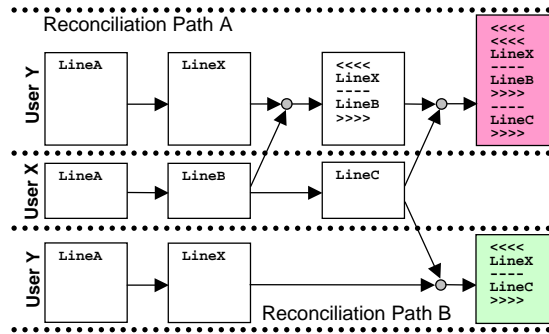
Fig. 5. CVS Conflict Resolution - The way conflicting updates are committed to CVS server influence the final state of the document

two reconciliation results are possible depending on when reconciliation is performed. In reconciliation path A, the user Y reconciles user X twice: the first time after user X has executed his first update, leading to two versions for the single line of the file; and the second time after user X has executed the second update, leading to a new conflict and three possible versions for the file. However, as user X has decided to evolve his own version, it seems reasonable to assume that he no longer wants his first version. Thus, the correct result would be the one presented in reconciliation path B that could be obtained using the CVS semantics if only one reconciliation is executed. In our approach, as we consider line versions first-class citizens, we can reach the expected result independently of the reconciliation path.

To this end, we have considered a text file as a sequence of text lines and we have defined the following basic operations: *InsertTextLine*, *DeleteTextLine* and *UpdateTextLine*. The following additional operations have been defined to manipulate line versions: *CreateTextVersion* (creates a line version with an existing version), *InsertTextVersion* (inserts a version for a line that has been deleted), *UpdateTextVersion* (updates a line version), *DeleteTextVersion* (deletes a line version). When a user closes a file that he has modified, the executed operations are inferred comparing the original and the final file versions (using a simple *diff* algorithm).

Conflicts are solved creating multiple text line versions. To this end, an *UpdateTextLine* operation that conflicts with a concurrent *UpdateTextLine* operation is transformed into a *CreateTextVersion* operation. When an *UpdateTextLine* conflicts with a *DeleteTextLine*, the conflict is solved transforming the *UpdateTextLine* into an *InsertTextVersion* or transforming the *DeleteTextLine* into a *CreateTextVersion*. When two *DeleteTextLine* operations act upon the same line, one of the operations is transformed into a *null* operation. Similar transformations have been defined for operation that manipulate versions.

For now, as it is common in operational transformation algorithms [3, 11], we are considering that an *InsertTextLine* never conflicts with other concurrent operations. This approach allows concurrent updates to *ChangeLog* files to be automatically solved, but it may be semantically incorrect in some situations.

We are investigating the best solution and may introduce the alternative of producing text versions for insertions in the same line.

5   Related Work

In recent years, several systems have been developed for mobile computing environments. In [13], the authors modify the Coda [8] file system to improve availability and performance using portable storage devices as *lookaside caches*. The Blue Filesystem [5] 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 [9] and Footloose [7] 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 [10] 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.

Regarding reconciliation, as we have explained in the previous section, we have introduced several modifications to operational transformation algorithms [11, 3] to adapt them to a peer-to-peer asynchronous setting. Our reconciliation solution for files with unknown type is similar to the one used in Coda (although the implementation is different) with the difference that we allow users to continue changing versions. As explained in the previous section, our solution for text files differs from the typical solution in CVS as it allows the evolution of line versions created for 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.

6   Final Remarks

In this paper we have presented the design of 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.

Besides the description of FEW, this paper presents two main contributions. First, the study of CVS logs showing that, in files shared in cooperative projects, conflicts are much more frequent that reported in distributed file systems environments [6, 8]. Second, the reconciliation mechanism based on OT and including

type-specific solutions for text files and for files with unknown internal structure. This shows that it is possible to adapt OT techniques to file systems and explore the improved reconciliation results obtained with this approach.

References

1. M. Bento. Desenho e implementação de um sistema de ficheiros com suporte para dispositivos de armazenamento portáteis. Msc thesis, FCT - Universidade Nova de Lisboa, May (expected) 2006.
2. A. Demers, D. Greene, C. Hauser, W. Irish, and J. Larson. Epidemic algorithms for replicated database maintenance. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pages 1–12. ACM Press, 1987.
3. A. Imine, P. Molli, G. Oster, and M. Rusinowitch. Achieving convergence with operational transformation in distributed groupware systems. Rapport de Recherche 5188, INRIA, May 2004.
4. Y.-W. Lee, K.-S. Leung, and M. Satyanarayanan. Operation-based update propagation in a mobile file system. In *Proceedings of the USENIX 1999 Annual Technical Conference*, Monterey, CA, USA, Jun 1999.
5. E. B. Nightingale and J. Flinn. Energy-efficiency and storage flexibility in the blue file system. In *Proc. 6th USENIX OSDI*, pages 363–378, San Francisco, CA, Dec. 2004.
6. T. W. Page, Jr., R. G. Guy, J. S. Heidemann, D. H. Ratner, P. L. Reiher, A. Goel, G. H. Kuenning, and G. Popek. Perspectives on optimistically replicated peer-to-peer filing. *Software – Practice and Experience*, 11(1), 1997.
7. J. M. Paluska, D. Saff, T. Yeh, and K. Chen. Footloose: A case for physical eventual consistency and selective conflict resolution. In *5th IEEE Workshop on Mobile Computing Systems and Applications*, pages 170–180, Monterey, CA, USA, October 9–10, 2003.
8. M. Satyanarayanan. The evolution of coda. *ACM Trans. Comput. Syst.*, 20(2):85–124, 2002.
9. S. Sobti, N. Garg, C. Zhang, X. Yu, A. Krishnamurthy, and R. Y. Wang. Personalraid: Mobile storage for distributed and disconnected computers. In *Proceedings of the First Conference on File and Storage Technologies*, January 2002.
10. S. Sobti, N. Garg, F. Zheng, J. Lai, Y. Shao, C. Zhang, E. Ziskind, A. Krishnamurthy, and R. Y. Wang. Segank: A distributed mobile storage system. In *Proceedings of the Third USENIX Conference on File and Storage Technologies (FAST'04)*, San Francisco, CA, March 2004.
11. C. Sun and C. A. Ellis. Operational transformation in real-time group editors: Issues, algorithms, and achievements. In *Computer Supported Cooperative Work*, pages 59–68, 1998.
12. D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *Proc. of the 15th ACM Symposium on Operating systems principles*, pages 172–182. ACM Press, 1995.
13. N. Tolia, J. Harkes, M. Kozuch, and M. Satyanarayanan. Integrating portable and distributed storage. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, pages 227–238, San Francisco, CA, March 31 - April 2, 2004.