FEW: File Management for Portable Devices*

Nuno Preguiça, U. Nova de Lisboa
Carlos Baquero, U. Minho
J.Legatheaux Martins, U. Nova de Lisboa
Marc Shapiro, Microsoft Research
Paulo Sérgio Almeida, U. Minho
Henrique Domingos, U. Nova de Lisboa
Victor Fonte, U. Minho
Sérgio Duarte, U. Nova de Lisboa

Abstract

In recent years, an increasing number of portable devices with large amounts of storage have become widely used. In this paper, we present the early design of the FEW system, a system that aims to ease file management in the new mobile environment. To this end, the system will manage file replicas stored in fixed and portable storage devices. It will provide an automatic mechanism to establish new file replicas by analyzing file system activity. The system will automatically and incrementally synchronize all file replicas exploring the available network connectivity and the availability of portable storage devices. To merge concurrent updates, operational transformation techniques will be used.

1. Introduction

In recent years, an increasing number of portable devices have become widely used. Laptops and PDAs have been joined by a myriad of new computing devices such as mobile phones, digital media players, portable storage devices (e.g. flash key rings/clocks), digital cameras, etc. Most of these devices have large amounts of storage, allowing users to carry gigabytes of data at any time. Unlike old portable storage (such as floppy disks, CD-ROMs, etc.), these devices tend to have performance and reliability comparable to hard disks. Moreover, some of these devices have intrinsic computing and communication capabilities, allowing them to act as computing devices as well as sophisticated, networked based storage devices.

The emergence of these new devices is creating a new mobile computing environment with characteristics that represents an important depart from assumptions taken in older mobile data management systems [19, 23]. In this paper, we present the early design and main research ideas of the Files EveryWhere (FEW) system, a system that we are currently developing to provide file management in this new computing environment.

FEW is a distributed file system that manages files stored in computing devices and portable storage devices. To improve availability, groups of related files can be replicated in multiple storage devices. Temporary file replicas of recently used files are automatically created in portable storage devices. Permanent file replicas can be easily created by users by copying files between storage devices.

In order to support mobility and ad hoc file access and update, the system uses a read-any-write-any model of data access. When accessing files in mobile computers, the system selects which of the available replicas to access based on the freshness, performance and energy characteristics of the multiple storage devices. When modifying a file, a single replica is changed. Additionally, invalidation reports are asynchronously sent to all other replicas using a best-effort approach.

File replicas are synchronized in pairwise epidemic update propagation sessions [5]. These sessions are triggered by invalidation reports and executed periodically to guarantee eventual convergence of all replicas even if invalidation reports are lost.

During synchronization sessions, update operations are propagated. To this end, when files are modified and closed, the system automatically computes file differences and logs correspondent operations. Reconciliation is based on operational transformation. The system will include automatic conflict resolution solutions for three file types: line-based text files (as in CVS/RCS [3]), XML text files and binary files. Type-specific solutions for other file types may be defined by programmers. Besides allowing peer-to-peer synchronization, our approach minimizes data propagated in synchronization sessions.

The remainder of this paper is organized as follows. Section 2 presents the system design. Section 3 discusses some issues related with replica management. Section 4 discusses synchronization and reconciliation. Section 5 discusses related work and section 6 concludes the paper with some fi-

* This work is partially supported by FCT/MCES – POSI/FEDER project #59064/2004.

nal remarks.

2. System Design

The FEW system comprises a set of participating machines. In each machine there is a set of internal storage units and a set of portable storage units (in portable devices, such as flash disks) that may vary over time. Portable storage units are either disconnected or controlled by a single machine. Storage units corresponds to what is normally called a disk partition.

Using FEW, a set of files (contained inside a storage unit) may be replicated in multiple units, possibly in distinct machines. Each machine, runs a file system intersection layer and a FEW server that is responsible for maintaining replicas synchronized. File replicas stored in portable storage devices are synchronized when connected to a machine running FEW.

2.1. File containers

The unit of replication is the file container. A file container contains a set of files, possibly a single one, and it has a unique system wide identity. Inside a file container, files have unique identifiers in a flat namespace. Additionally, files have names in a tree-based namespace within each container.

Each machine hosts a directory tree composed of one of more mounted storage units. In each machine's directory tree, a container replica has a base directory that may differ from machine to machine. However, any given container replica must reside in a single storage unit (naturally, containers are replicated across many different units).

In figure 1, we present an example of a container replicated in a portable storage device and in the internal hard drive of some computer. The base directory of the container is */mnt/sda1/work/proj* in the portable storage device and */home/proj* in the internal hard disk of the computer. The files that belong to the container have a grey background. As depicted, in each device, some additional files (*f.o*, *f.dvi*) are stored in the same tree hierarchy, but are not replicated across devices.

FEW will provide tools for explicit management of containers and container replicas. For example, a user may create a container based on the contents of a directory (and its subdirectories), create a local replica of a container from any existing container in a reachable FEW machine or add files to a container.

However, to minimize administrative costs, we provide automatic mechanisms for container (and container replica) creation and retirement. By default, when a user copies a set of files between two storage units[1], a container is created

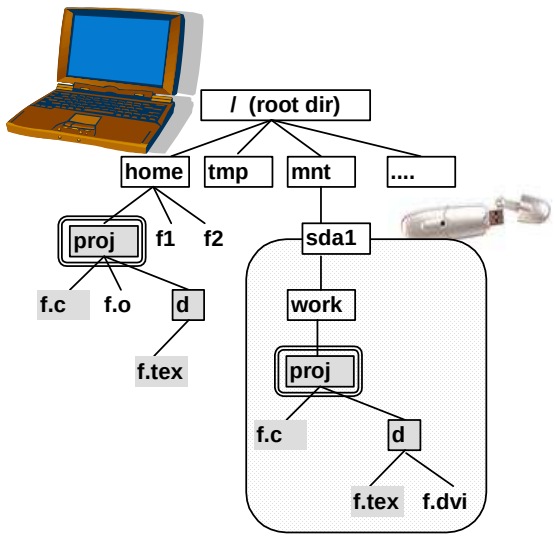

Figure 1. Filesystem namespace for a computer with a portable storage device. A single container is replicated in both the portable storage device (base directory: /mnt/sda1/work/proj) and in the internal hard disk (base directory: /home/proj).

with all the transferred files and each copy of the container is considered a container replica. If the files that are being copied already belong to a container, a container replica is created in the destination unit. Replica retirement is also automatically inferred when all files contained in a container are deleted.

When a container is created and after its creation, additional files may be added to the container based on information automatically extracted from access traces and by rules specified by the users (a set of pre-defined rules is specified in FEW). For example, a rule may specify that when creating a container with a file with ".cc" extension, all files with ".h" and ".cc" extensions in the same directory should also be added to the container. Moreover, it will be possible to automatically add to a container a file that is found to be closely related to a file already in the container, based on access traces. This solution is similar to cache hoarding for supporting disconnected operation and combines and extends ideas proposed in Coda [8] and SEER [9].

A container replica may only contain the data of a subset of the files in the container. However, a container replica always has knowledge about all files in the container (not

1 When the copy is executed between local and remote storage units mounted on the machines' directory structure, it is easy to identify such occurrence. However, we intend to identify copies executed from/to remote file systems relying on external file-transfer programs such as *scp* by allowing FEW servers to exchange information about recently accessed files.

necessarily in a strict consistency sense). When a user lists the files in the directory that contains a container replica, all files in the container are listed. When a user accesses a file that is not currently replicated in the given unit, FEW downloads the file contents from a nearby container replica. When accessing a file replicated locally, FEW may also download the file contents from a nearby replica to improve freshness, performance or reduce energy-usage. Note that freshness improvement is closely related with file synchronization and will be discussed later.

As container replicas are typically scattered along different machines, it is necessary to provide a mechanism to locate them and efficiently manage communication among them. To this end, FEW servers establish an overlay network, on top of which, a multicast dissemination group is set for each replicated container. Each FEW server should join all multicast groups for the containers replicated locally (either on fixed or portable storage units). Location queries are propagated using this mechanism. We intend to build this binding and dissemination facility using an existent system, such as Scribe [2]. As most existing user level multicast overlays have been designed for stationary, well connected computers, we will complement it with features geared towards the support of the mobile branches of the dissemination trees.

FEW automatically creates temporary container replicas. When a user accesses a container replica in a portable storage unit, FEW creates a temporary container replica in a resident unit. Conversely, the same happens when accessing replicas in resident units. All temporary container replicas are stored in a well-known directory of the target units and may be directly accessed by users.

2.2. Container synchronization

When a user modifies a file in a file container, that single replica is written. Additionally, FEW creates an invalidation report that is asynchronously sent to all servers containing container replicas using a best-effort policy. Invalidation reports for container replicas that are not currently available (either because they are stored in an unreachable computer or in a disconnected portable storage device) are lost. Upon reconnection of a container replica or periodically, consistency is checked using large granularity techniques [12].

Synchronization of container replicas occur during pairwise epidemic propagation sessions. When an invalidation report is received, the server schedules a new synchronization session for some time in the near future. Additionally, synchronization sessions are scheduled periodically until all reachable replicas converge to the same state. To this end, each container replica collects and maintains large granularity information about the state of all other replicas.

FEW adopts a log propagation approach, where update operations are propagated during synchronization sessions. Update logs are automatically created when users modify and close files, using type-specific algorithms. Conflicts may occur as FEW uses a read-any-write-any model of data access and supports disconnected operation. Conflict resolution is solved using operational transformation techniques detailed later.

In order to accommodate heterogeneous communication facilities and an incremental style of synchronization, FEW will order the logs to be propagated by system and user relevance. Replica management related events and directory update logs will have priority. File update logs will be sorted by file ranking. File ranking will be automatically computed based on previous read / write access patterns. Synchronization will proceed in background as connectivity allows. User access to local replicas will not be prevented by ongoing synchronization and, when the user opens a file, the system will try to optimize freshness and energy consumption. If needed, file synchronization will be completed before the return of the open call.

We are building FEW using a stackable file system (using FiST [25]) that allows to intercept all file system operations. This approach is necessary to implement automatic container (and container replica) creation. Moreover, it allows a directory to contain files belonging and not belonging to a container replica.

An additional feature that we want to support is the possibility to allow users to modify file replicas in machines that do not run FEW. Assuming that users do not modify files that maintain FEW-specific meta-data, that are stored by the FEW stackable filesystem as hidden files, it seems that it is only necessary to verify the validity of the meta-data when a filesystem is mounted under the control of FEW. External modifications can be detected by storing file digests in the FEW meta-data. However, when a file is modified outside the control of FEW, the log of update operations is not available. To compute it, FEW needs to access the old and new file version. To this end, we intend to evaluate two possibilities: maintaining an additional copy of each file in the storage unit or obtaining it from other replicas.

3. Issues in replication management

FEW explores connectivity in order to limit replica divergence, giving to each user the most up to date vision of the state in reachable replicas. This is subject to tunable constraints regarding energy and bandwidth consumption when these factors impose a non neglectable cost.

In a given machine, both external and self-imposed limitations will limit its horizon of connectivity. Its horizon includes passive storage units currently mounted in the ma-

chine and remote storage units mediated by other machines in its reachable network.

In FEW, replica creation and retirement is not constrained by small horizons. In order to create new replicas it is sufficient to access an existing replica. By having access to an existing replica one has access to both the identity of the replicated entity, and to the state and version of that replica.

In fact, only the identity of the replicated entity is strictly necessary for new replica creation, since these replicas can be assigned an empty state and an appropriate version id. The version id is such that this isolated replica will be dominated by any other replica (getting its state upon synchronization) or made concurrent with all replicas if written before the first contact with other replicas.

Since in FEW we are operating over file systems we can distinguish two types of replicated entities, file containers and files. Recall that file replicas will always be associated to a container replica, and containers can be partially replicated. Although file replica identity and version can be related to its container envelope's identity and version, both container replicas and individual file replicas require a version control mechanism that traces their evolution under optimistic operation.

In general, the version id of each replica will represent, in a compact way, the causal history of update events that have been incorporated in the replica state. The high degree of autonomy that is allowed for replica creation, precludes the use of version vectors for causal history characterization as they depend on global replica identifiers. Autonomous version tracking can be achieved by recursive techniques [1] for id assignment. These techniques operate under permanent partitioning and ensure a correct characterization of data causality among active replicas.

Alternative approaches to autonomous version tracking imply probabilistic techniques whose merits with respect to deterministic approaches are still in debate. The development of FEW will help to shed some insight over the practical implications of both techniques.

Replica retirement is an opportunity for the reduction of version identifiers that need to be tracked on the system. Although some replicas can be out of contact for very long periods, and possibly lost, this should not impact the correctness of version tracking. Consequently, only detected retirements will lead to reductions on version tracking load.

4. Synchronization and Reconciliation

FEW adopts an optimistic replication strategy with a read-any-write-any model of data access. This approach allows to modify a container replica without accessing any other container replica. In mobile environments, a traditional usage scenario that requires this property is the support of operation in disconnected devices. Additionally, in our environment, where container replicas may reside in portable storage devices that are disconnected from computing devices for long periods of time, the previous property is important to allow users to access a container replica in a connected devices when it is impossible to contact all (or most) of container replicas (because they reside in disconnected portable devices). Finally, our approach is important to support data sharing and collaborative work styles. For example, two users may want to concurrently modify different parts of the same document, relying on automatic reconciliation mechanisms to merge their changes (as they would do today relying, for example, on CVS to merge changes).

In FEW, replicas are synchronized during pairwise epidemic propagation sessions. In each synchronization session, each replica propagates the set (or a subset) of updates unknown to the other replica. This approach allows peer-to-peer replica synchronization exploring available network connectivity to propagate updates from and to portable storage devices when these devices are connected to a computer.

The optimistic replication approach adopted may lead to data divergence. To handle existing conflicts, we have decided to adopt a reconciliation mechanism based on operational transformation [6]. To this end, updates are propagated as operations. We detail our approach in the next subsections.

4.1. Inferring the update log

Previous works [11, 4] have shown that operation-based update propagation may be more efficient than state-based update propagation. Moreover, during reconciliation, the extra semantic information encoded in the operations may be used to provide better solutions. These reasons lead us to the use of operation-based update propagation.

Adopting operation-based update propagation in a file system has a preliminary problem: how to log semantic-rich operations when applications use the traditional file system interface (reading and writing sequences of bytes). To this end, previous systems have decided to log user commands in the interactive shell [11] or the raw user activity at the application level [4]. To obtain the same final state in a different computer, the log of operations has to be replayed in a system with the same software. An additional mechanism is provided to verify that the replay leads to the same state.

In FEW we have decided to adopt a different strategy. The log of (semantic-rich) operations is inferred when an application closes a file. In this moment, the system automatically calls a type-specific application that must infer the executed operations by comparing the initial and final file state. This application returns the sequence of opera-

tions that will be logged and later propagated to other replicas. As in other log-based data management systems, a log-compression algorithm will run to compress the operations in the log, thus minimizing the information that is locally stored and that it is propagated to other replicas.

FEW will include the following type-specific solutions. For line-based text files (as considered in systems like RCS/CVS), we use traditional algorithms for computing differences in text files as the basis to create the sequence of executed operations (insert/remove/replace line). For XML text file, we intend to use XML-specific algorithms for computing differences (e.g. [10]). As before, specific operations to change the document structure and elements' contents will be generated based on these differences. Finally, for files of unknown types, an operation that sets a new state to a file is created (with the contents of the new file). As a *set state* operation overwrites causally-dependent *set state* operations, consecutive write to the same file are encoded as a single *set state* operation. In this case, operation based update propagation is similar to state-based update propagation.

For other file types, users may register in a container, type-specific applications to infer operations and provide the needed semantic information for the log-compression algorithm. Additionally, we intend to provide an alternative file system interface to allow applications specially designed for FEW to explicitly modify a file using a type-specific interface.

4.2. Executing the logged operations

When a sequence of operations is received in some replica, the operations are locally stored and immediately executed in the local file replica using operational transformation (OT) techniques [6, 22]. The basic idea behind OT is to modify the original operation in a way that its execution in a different state leads to the same expected results. To this end, OT algorithms need to transform each operation against all concurrent operations.

For example, suppose that user A adds line 5 to some text file in replica R_1 and user B adds line 7 to the same text file in replica R_2. When the operation of user A is propagated to replica R_2 (that already contains the new line 7), it can be applied without being changed, thus merging the operations of the two users. However, when the operation of user B is propagated to replica R_1, the position of line 7 has been shifted by the insertion of line 5. Therefore, to obtain the same expected result, it is necessary to modify the operation that inserts line 7 to insert the new line at position 8 instead. This is achieved by transforming the operation executed by user B against the operation executed by user A.

This approach allows replicas to eventually converge while being synchronized using a peer-to-peer communication model. Additionally, each replica always reflects all known updates.

OT is mostly useful for preserving users intentions when concurrent updates can be automatically merged (i.e. when updates do not conflict). However, OT also allows to easily detect real conflicts, as each update is transformed against all concurrent updates. Conflict resolution strategies can be implemented by defining specific transformation rules for conflicting operations.

In FEW we will adopt the following pre-defined conflict resolution strategies. For line-based text files, OT rules will create multiple versions of each line. Multiple versions will be encoded in the text file using RCS/CVS conventions. For XML text files, OT rules will also create multiple versions of concurrently modified elements. We are still investigating the best way to encode the multiple versions. For files of unknown type, two solution will be available. The first solution creates multiple versions of the file. This solution should be used with files directly edited by users. The basis to achieve this property is to coherently transform some *set state* operations into *create file version* operations whenever there is a conflict. The second solution will keep a single version. The basis to achieve this property is to transform *set state* operations that do not correspond to the selected version into *void* operations. As before, users may define and register OT rules for additional file types.

To perform OT, each replica will maintain a log containing the operations recently executed. Additionally, for each file, it will exist one replica, named the golden replica, that maintains the complete log of operations. Thus, this replica allows to create all file versions and perform all needed OT. Whenever possible, this replica should be stored in a highly available host.

Using this approach, garbage collection of the log is quite simple. For each file, the information about updates known in the golden replica is propagated to all replicas during synchronization sessions. A replica may delete all updates that are known in the golden replica. Usually, a replica will not immediately delete recent updates as they are necessary to locally perform OT and to perform peer-to-peer synchronization.

5. Related Work

Several data management systems for *traditional* mobile computing environments have been developed relying on optimistic replication (see [18] for a recent survey). Coda [19] is a distributed file system that enables mobile file access through its support for disconnected and weakly connected operation. Bayou [23] is a replicated database system, where users may specify application-specific conflict detection and resolution rules with each update.

Although supporting data access in mobile computing environments, these systems were not designed to face the problems and to exploit the opportunities opened by the increasing number and capacity of portable storage devices that are now available. Recently, several systems have been designed for this new environment.

In [24], the authors extend the Coda system by using the data stored in portable storage devices to improve performance (as a lookaside cache). Blue File System (BFS) [13] is a client/server file system that provides energy-efficiency by using a flexible cache hierarchy that includes data stored in portable devices. Unlike these systems, FEW intends to handle portable storage devices as first-class citizens, managing replication of shared data in all devices and allowing peer-to-peer synchronization.

PersonalRAID [20] and Footlose [14] are systems designed to manage data stored in devices from a single user. Although their architecture is different, they both synchronize by using one-to-one interactions without relying on a central server. Unlike these systems, FEW intends to manage data that can be shared by multiple users, which imposes additional problems for guaranteeing freshness and consistency, as data can be modified concurrently by multiple users.

In Segank [21], a user always has a portable device (MOAD) that maintains information about the most recent version of an object, based on the user's activity. The system assumes that all devices are always connected to the network. When accessing an object, the system searches the most recent version, based on the information stored on the MOAD. In FEW, we intend to address the disconnection of portable devices, as it seems an obvious consequence of the limited battery, connection costs or usage model of some devices (e.g. flash keyrings).

In our system we will also use and improve techniques previously developed for data management in mobile as well as stationary environments, namely: creation and deletion of file replicas to provide high availability and good performance [17, 7], epidemic update propagation [5], update propagation based on operations logs [23], version identifiers allowing the computation of relations among replicas [15, 1] and the use of semantic information and operations transformation to reconciliate concurrent versions of data [6, 22, 16].

FEW, as compared to previous systems geared towards the exploitation of portable storage, will try to integrate in a coherent set the following characteristics and functionalities: multiuser, disconnected operation, easy and transparent replica creation and deletion, incremental P2P replica synchronization and automatic reconciliation of concurrent replicas.

6. Final remarks

In this paper we have presented the early design of the FEW system, a file management system for mobile computing environments that include portable storage devices. In FEW, files are grouped in containers and containers may be replicated in multiple storage devices. The system will explore the multiple available replicas to improve freshness, performance and to reduce power consumption. To ease the administrative cost related with replica management, we will provide an automatic mechanism to group files in containers and to create a new replica by simply copying files between two different file systems.

Containers are synchronized during peer-to-peer interactions, exploring available network connectivity and the availability of portable storage devices. Unlike previous systems, our approach will infer the log of executed updates by comparing file states and use use operational transformation techniques to merge concurrent updates. The system will use operational reconciliation techniques to reconcile concurrent updates. Pre-defined solutions for line-based text files, XML text files and files of unknown types will be provided. Type-specific solution may be added to the system.

We are currently in the early stages of system implementation. Concurrently, we are still refining the system design.

References

[1] P. S. Almeida, C. Baquero, and V. Fonte. Version stamps - decentralized version vectors. In *Proc. of the 22nd International Conference on Distributed Computing Systems (ICDCS'02)*, page 544. IEEE Computer Society, July 2002.

[2] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron. Scribe: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communication (JSAC)*, 20(8), Oct. 2002.

[3] P. Cederqvist, R. Pesch, et al. Version management with CVS, date unknown. `http://www.cvshome.org/docs/manual` .

[4] T.-Y. Chang, A. Velayutham, and R. Sivakumar. Mimic: raw activity shipping for file synchronization in mobile file systems. In *MobiSYS '04: Proceedings of the 2nd international conference on Mobile systems, applications, and services*, pages 165–176. ACM Press, 2004.

[5] 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.

[6] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. In *Proceedings of the 1989 ACM SIGMOD international conference on Management of data*, pages 399–407. ACM Press, 1989.

[7] M. Kim, L. P. Cox, and B. D. Noble. Safety, visibility, and performance in a wide-area file system. In *Proceedings of*

the First USENIX Conference on File and Storage Technologies (FAST'02), pages 131–144. USENIX Association, Jan. 2002.

[8] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. *ACM Transactions on Computer Systems (TOCS)*, 10(1):3–25, 1992.

[9] G. H. Kuenning and G. J. Popek. Automated hoarding for mobile computers. In *Proc. of the* 16^{th} *ACM Symposium on Operating Systems Principles*, pages 264–275. ACM Press, 1997.

[10] K.-H. Lee, Y.-C. Choy, and S.-B. Cho. An efficient algorithm to compute differences between structured documents. *IEEE Transactions on Knowledge and Data Engineering*, 16(8):965–979, 2004.

[11] 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.

[12] L. B. Mummert and M. Satyanarayanan. Large granularity cache coherence for intermittent connectivity. In *Proceedings of the USENIX Summer Conference*, pages 279–289, June 1994.

[13] E. B. Nightingale and J. Flinn. 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, December 2004.

[14] J. M. Paluska, D. Saff, T. Yeh, and K. Chen. 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, Oct. 2003.

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

[16] N. Preguiça, M. Shapiro, and C. Matheson. Efficient semantics-aware reconciliation for optimistic write sharing. Technical Report MSR-TR-2002-52, Microsoft Research, Cambridge (UK), May 2002.

[17] Y. Saito, C. Karamanolis, M. Karlsson, and M. Mahalingam. Taming aggressive replication in the pangaea wide-area file system. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, Massachusetts, December 2002.

[18] Y. Saito and M. Shapiro. Replication: Optimistic approaches. Technical Report HPL-2002-33, Hewlett-Packard Laboratories, Mar. 2002.

[19] M. Satyanarayanan. The evolution of coda. *ACM Trans. Comput. Syst.*, 20(2):85–124, 2002.

[20] 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 Conference on File and Storage Technologies*, pages 159–174. USENIX Association, 2002.

[21] S. Sobti, N. Garg, F. Zheng, J. Lai, Y. Shao, C. Zhang, E. Ziskind, A. Krishnamurthy, and R. Wang. Segank: A distributed mobile storage system. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST '04)*. USENIX Association, Mar. 2004.

[22] C. Sun and C. Ellis. Operational transformation in real-time group editors: issues, algorithms, and achievements. In *Proceedings of the 1998 ACM conference on Computer supported cooperative work*, pages 59–68. ACM Press, 1998.

[23] 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* 15^{th} *ACM Symposium on Operating systems principles*, pages 172–182. ACM Press, 1995.

[24] 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 (FAST '04)*. USENIX Association, Mar. 2004.

[25] E. Zadok and J. Nieh. FiST: A Language for Stackable File Systems. In *Proceedings of the Annual USENIX Technical Conference*, pages 55–70, June 2000.