Published in the *Proceedings of the The Eighth International Workshop on Collaborative Editing Systems - integrated in ACM CSCW 2006*, 2006.

Operational transformation based reconciliation in the FEW File System*

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

ABSTRACT

Optimistic replication is used to provide high data availability and support for disconnected operation in mobile computing environments. As this approach may lead to concurrent updates, a system based on optimistic replication must include a replica reconciliation mechanism. In this paper, we describe the reconciliation mechanism implemented in the Files EveryWhere (FEW) file system for mobile computing. Our reconciliation approach is based on operational transformation. However, as our system supports only asynchronous collaboration, we propose a set of new techniques for operational transformation. 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. Our approach is implemented as a new file system, thus allowing users to continue using their favorite applications.

Author Keywords

Optimistic replication, reconciliation, operational transformation.

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. Most of these devices have large capacity and can be used to transport users' data.

The Files EveryWhere (FEW) [6] 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, allowing replicated data to be shared amongst multiple users.

The system uses an optimistic replication approach, allowing users to produce their contributions asynchronously and concurrently. These approach may lead to concurrent and conflicting updates. Unlike file systems with support for mobile computing [8, 9, 13], our synchronization process is based on update propagation and includes a generic reconciliation mechanism based on operational transformation [4, 11]. 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. This paper ends with the discussion of related work and some final remarks.

FEW FILE SYSTEM

The FEW system is a distributed file system that manages groups of files, called containers. A container typically stores the data of some (cooperative) project. A container can be replicated at multiple storage devices, including portable storage devices and internal storage units. Data stored in a container can be shared among multiple users. To this end, each user will typically have one (or more) replica of the container in his storage devices.

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. 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 iso-

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

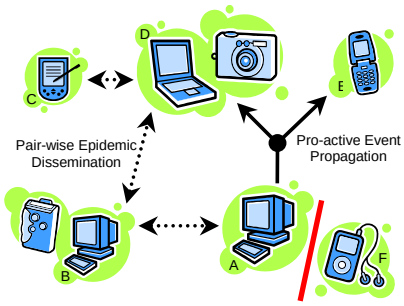

Figure 1. FEW architecture.

lation 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 [3]. In these sessions, replica A propagates to replica B all updates known to A and unknown to B independently of the replica where the update has been performed, and vice-versa. This approach guarantees that all replicas will receive all updates, even if they never communicate directly. In Figure 1, we depict a sample FEW system, consisting of several computing devices with connected portable storage devices.

Using FEW, users may continue to use their favorite applications to modify files as the system is accessible through the traditional file system interface. To this end, our prototype uses FUSE to intercept file system calls and redirect them to the FEW daemon that is responsible to process them (according to FEW functionalities).

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 2.

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. 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 during 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.

Reconciliation

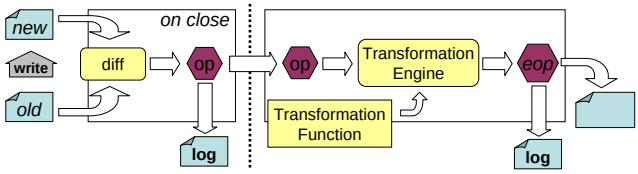

Figure 2. Synchronization process.

For reconciliation, we are using the Generical Operational Transformation Optimized (GOTO) algorithm [11] to control the integration process. This algorithm (as all other 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 GOTO algorithm expects to receive the original operation, propagation of third-party operation 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.

We propose an alternative solution, where a replica propagates the transformed version of the operation. To guarantee the correctness of the GOTO 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 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 GOTO 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 [1].

The proposed technique is not specific to the GOTO algorithm 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.

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 [2]). 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 [5]), 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 3, 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 [5]), 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 reconciliation step 1 had not been executed.

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.

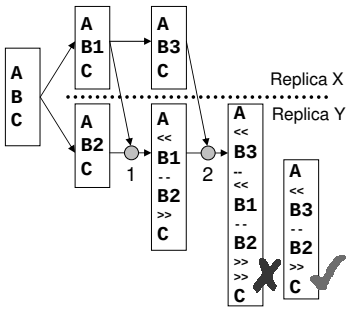

Figure 3. CVS conflict resolution.

```
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, pw) - inserts a new text line *t* at position *p* according to information stored in the position word *pw*;
- 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, concurrency 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 GOTO, we had to define transformation functions for each pair of possible operations. Due to space limitations, we cannot exhaustively present all transformations functions in this paper (they are presented in [1]). We just exemplify our solution by showing how the system resolves an *update/update* and an *update/delete* conflict – see figure 4.

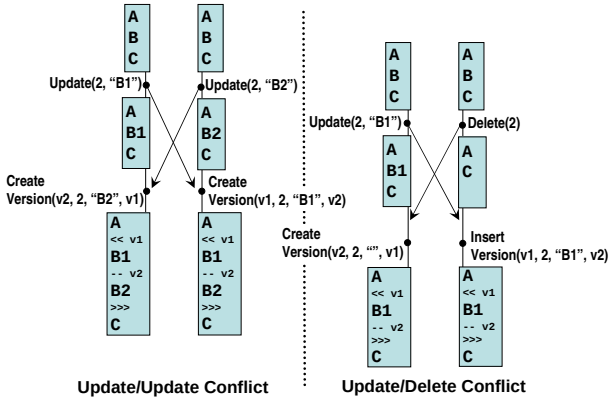

Figure 4. Example of conflict resolution.

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. 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.

Opaque Content File Reconciliation

We have also defined two solutions that can be used with files for which the contents are considered 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 solution is similar to the original Lotus Notes solution and can be used, for example, for executable files.

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.

RELATED WORK

Several generic operational transformation algorithms and specific solutions for real-time text editors have been proposed in the past [4, 7, 10–12]. Our solution differs from all these solutions by proposing a technique that allows operation to be propagated efficiently using an epidemic propagation model.

Our solution for text files differs from typical solutions in version management system like CVS [2] 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 [5] 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.

FINAL REMARKS

This paper presents the reconciliation solution in the FEW system. We propose a new set of operational transformation techniques suited for mobile computing environments, where updates are propagated among replicas using epidemic propagation. We also propose a new solution for handling concurrent updates to text files in an asynchronous setting, that allows multiple line version to be maintained. Our approach is implemented as a new file system, thus allowing users to continue using their favorite applications.

REFERENCES

1. M. Bento. Contributions for the design and implementation of a file system for portable devices. Master's thesis, FCT/UNL, August 2006.
2. P. Cederqvist, R. Pesch, et al. Version management with CVS, 2005.
3. A. Demers, D. Greene, C. Hauser, W. Irish, and J. Larson. Epidemic algorithms for replicated database maintenance. In *Proc. of the 6th annual ACM PODC*, pages 1–12. ACM Press, 1987.
4. C. Ellis and S. Gibbs. Concurrency control in groupware systems. In *Proc. of the 1989 ACM SIGMOD'89*, pages 399–407, NY, USA, 1989. ACM Press.
5. P. Molli, G. Oster, H. Skaf-Molli, and A. Imine. 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, 2003. ACM Press.
6. N. Preguiça, C. Baquero, J. L. Martins, M. Shapiro, P. Almeida, H. Domingos, V. Fonte, and S. Duarte. Few : File management for portable devices. In *Proc. of the IWSSPS'05*, 2005.
7. M. Ressel, D. Nitsche-Ruhland, and R. Gunzenhäuser. An integrating, transformation-oriented approach to concurrency control and undo in group editors. In *Proc. of CSCW'96*, pages 288–297, NY, USA, 1996. ACM Press.
8. M. Satyanarayanan. The evolution of coda. *ACM Transactions on Computer Systems (TOCS)*, 20(2):85–124, 2002.
9. 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 *Proc. of FAST'04*, San Francisco, CA, March 2004.
10. M. Suleiman, M. Cart, and J. Ferrié. Concurrent operations in a distributed and mobile collaborative environment. In *Proc. of ICDE '98*, pages 36–45, Washington, DC, USA, 1998. IEEE Computer Society.
11. C. Sun and C. Ellis. Operational transformation in real-time group editors: issues, algorithms, and achievements. In *Proc. of CSCW '98*, pages 59–68, NY, USA, 1998. ACM Press.
12. C. Sun, X. Jia, Y. Zhang, Y. Yang, and D. Chen. Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. *ACM Trans. Comput.-Hum. Interact.*, 5(1):63–108, 1998.
13. N. Tolia, J. Harkes, M. Kozuch, and M. Satyanarayanan. Integrating portable and distributed storage. In *Proc. of FAST'04*, San Francisco, CA, March 2004.