SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2009; **00**:1–18 *Prepared using* speauth.cls *[Version: 2002/09/23 v2.2]*

User-level operating system transactions

Diomidis Spinellis[1]*

[1] *Department Management Science and Technology, Athens University of Economics and Business, Greece. email: dds@aueb.gr*

SUMMARY

User-level operating system transactions allow system administrators and ordinary users to perform a sequence of file operations and then commit them as a group, or abort them without leaving any trace behind. Such a facility can aid many system administration and software development tasks. The snapshot isolation concurrency control mechanism allows transactions to be implemented without locking individual system calls; conflicts are detected when the transaction is ready to commit. Along these lines we have implemented a user-space transaction monitor that is based on ZFS snapshots and a file system event monitor. Transactions are committed through a robust and efficient algorithm that merges the operations performed on a file system's clone back to its parent. Both the performance impact and the implementation cost of the transaction monitor we describe are fairly small.

KEY WORDS: User-level transaction; snapshot; ZFS; snapshot isolation concurrency control

1. Introduction

Complex operations consisting of interrelated steps often benefit from being organized as a *transaction*: an atomic sequence of operations that can succeed or fail as a whole. Transactions already play an important role in a diverse set of fields, such as applications of database systems [1, 2], the implementation of file systems [3, 4], the recording of changes in version control systems [5], and, recently, the memory access in multi-core processors [6].

There are many ways in which a system administrator or a software developer end-user could benefit from an operating system's transaction support. First, consider complex software installations. Many types of software, such as MediaWiki, Ruby on Rails, and Cyrus IMAP, have very complex installation

Software: Practice & Experience, 39(14):1215–1233, September 2009. (doi:10.1002/spe.935).
This is a machine-readable rendering of a working paper draft that led to a publication. The publication should always be cited in preference to this draft using the reference in the previous footnote. This material is presented to ensure timely dissemination of scholarly and technical work. Copyright and all rights therein are retained by authors or by other copyright holders. All persons copying this information are expected to adhere to the terms and constraints invoked by each author's copyright. In most cases, these works may not be reposted without the explicit permission of the copyright holder.

Copyright © 2009 John Wiley & Sons, Ltd.

requirements with tens of dependencies on (often specific versions of) other components. Making a mistake or encountering a bug on such an installation can render a system unusable. Although many installation systems offer an uninstall facility, this seldom provides the assurance of undoing all the installation's side-effects. Furthermore, performing a major upgrade, for instance from the Apache web server version 1.3 to 2.2, requires the careful manual adjustment of many configuration files. Again, mistakes can bring down a service for many hours.

Finally, software developer end-users, although they are increasingly putting their everyday work under a version control system (VCS) [7], often perform small (often experimental) steps within a larger change. Typically programmers, in order to minimize churn and noise, do not wish to put such small changes under revision control, but they still want to retain the ability to undo their changes.

System administrators currently employ a number of alternatives to address the situations we have described, but each has some disadvantages. Business-critical installations often operate a production and a test system. Changes are first introduced in the test environment, verified, and then brought over to the production system. However, as the performed changes often have to be re-applied to the production system by hand, there is always the possibility that an error will creep in.

Another approach involves the use of virtual machine technology. The production virtual machine is cloned, the changes are applied on the clone, and then the clone switches roles with the production machine. This approach fails to work when the production system continuously writes data to files or databases during its operation. In such a case, data written in the clone's parent between the original cloning time and the later switchover time would be lost. Similar problems occur when using file system snapshots or backup utilities, such as Apple's time machine: changes unrelated to the problematic rolled-back change can be lost. If, for instance, an administrator in order to undo a failed upgrade rolls back a file system to a previous snapshot, work done by end-users on that file system since the snapshot's time will be lost. Furthermore, the switch between a clone and its parent can be disruptive. For instance, under ZFS promoting a file system's clone to its parent requires unmounting the file systems and therefore interrupting the operation of the production system.

System administrators can also put all files of the production environment under version control [8]. This setup offers the possibility of rolling back a failed change. However, because system administrators will perform changes in steps and not in an atomic operation, the production system will be unstable during its maintenance. A more sophisticated way involves using a VCS to keep a production and test environment synchronized. This approach has the advantage of leveraging the VCS's mature heuristics for resolving conflicting changes. The downside is that VCSs typically do not deal gracefully with large binary files. For instance, under such a scenario each commit in the production environment would store in the VCS repository a complete copy of the system's database files.

Facilities currently being introduced in mass-market operating systems allow the efficient provision of transactions at an operating system's user level. In this paper we explore one design approach using file system snapshots and event monitoring. The main contributions of the work reported here are

- an overview of the design and implementation space for user-level transactions,
- the adoption of the snapshot isolation concurrency control mechanism from the database community [9] as an efficient design choice,
- the proposal to use file system snapshots and event monitors as tools for implementing user-level transactions, and

Copyright © 2009 John Wiley & Sons, Ltd.

Prepared using speauth.cls

- the design and implementation of a robust algorithm for merging the operations performed on a file system's clone back to its parent (see Section 4.5).

Compared to other work, our approach minimizes the transaction monitor's load by not monitoring read operations and by having the file system clone facility perform all intermediate write operations. For many workload scenarios this can result in higher efficiency.

2. Transactions as User-Level Entities

Transactions in computing systems are defined in terms of the properties of atomicity (a transaction succeeds or fails as a whole), consistency (the transaction leaves a consistent store consistent), isolation (a transaction's effects are not visible to other transactions until it commits), and durability (after a commit the transaction's effects persist, even if system crashes) [1, 10].

The design space we explore seeks to address mainly atomicity, and, to a lesser degree, isolation at the user space: an application's view of the system. In common with database systems, we look at consistency at the application level and assume that it is the user's responsibility (for instance an application will not be left in a partially installed state). Furthermore, we consider durability a responsibility of the file system, the installation's UPS, and graceful shutdown procedures. Others [11, 12] have taken a lower-level view, looking at the consistency and durability of a file system as properties that can be guaranteed by transactions.

Transaction conflicts can be detected at various levels of precision. For instance, we can define as a conflict when two separate transactions modify the same file, or we could be more precise and look for overlaps between the lines changed by the two transactions, as is the practice in many VCSs. In addition, we can take into account or ignore the side-effects of a transaction's changes, such as the updating of a directory's modification time when a new file is created in it. To match the profile of the problems we seek to address we decided to work at the level of files. For typical scenarios, conflicting changes are either rare (e.g. simultaneous installations of packages that all change the installed package manifest) or irrelevant (e.g. writes to the same cache file). Furthermore, correctly handing such conflicts requires tricky and error-prone heuristics.

We decided to ignore second-order changes to file metadata, such as directory modification times and file access times. For aborted transactions this metadata will not be affected, but for committed transactions many directory modification and file access times will differ from their original time, ending up close to the transaction's commit time. Handling such changes as part of a transaction would significantly increase the chance of conflicts, without any important corresponding gains in the system's utility. Moreover, the handling of such changes would make the system more difficult to implement. In contrast to file modification times, which are used by tools such as *make* [13] in order to track build artifact dependencies, few if any tools depend on perfectly accurate file access and directory modification times. In fact, in many systems these times are routinely changed outside the end-user's control by systems like file name and content indexers.

An alternative approach would be to consider changes to file system blocks, instead of files. This was our initial approach, because it matched closely the way file system snapshots and clones are implemented using copy on write mechanisms. Implementing it would require extensive file system–specific changes to a system's kernel. In addition, under such an approach it would be difficult to report

Copyright © 2009 John Wiley & Sons, Ltd.

to the user the conflicts in a meaningful way. Once we realized that user-level transactions could be efficiently implemented (mostly) through user-space processes tracking files, we adopted the file-level approach as an elegant, modest-effort implementation choice that would be worth exploring.

In database systems isolation can be achieved through a variety of approaches: two-phase locking, the association of read and write timestamps with each object, the maintenance of multiple versions of each object, or through the adoption of an optimistic concurrency control.

Two-phase locking involves a phase where locks are acquired but not released, and a second phase where locks are released, but no new locks are acquired. This scheme does not fit our purpose, because it requires modifications in existing applications to follow its protocol, it adds a locking overhead to each operating system operation, and it can also lead to deadlocks.

A timestamp-based approach has the advantage of being able to run incrementally. However, it requires monitoring both read and write operations, which can be an expensive proposition. This can be overcome by maintaining multiple versions for each object, each with its own timestamps, thus providing the advantage that reads are never blocked.

Optimistic concurrency control matches the profile of the typical use cases we envision, and can also offer the advantage of allowing the user to consolidate conflicting changes. On the other hand, it requires an expensive validation step, which has to be based on recording the read and write times for all actions. An alternative approach involves the case where user involvement can be tolerated, and each time only a single transaction is allowed to execute concurrently with other operations performed outside its scope. In such a case the validation step can be replaced by a three-way-merge between the data: in its original form, as modified by the transaction, and as modified by other changes that have occurred outside the transaction.

In the end, what guided our design was the fact that ZFS supports the efficient creation of live snapshots of a file system, named clones, and that *Fsevents*, the event monitoring framework we chose (see Section 4.1), monitors changes but not reads. We thus chose to base our design on these technologies, running a variant of the multi-version concurrency control mechanism called in the database world *snapshot isolation* [9].

3. System Design

Our design is based on a daemon that controls all transactions and a command-line utility that interfaces with the daemon to perform the following actions.

Begin a new transaction. The user specifies as parameters the file systems that are to be put under transaction control. The tool reports back the transaction's unique identifier and the names of file system clones (live snapshots of the specified file systems) where the operations of the transaction should be executed.

Commit an existing transaction. The daemon verifies that the changes made by the transaction do not conflict with the changes made by other committed transactions, or changes made by processes running in parallel with the transaction. If conflicts are detected, they are reported, and the transaction is aborted. Otherwise, the changes in the file system clones are merged back to their parent (the file systems from which the clones were derived), and the clones are destroyed. After a commit the transaction cannot be rolled back.

Copyright © 2009 John Wiley & Sons, Ltd.

Abort a specified transaction, or all transactions. An abort can only be performed on transactions that have not yet been committed. The daemon simply destroys the corresponding file system clones. Non-root users can abort a specific transaction, or all transactions they have initiated, while root can also abort all transactions. When multiple transactions are aborted, an informative message is displayed on the terminal from which each transaction was initiated.

List active transactions, events, and monitored file systems. This facility can be used for debugging the daemon's operation.

The transaction monitoring daemon accepts requests for the execution of user-initiated commands, and also monitors changes performed on all file systems to detect conflicts. It maintains a global list of objects (absolute file names) that are modified while transactions are in progress, as well as state data and a modification log related to each transaction.

When a transaction begins, the monitor first obtains exclusive access to the file systems that will be placed under transaction control. It then creates a snapshot of those file systems, and subsequently a clone: an exact duplicate of the file system's data that is mounted under the file system's root and can be independently modified. The monitor also timestamps the transaction's start time, and, if no other transactions are running at the time, starts to monitor file system events (otherwise, monitoring is already taking place). Finally, it releases the exclusive access it obtained and reports back the transaction's unique identifier.

The monitoring of file system events is used for two purposes: to detect conflicts between various transactions, and to merge in an efficient way a transaction's changes into the file system from which it was cloned. When a new event arrives, the transaction monitor determines whether the event is associated with a file system clone, (i.e. part of an executing transaction), with a corresponding parent, or with a file system that is not under transaction control. Events of the third category can be safely ignored. File systems from which transactions start, i.e. parent file systems, notionally operate under what is called in the database world as *auto-commit* mode. This means that changes to them are immediately considered to be committed, since there is no transaction running on them that will commit at a later point. Therefore, if an event is associated with a parent file system, the object associated with the event is immediately marked with a commit timestamp of the current time. On the other hand, events associated with a file system cloned for a transaction simply associate the object with the corresponding transaction, without however setting the object's commit time. Furthermore, these events are also stored in the transaction's log, so that they can be replayed when a transaction commits. Note here that although a clone and its parent file system are mounted under differing paths, the objects associated with them are normalized to always refer to the name of the parent file system. Some events, like that occurring after a rename operation, are associated with two objects; in such a case the transaction housekeeping is performed for each object. An event can also signal that the kernel was unable to deliver all events to the transaction monitor. If this happens the transaction monitor cannot function reliably, and will therefore abort all transactions.

Aborting a transaction is simply a cleanup operation. The monitor will first verify the user credentials, so that only a transaction's owner or root can abort it. It will then clear the data associated with the transaction, like file systems that required monitoring and objects that were not committed, and destroy the snapshots and clones associated with it. If at this point no other transactions are running, event monitoring can also cease.

Copyright © 2009 John Wiley & Sons, Ltd.

Prepared using speauth.cls

Table I. Representative transactions in action.

Time	User Action	Transaction Logic	File System Changes
12	transaction begin	transaction id ← 1239; start_time ← 12	create clone-1239
13	touch a	commit_time[a] ← 13 (auto-commit)	
14	touch clone-1239/b	commit_time[b] ← 0 (pending commit)	
15	transaction begin	transaction id ← 1240; start_time ← 15	create clone-1240
16	transaction commit 1239	OK (commit_time[b] < start_time)	copy clone-1239/b b
		commit_time[b] ← 16	destroy clone-1239
17	touch clone-1240/b	commit_time[b] = 16 (unchanged, pending commit)	
18	touch c	commit_time[c] ← 18 (auto-commit)	
19	transaction begin	transaction id ← 1241; start_time ← 19	create clone-1241
20	transaction commit 1240	Conflict (commit_time[b] > start_time)	destroy clone-1240
21	touch clone-1241/c	commit_time[c] = 18 (unchanged, pending commit)	
22	transaction commit 1241	OK (commit_time[c] < start_time)	copy clone-1241/c c
		commit_time[c] ← 22	destroy clone-1241
23	transaction begin	transaction id ← 1242; start_time ← 23	create clone-1242
24	touch clone-1242/c	commit_time[c] = 22 (unchanged, pending commit)	
25	touch c	commit_time[c] ← 25 (auto-commit)	
26	transaction commit 1242	Conflict (commit_time[c] > start_time)	destroy clone-1242

The committing of a transaction is the most delicate operation. After the user's credentials are verified, the transaction monitor obtains again an exclusive lock on the file systems associated with the transaction. It then sets the transaction's commit time timestamp and checks for conflicts using the *snapshot isolation* multi-version concurrency control algorithm. Specifically, for each object that the transaction modified, the monitor checks if the object's commit time is larger than the transaction's start time. If this is the case, a conflict has arisen and the transaction is aborted. (In section 7 we describe that these occurrences can be gracefully handled by allowing the user to explicitly deal with conflicts.) Otherwise, the transaction can safely commit. The use of the multi-version concurrency control algorithm allows for the efficient detection of conflicts and a fast merging of changes. The actual commit involves setting the commit time of all objects that the transaction modified to the transaction's start time, and replaying the event log associated with the transaction on its parent file system. Finally, exclusive access to the file systems can be released, and the cleanup operation we described on abort can be performed.

Table I shows some cases of successful and failed transactions. The command *cmd* is assumed to run in the same directory, modifying various files; the directories *clone-nnnn* are the clones created for the corresponding transaction. Transaction **1239** succeeds, and is allowed to commit, because the change in the transaction's clone file system and in its parent affect two different files (*a* and *b*). When it

Copyright © 2009 John Wiley & Sons, Ltd.
Prepared using speauth.cls

commits, the file it modified (*b*) gets timestamped with the transaction's commit time (16). Transaction **1240** is not allowed to commit, because at time step 17 it also modifies a file named *b*. At commit time it is found that file *b* is timestamped after the transaction started. Transaction **1241** modifies file *c* (at time step 21) that is also modified outside the transaction (at time step 18). However, because the timestamp of *c* is earlier than the transaction's start time the transaction is allowed to commit (the changed version of *c* was already included in the transaction's cloned file system). Finally, transaction **1242** gives rise to a conflict, because the transaction modifies file *c* at time step 24, and at time step 25 the file is also modified outside the transaction while it is executing. Thus, at its end the commit time of *c* is greater than the transaction's start time.

4. Prototype Implementation

By shopping around for an operating system platform that could readily provide most of the facilities we needed, we put together a research prototype of the system with modest implementation effort. The system runs under Mac OS X modified with the installation of writable ZFS functionality. It uses the ZFS snapshots and clones, and also relies heavily on Apple's *Fsevents* file system event functionality [14, Section 11.8.2]. The system comprises less than 2000 lines of code. The command-line interface, the daemon startup code, and the file system event monitoring are written in C, while the transaction processing logic is written in C++. The robust and efficient data structures provided by the C++ standard template library (*set*, *map*, *string*, *vector*) greatly simplified the implementation of the transaction processing logic.

The transaction processing monitor is implemented as a single-process, single-threaded server. A *select* call is used to multiplex between the monitoring of events and the processing of user-sent commands. Avoiding multiple processes or threads ensures the integrity of the monitor's (non-trivial) data structures and algorithms. Although this implementation limits the monitor's throughput, we do not think that this will be a problem in practice, because we envision transactions to be few large-grained operations.

4.1. Event Monitoring

From the two facilities required for implementing our systems, a versatile event monitoring mechanism turned out to be the most difficult to locate. File system clones as implemented by Sun's ZFS [15] are currently available under the Solaris, FreeBSD, Mac OS X (as a downloadable module), and Linux (under FUSE) operating systems. On the other hand, the file system monitoring facilities available under Linux and FreeBSD (the *inotify* and *kevent* interfaces) could not statisfy our needs, because they require registering every directory for which changes are monitored. A simple experiment showed that simply locating all available directories in a typical *usr* partition in the most efficient manner could take three minutes, and therefore these interfaces were unsuitable for our purpose. Furthermore, at the time of writing the file event monitoring facility of Solaris (FEM) lacked a userland API. Using DTrace [16] was not an option, because it might silently drop events. Finally, we feared that the cost of injecting a monitor for all file-system calls, using the *ptrace* mechanism [17, pp. 129–131] would be prohibitively intrusive and expensive. This left Apple's *Fsevents* facility and Mac OS X as the only contender.

Copyright © 2009 John Wiley & Sons, Ltd.

Prepared using speauth.cls

Fsevents is a file system event monitoring framework that is part of Apple's Spotlight: a system that allows the efficient harvesting and searching of file metadata. Under *Fsevents* one can register an event monitor through a special file (*/dev/fsevents*). From that point onward the kernel will send to the monitor packets describing the operations and arguments for any performed file changes. The arguments include the file's path name as well as the corresponding device and inode numbers. As we shall see these turned out to be crucial for the system's implementation. The facility's official documentation is minimal, but having access to the operating system's source code worked out as a perfectly adequate alternative.[*]

As the kernel uses a fixed-size buffer for storing packets, it is important for applications using the monitor to constantly retrieve all events. When the buffer fills, the operating system will coalesce events and finally also drop them. Dropped events are signalled to the monitoring applications through a special event type, and applications receiving such an event will not take the issue lightly. Our transaction monitor will abort all executing transactions, while Spotlight will reportedly rescan all disks [14, p. 1420]. In order to minimize the chance of missing events, we turnoff event monitoring while a transaction is committing. A commit can take considerable time, during which the transaction monitor cannot process file change events. However, during a commit the file systems on which transactions are executing are locked, and therefore event monitoring can be safely turned off.

4.2. ZFS Interface

Interfacing with ZFS for creating clones and snapshots is performed by invoking the corresponding ZFS commands. On our system these commands can sometimes take more than a second to complete, especially if a file system is stored in flash memory. To minimize the impact of this delay, we let the commands that destroy the snapshot after a transaction run in the background. Unfortunately, the command for setting up a clone must be executed synchronously, and therefore the user experiences a small delay when initiating a transaction.

Another issue concerned the mapping between the ZFS dataset names, which the user specifies as the targets for a transaction, and the integer device identifier of their corresponding mount points, which the system receives as part of the file system change events. Fortunately, ZFS provides a command (*zfs get -H -o value mountpoint* datasetname) for programs to query values and obtain the results in a format that is easy to parse. We used this to create the corresponding map.

4.3. File System Monitoring

As we described in Section 3, file system events are divided among those that belong to a transaction's file system, those that belong to its parent, and those that can be ignored. Making this distinction based on a file's name can be very difficult, if one takes into account the various ways in which file systems can be mounted. Luckily, ZFS allocates a different device identifier (*dev_t*) for each file system and

[*]After implementing the transaction monitor we came across another facility, Apple's file system events API (`http://developer.apple.com/documentation/Darwin/Conceptual/FSEvents_ProgGuide/FSEvents_ProgGuide.pdf`). It is unclear whether using this API instead of directly tapping into the event monitor file would have been a better implementation choice.

Copyright © 2009 John Wiley & Sons, Ltd.

clone. We were thus able to implement an efficient map from a file system's identifier to the type of monitoring required for that file system.

4.4. File System Locking

Although our system requires the ability to obtain exclusive access to file systems when a transaction begins and ends, Mac OS X does not provide this functionality, and we did not implement it due to lack of support for a standalone Darwin kernel derivative.[†] Lack of file system locking means that, under our current implementation, conflicting changes performed while a transaction is committing will not be detected. In practice we do not think that this will be a significant problem: conflicts will be rare and the value of transactions will mainly lie in the ability to abort flawed changes and commit successful ones.

Nevertheless, the implementation of this facility seems to be straightforward. It has already been described and implemented in the context of performing snapshots in the 4.4 BSD FFS [18, Section 6], [17, p. 350]. This involves suspending activity on the corresponding file system (by calling *vfs_write_suspend*), and allowing system calls writing to the file system to finish. The facility is implemented by inserting a gate at the top of each system call that can write to the file system. The gate counts the processes using the file system, and can also be closed to suspend them while a transaction is in a critical region. All that is missing is a system call (e.g. *fslock*) that will take as arguments an array of file system names, its length, and a flag of the required operation (lock or unlock). When implementing *fslock*, care must be taken to avoid deadlocks; allowing only one *fslock* call to execute at a time seems to be a reasonable approach.

4.5. Event Replay

Replaying a transaction's event log on its parent file system is not trivial. The complexity arises from the fact that for the sake of efficiency we store in the event log only the metadata of each operation (operation type, file path, file type), and not the actual data (file contents, permissions, owners, extended attributes, and ACLs). Instead, we rely on the availability of this information on the cloned file system to copy it to its parent when it is required. Unfortunately, simply accessing a file's contents through its name is not always possible because subsequent rename operations may change the name under which its contents are stored.

Initially we attempted to solve this problem by associating with each file's unique identifier (the device and inode number pair) its updated path. A first pass through the event log would setup this map, which could then be used for accessing a file's contents through its identifier. This scheme is relatively efficient; for a transaction comprising N events of which M are rename operations and a map stored in a data structure with a retrieval cost $O(\log N)$ the algorithmic cost of replaying the event log is $O(N+(N+M)\log N)$. However, although this scheme handles correctly the renaming of a file path's last component (the file name), it will fail when an intermediate part of the path is renamed or moved.

[†] http://web.archive.org/web/20070409155747/http://www.opendarwin.org/en/news/shutdown.html

Copyright © 2009 John Wiley & Sons, Ltd.

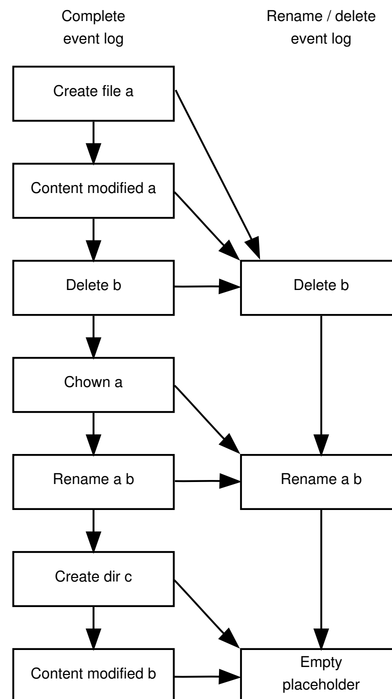

Figure 1. Example of the full and compressed event logs.

To address this deficiency, we implemented an alternative solution, based on a rename log. This is a list of all rename operations performed during the course of the transaction. When the event log is replayed in order to transfer the effects of the transaction from the clone to its parent, each source file name is first transformed by applying to it all recorded rename operations. Note that the target file name does not need to be transformed, because it will be correctly renamed or moved when the corresponding events are processed on the parent file system. For a transaction comprising N events of which M are rename operations, the algorithmic cost of replaying the event log is $O(N + N \times M)$. This approach also failed, when the same (typically temporary) file name is reused and renamed multiple times during the course of a transaction.

In the end, we had each event follow the name's rename operations from that event's start until the end of the event log (or the file's deletion). Normally, the cost of this operation is $O(N(N-1)/2)$, which proved expensive: in a transaction comprising half a million events the daemon spent 76 CPU minutes in user mode, most of it presumably recalculating file names. We therefore optimized the search by keeping in a separate list only the renames and deletes, and associating with each event the corresponding start position of the list (see Figure 1). Because the future first rename or delete operation

Copyright © 2009 John Wiley & Sons, Ltd.

to be associated with an event is not known when an event is processed, the list always contains at its end a dummy placeholder event. Once a rename or delete event is encountered, the placeholder is updated to match the contents of that event, and a new placeholder is appended to the list. Assuming that the M rename operations are uniformly distributed among the N events, the total cost of calculating file names during an event replay is

$$O\left(\frac{N}{M} \times \frac{M(M-1)}{2}\right) = O\left(\frac{N(M-1)}{2}\right).$$

The cost of this revised scheme is not prohibitive, because rename operations are typically only a small percentage of the total number of operations, and I/O is likely to dominate the cost of merging the committed files to a clone's parent directory.

A final stumbling block for the implementation of the event log replay seemed to be the correct transfer of all the file's metadata, including permissions, extended attributes, access control lists, and finder data, from the clone file system to its parent. Fortunately, Mac OS X provides a *copyfile(3)* library function, which is designed to handle all these cases. Although our design does not depend on the existence of the *copyfile* function, its availability simplified the implementation.

4.6. Interprocess Communication

The transaction monitoring daemon communicates with the command-line control tool through a Unix domain socket. This allows the command-line tool to pass reliably to the daemon the user's credentials and a descriptor associated with its standard error stream using the socket's out-of-band message communication facility. The user's credentials are stored with each associated transaction, and ensure that only its user (or root) can commit or abort it. They are also used for setting the daemon's effective user id, when replaying the event log to each parent file system. One could argue that this is not required, because the daemon is playing back events that have already succeeded, and, as there were no conflicting events, one could assume that the user will have sufficient privileges to execute the corresponding actions. This however ignores situations where during the course of a transaction the permissions of a directory change.

The file descriptor associated with the command-line tool's standard error stream is also stored as part of a transaction's data. This is used to notify the transaction's owner when the transaction is forcibly terminated by the administrator or due to a system error. In such cases the corresponding ZFS clones are destroyed, and therefore subsequent system calls of the transaction's processes related to its file systems will fail; emitting a message with the underlying cause is a reasonable user interface design choice.

4.7. Data Structures and Resource Management

The system's data structure schema is illustrated in Figure 2. Each transaction maintains a list of associated events: operations on files located in the cloned file system. When the transaction commits, these are replayed (as described in Section 4.5) to replicate the cloned file system on its parent.

Transactions are also associated with transacted objects: files and directories that are modified by a transaction. These objects are always identified by the name of the corresponding object in the parent file system. They can come to life either when a transaction modifies an object in a cloned file system,

Copyright © 2009 John Wiley & Sons, Ltd.

Prepared using speauth.cls

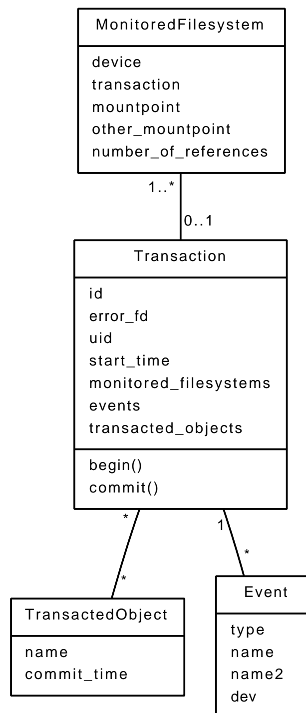

Figure 2. Logic schema of the maintained data.

or when another process modifies an object on a cloned file system's parent; such a modification is immediately auto-committed. Every transaction maintains a set of the objects it has modified. A transacted object can be associated with more than one transaction, but the link from the object to the transaction is not maintained. Instead, the object's commit time is used to detect conflicts during a transaction's commit. Transacted objects are garbage collected when all transactions commit.

Finally, the system also maintains a bidirectional link between transactions and the file systems that each transaction monitors. Every transaction needs to monitor one or more cloned file systems. The link from the cloned monitored file systems to the corresponding transaction is used by the event monitor to update the transaction with the transacted objects associated with each event. In addition, the system also monitors the corresponding parent file systems. This is used by the event monitor to auto-commit objects that are modified in those file systems.

Monitoring a file system is an expensive operation, because it burdens the transaction monitor with some processing for every operation performed on that file system. A map between device numbers and monitored file systems allows the event monitor to efficiently determine when an event requires

Copyright © 2009 John Wiley & Sons, Ltd.

Prepared using speauth.cls

further processing. In addition, each monitored file system maintains a reference count. When this count drops to zero the object is deleted, and monitoring on the file system ceases. When no file systems are monitored (i.e. when no transactions are in progress), the event monitor is completely disabled, thus zeroing the CPU load of a quiescent transaction monitor.

4.8. Limitations

As implemented, the command-line tool provides a relatively low-level interface, allowing a user to perform transacted operations on file system clones. Ideally, we would want a transaction to transparently provide a complete environment where operations can take place in isolation, and later committed or aborted. This may involve performing a *chroot* on a specially crafted file system that includes the appropriate clones, or setting up an application container through a *zone* (under Solaris) or *jail* (under FreeBSD). Additional complications of a more comprehensive approach include a scheme for specifying which users can use transactions (currently we allow any user to start a transaction), and the handling of changes that are not mirrored in a file system (e.g. *settimeofday*).

Currently the system doesn't perform the requisite file system locking (see Section 4.4), and also fails to handle correctly hard links, mirroring them to the parent file system as a copy of the linked file. The transaction monitor sees hard links as file creation events for a specific inode. As it is very expensive to map from an inode to a file name, and the system does not provide a system call to directly link a file name to a given inode, a more faithful implementation of hard links requires an operating system extension.

Finally, the transaction monitor is not integrated with the operating system startup and shutdown operations. The requirements for addressing this deficiency boil down to prohibiting a shutdown while a transaction is committing, aborting all active transactions when a shutdown commences, and cleanning up file system clones left by an ungraceful shutdown. If a system is equipped with a UPS, so that it will not shut down on a power failure while a transaction is committing, satisfying these requirements will address the transaction *durability* property.

5. Performance Evaluation

To measure the performance of our system we used the SSH workload [19] and the PostMark synthetic benchmark [20]. In common with the tests performed by the developers of the Amino transaction system [12] we used version 4.2p1 of the SSH package, and we split the workload into three phases: *unpack*, which involves decompressing and untarring the distribution file, *configure*, which involves running the configuration script, and *make*, which involves compiling the package. Running PostMark with the configuration used in Amino was not feasible, because the rapid sequence of I/O operations resulted in dropped events. We therefore used a configuration of file sizes 5–100k, a read and write block size of 64k, 2500 simultaneous files, and 9000 operations (transactions in the benchmark's terminology). To test a more substantial workload, we also run an unpacking (*tar xzf*) of the Linux kernel version 2.6.28, and the unpacking (*tar xzf*), configuration (*configure*), build (*make*), test (*make test-force*), and installation (*make install*) steps on the version 5.1.30 source distribution of MySQL.

Copyright © 2009 John Wiley & Sons, Ltd.

Prepared using speauth.cls

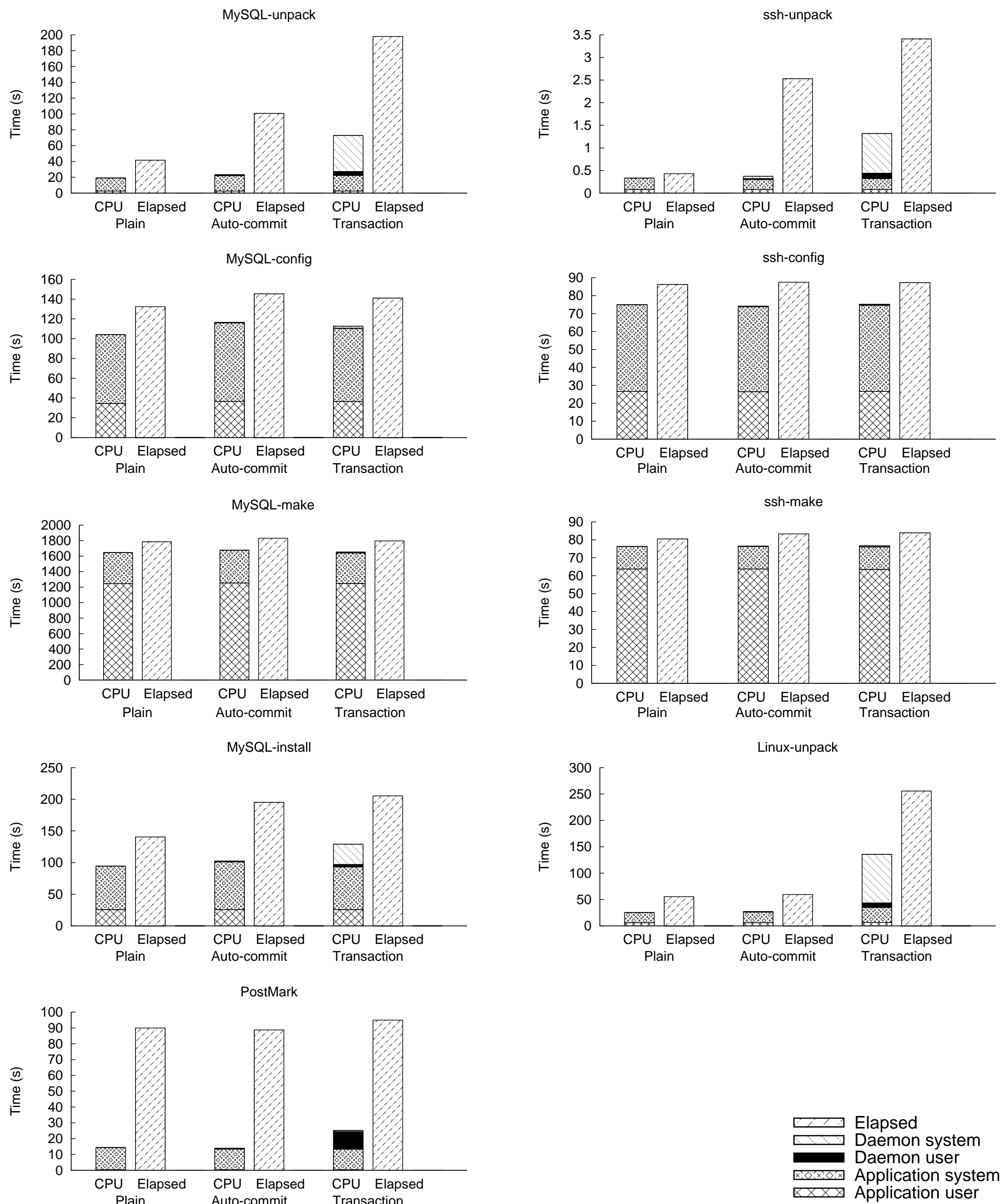

Figure 3. Time performance measurements.

Copyright © 2009 John Wiley & Sons, Ltd.

Prepared using speauth.cls

Table II. Benchmark configuration.

Item	Description
Computer	Mac mini model A1103
CPU	PowerPC 7447a (G4)
CPU clock speed	1.25 GHz
L2 cache	512kB (on chip)
RAM	1GB 333 MHz DDR SDRAM
External disk	Seagate model ST3320620A
Interface	Ultra ATA 100
Cache	16MB
Average latency	4.16 ms
Spindle speed	7200 RPM
Heads	4
Disks	2
Bytes per sector	512
Disk enclosure	IB-351U-B-BL
Interface	USB 2.0
Kernel version	9.6.0 (xnu-1228.9.59)
ZFS version	119

We run the benchmarks on an (otherwise idle) Apple Mac mini with an external hard disk formatted as a single ZFS volume. The configuration's details appear in Table II. We run each benchmark in three setups.

Plain involves running the benchmark on the ZFS volume without any transaction overhead. This establishes the base case for the configuration's performance.

Auto-commit has the benchmark running on a ZFS directory on which another (empty) transaction is running. The benchmark's operations are auto-committed to the directory immediately as they occur. This measurement establishes the overhead that the transaction monitor imposes on a system's regular operations.

Transaction has the benchmark running on a ZFS file system clone, and then its results copied back to the clone's parent file system. This measurement establishes the overhead of performing the operation as a transaction, which can be committed or aborted.

We unmounted the ZFS volume between each measurement to clear its cache.

The benchmark time results appear in Figure 3. As we can see, the *unpack* and *install* operations is where the system's overhead is most pronounced. This happens because these operations mainly create

Copyright © 2009 John Wiley & Sons, Ltd.

Table III. Memory performance measurements.

Operation	Events	Objects	Overhead (MB)
MySQL-unpack (A/C)	0	1,133	1
MySQL-make	8,826	4,264	1
PostMark	25,546	6,994	2
MySQL-install	39,480	4,507	4
MySQL-unpack	6,426	1,133	5
Linux-unpack	107,384	26,846	13

new files, which then have to be copied again from the file system clone back to its parent. This fact is substantiated by the small percentage of daemon user time. In the CPU-intensive *config* and *make* operations the daemon's overhead is barely perceptible, thus showing the practicality of our approach for various typical workloads. Furthermore, the effect of the transaction daemon on auto-committed transactions is also minimal; in quick-running tasks the difference stems from the time required to create and destroy the ZFS snapshot and clone. The scalability of our approach is demonstrated by the fact that the daemon's user CPU time remains a fixed percentage of the total CPU time in the two *unpack* operations, which differ by an order of magnitude (1s for ssh versus 50s for MySQL). The PostMark benchmark is the operation where the daemon spends a significant amount of CPU time in user mode. This is probably the cost of creating and searching objects in a tree data structure containing thousands of elements.

We also measured the memory performance of the transaction monitor by tracking its virtual memory size at the end of each commit operation. For most tasks there was no measurable increase in the daemon's memory size. The tasks for which the size of the daemon increased are listed in Table III. Only the MySQL-unpack task increased the daemon's memory size in the autocommit (A/C) operation; all other tasks refer to committed transactions. The largest increase we measured was 13MB: it occurred during the unpacking of the Linux kernel. From these results one can conclude that the daemon's memory requirements are very modest.

6. Related Work

Overviews of the transaction mechanism in the context of database systems can be found in references [10, 2], while an excellent description behind the motivation for supporting transactions as an operating system service and an analysis of related work can be found in reference [12].

Transactions have often been proposed as a service provided by the operating system's virtual memory storage pool [21] or a file system. The XFS file system was designed with internal support for transactions [3]. An alternative approach involves offering transaction support on top of a rudimentary

Copyright © 2009 John Wiley & Sons, Ltd.
Prepared using speauth.cls

block repository file system, which is explicitly designed to service the implementation of database management systems [22]. More recently, the Reiser4 file system offered a so-called *transcrash* facility, which is an atomic set of operations that can survive a crash [4]. However, the fine-grained granularity makes it more suitable for use within individual applications, such as a database system, rather than use in user-level operations that can span the execution of many programs. Furthermore, *transcrash* operations do not imply isolation and serializability between operations, nor the ability of a rollback.

The *QuickSilver* distributed operating system uses transactions as a way for providing applications with a consistent view of distributed processes. It differs from our approach in that it is a complete operating system, rather an addition to an existing system. This allows for better control on how transactions are implemented, but renders existing application binaries unusable. To handle this problem, the *Amino* file system [11, 12] starts from premises very similar to ours, but implements transactions by intercepting system calls using *ptrace*, and recording data in a Berkeley embedded database (BDB). *Amino*'s design offers increased flexibility in the design of transaction handling, at the expense of interposing the transaction monitor on all system processes. This has a cost of about 15% for all applications, but, interestingly, can increase performance when *Amino* is used instead of synchronous writes to provide durability.

Some systems, like Microsoft Windows[‡] and Juniper's JUNOS,[§] offer a way to rollback a failed configuration to a version committed at an earlier point. Our approach generalizes such a facility by offering it to both end users and system administrators, and formalizes its functionality by allowing the distinction between actions that are part of a transaction (those that occur in the file system clone) and actions that are not.

As a production-quality offering, the transactional NTFS technology available on Windows Vista provides atomicity, isolation, and rollback.[¶] Building on top of the transactional NTFS, the Windows Kernel Transaction Manager (KTM) provides an API for creating, committing, and rolling back a transaction.[‖] More than one process can participate in a transaction, but, in contrast to our approach, these processes must be explicitly coordinated by arranging for their code to pass and receive the transaction's handle, and have their code call appropriate functions. For instance, to create a new file an application should call the function *CreateFileTransacted*, instead of *CreateFile*. In our approach transactions are transparent to applications; the transaction's handle is only required by the stand-alone command-line program in order to commit or abort a transaction. On the other hand, KTM offers a wider range of services, such as an API, the ability to write resource managers, and support for distributed operation [23].

Finally, another related area that influenced our work concerns the merging of a cloned directory's contents with those of its parent. Our approach avoids conflicts by using the *snapshot isolation* multi-version concurrency control algorithm [9]. An alternative approach involves *optimistic replication* [24], and has been used, among others, in the Coda file system [25] and for the synchronization of mobile devices [26]. Another approach in the same design space implements *isolation-only transactions*

[‡]Windows System Restore facility: http://support.microsoft.com/kb/306084
[§]JUNOS commit command: https://www.juniper.net/techpubs/software/junos/junos54/swconfig54-getting-started/html/cli-summary-configuration-mode4.html
[¶]http://msdn.microsoft.com/en-us/library/aa365456.aspx
[‖]http://msdn.microsoft.com/en-us/library/aa366295.aspx

Copyright © 2009 John Wiley & Sons, Ltd.

providing read-write conflict detection and a variety of resolution mechanisms [27]. Our transaction monitor could be extended to inform users about a transaction's conflicts, and allow them to override them by using methods such as the preceding ones.

7. Conclusions and Further Work

Being able to undo all the effects of an operation performed on a system increases the system's usability [28, p. 75] and decreases the possibility of catastrophic errors. A file system clone can be used as a sandbox for experimenting with extensive and risky changes, but merging those changes back to the file system's parent can be an error-prone operation. The concept of a transaction offers a way to describe these changes as a set that can be atomically committed (or aborted), and as operations that can potentially conflict with others that are performing concurrently. For many real-world tasks conflicts between independent actions are rare. Consequently, detecting such conflicts at commit time through the snapshot isolation multi-version concurrency control algorithm and aborting conflicting transactions (or allowing the user to handle conflicts) is an attractive design approach: it is easy to implement, its performance impact is minor, and it avoid deadlocks.

The prototype transaction monitor we designed and implemented can be improved in a number of ways. Operating system support for locking file systems when a transaction begins or commits, and for linking a file to an inode will increase the monitor's reliability and faithfulness. Currently, when a conflict is detected at commit time, the conflicting file is reported, and the transaction is aborted. Listing these conflicts with an identifying number, and allowing users to override specific conflicts (presumably after they have established that they are immaterial, or they have merged the conflicts by hand) will help users to deal with most conflicts in a productive way. Finally, making transactions operate within an isolated environment, like a jail or zone, will increase their usefulness for performing large-scale system administration tasks.

Acknowledgements

I was extremely lucky to benefit from the brilliant advice of a number of colleagues in the course of this work. Pawel Jakub Dawidek suggested the possibility of using ZFS clones, an approach that shielded me from messing with a file system's internal structures. Georgios Gousios proposed the use of the *Fsevents* facility and the provision of a way for users to merge conflicting changes. This made me decide to implement the transaction monitor in user space, thus further simplifying the system's implementation to a degree that I could realistically handle. Damianos Chatziantoniou prompted me to search for related approaches in the database literature, where I discovered the—key to this work—snapshot isolation concurrency control mechanism. I would also like to thank Georgios Gousios, Panagiotis Louridas, Alexios Zavras, and the paper's anonymous reviewers for their helpful comments and suggestions on earlier versions of this work.

Software Availability

The source code for the software described here is available for download under a BSD license from `http://www.spinellis.gr/sw/ostran`.

Copyright © 2009 John Wiley & Sons, Ltd.

REFERENCES

1. K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633, 1976.
2. Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, 15(4):287–317, 1983.
3. Adam Sweeney. xFS transaction mechanism, 1993. Available online at http://oss.sgi.com/projects/xfs/design_docs/xfsdocs93_ps/ Accessed December 2008.
4. Joshua MacDonald and Hans Reiser. Reiser4 transaction design document, 2001. Available online at http://lwn.net/2001/1108/a/reiser4-transaction.php3. Accessed December 2008.
5. Huzefa Kagdi, Shehnaaz Yusuf, and Jonathan I. Maletic. Mining sequences of changed-files from version histories. In *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*, pages 47–53, New York, NY, USA, 2006. ACM.
6. Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA '93: Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300, New York, NY, USA, 1993. ACM.
7. Diomidis Spinellis. Version control systems. *IEEE Software*, 22(5):108–109, September/October 2005.
8. Ray Miller. Configuration management with Subversion, YAML and Perl template toolkit. In Alexios Zavras, editor, *Proceedings of the 5th International System Administration and Network Engineering Conference SANE 06*. NLUUG, Stichting SANE, May 2006.
9. Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ANSI SQL isolation levels. In *SIGMOD '95: Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, pages 1–10, New York, NY, USA, 1995. ACM.
10. Jim Gray. The transaction concept: Virtues and limitations. In *VLDB '1981: Proceedings of the seventh international conference on Very Large Data Bases*, pages 144–154. VLDB Endowment, 1981.
11. Charles P. Wright, Richard Spillane, Gopalan Sivathanu, and Erez Zadok. Amino: Extending ACID semantics to the file system. In *FAST 2005: 2nd Usenix Conference on File and Storage Technologies*. USENIX Association, April 2005. Work in progress report.
12. C. P. Wright, R. Spillane, G. Sivathanu, and E. Zadok. Extending ACID semantics to the file system. *ACM Transactions on Storage*, 3(2):1–42, June 2007.
13. Stuart I. Feldman. Make—a program for maintaining computer programs. *Software: Practice & Experience*, 9(4):255–265, 1979.
14. Amit Singh. *Mac OS X Internals: A Systems Approach*. Addison-Wesley, Boston, 2007.
15. Jeff Bonwick, Matt Ahrens, , Val Henson, Mark Maybee, and Mark Shellenbaum. The zettabyte file system. In *FAST 2003: 2nd Usenix Conference on File and Storage Technologies*. USENIX, USENIX Association, April 2003. Work in progress report.
16. Richard McDougall, Jim Mauro, and Brendan Gregg. *Solaris Performance and Tools: DTrace and MDB Techniques for Solaris 10 and OpenSolaris*. Prentice Hall PTR, Upper Saddle River, 2006.
17. Marshall Kirk McKusick and George V. Neville-Neil. *The Design and Implementation of the FreeBSD Operating System*. Addison-Wesley, Reading, MA, 2004.
18. Marshall Kirk McKusick and Gregory R. Ganger. Soft updates: A technique for eliminating most synchronous writes in the fast filesystem. In Jordan Hubbard, editor, *Proceedings of the USENIX 1999 Annual Technical Conference, Freenix Track*, Berkeley, CA, June 1999. USENIX Association.
19. Margo I. Seltzer, Gregory R. Ganger, M. Kirk McKusick, Keith A. Smith, Craig A. N. Soules, and Christopher A. Stein. Journaling versus soft updates: Asynchronous meta-data protection in file systems. In *USENIX '00: Proceedings of the Usenix Annual Technical Conference*, pages 6–21, Berkeley, CA, USA, 2000. USENIX Association.
20. Jeffrey Katcher. Postmark: A new file system benchmark. Technical Report 3022, NetApp, Sunnyvale, CA, 1997. Available online at http://communities.netapp.com/servlet/JiveServlet/download/2609-1551/Katcher97-postmark-netapp- Accessed December 2008.
21. M. Satyanarayanan, Henry H. Mashburn, Puneet Kumar, David C. Steere, and James J. Kistler. Lightweight recoverable virtual memory. *ACM Transactions on Computer Systems*, 12(1):33–57, 1994.
22. Jason Evans. Design and implementation of a transaction-based filesystem on FreeBSD. In *Proceedings of the USENIX 1999 Annual Technical Conference*, Berkeley, CA, June 1999. Usenix Association. Freenix track.
23. Pradeep Jnana Madhavarapu, Shishir Pardikar, Balan Sethu Raman, Surendra Verma, Jon Cargille, and Jacob Lacouture. Method and system for transacted file operations over a network. United States Patent 7,231,397, 2007.
24. Yasushi Saito and Marc Shapiro. Optimistic replication. *ACM Computing Surveys*, 37(1):42–81, 2005.
25. Puneet Kumar and M. Satyanarayanan. Log-based directory resolution in the Coda file system. In *PDIS '93: Proceedings of the 2nd International Conference on Parallel and Distributed Information Systems*, pages 202–213, Washington, DC,

Copyright © 2009 John Wiley & Sons, Ltd.

USA, 1993. IEEE Computer Society.

26. Tancred Lindholm, Jaakko Kangasharju, and Sasu Tarkoma. A hybrid approach to optimistic file system directory tree synchronization. In *MobiDE '05: Proceedings of the 4th ACM International Workshop on Data Engineering for Wireless and Mobile Access*, pages 49–56, New York, NY, USA, 2005. ACM.
27. Qi Lu and M. Satyanarayanan. Isolation-only transactions for mobile computing. *Operating Systems Review*, 28(2):81–87, 1994.
28. Ben Shneiderman. *Designing the User Interface: Strategies for Effective Human-Computer-Interaction*. Addison-Wesley, Boston, MA, third edition, 1998.

Copyright © 2009 John Wiley & Sons, Ltd.

Prepared using speauth.cls