# MojaveFS: A Transactional Distributed File System

Jason Frantz, Cristian Ţăpuş

Justin D. Smith, Jason Hickey

Caltech Computer Science 256-80

Pasadena, CA 91125

{buckweat,crt,justins,jyh}@cs.caltech.edu

October 23, 2002

**Abstract**

MojaveFS is a key component of an infrastructure supporting distributed transactional computing. It is closely tied to the Mojave Compiler Collection, which provides high-level programming language constructs for reliable distributed programming. Among the most important features of the system are transparency, consistency, and fault-tolerance. We use group communication systems to ensure the consistency and serializability of all operations. The file system provides a global uniform namespace which allows for mobile computing. Transactions are supported through a journalling mechanism, and replication provides fault tolerance. The design provides a UNIX-style file system.

Keywords: checkpoints, file system, group communication, migration, transactions

# 1 Introduction

The design of distributed applications is challenging because many of the usual program abstractions are not valid in a distributed framework. Network and processor failures do not allow reliable sharing of data structures without explicit replication; much of the difficulty in the development of distributed software is in the design and implementation of reliable replication. The world exists; convenient design models like functional programming simply do not work due to side-effects that are beyond program control. Yet, simple programming abstractions are the foundation of reliable software: spaghetti code is unlikely ever to be reliable.

One of the earliest and simplest abstractions for reliable concurrent programming is the transaction [4]. Transactions provide source-level fault isolation: from a process's point of view, a failure cannot occur during a transaction; if a failure occurs, it must occur before or after. While transactional models are ubiquitous in the database community, they have not been frequently applied to traditional file systems, nor have they been incorporated in traditional programming languages.

The Mojave File System (MojaveFS) is a distributed file system that uses transactions to facilitate reliable concurrent programming. MojaveFS is a central component of a distributed, fault-tolerant computing platform, and it evolves in parallel with the development of the Mojave Compiler Collection (MCC), which provides language primitives for distributed computation to user-space programs [6]. MCC is a multi-language compiler supporting both imperative languages like C and Pascal, and semi-functional languages like ML.

MCC's extensions include primitives for atomic transactions and process checkpointing. When a failure terminates some processes involved in a dis-

tributed computation, these primitives allow each surviving process to roll back its state to a previous valid state. The primitives are integrated with a journalling component in MojaveFS, which allows transactions to be applied to file I/O performed by the process; this integration is discussed in Section 3. The interaction between MojaveFS and MCC allows developers to use transactions transparently, without having to be concerned with the details of how I/O operations are performed within the transaction.

Transparency is a fundamental concern in MojaveFS. Not only should the details of the implementation be invisible to the programmer, the file system must also provide location transparency. MCC uses process checkpointing to provide a migration facility where processes can migrate to a new machine at any time. File handles are migrated along with the process state, and the new process incarnation must resume seamlessly. Similarly, a process may use checkpoints to store itself in a file for resumption later, and file references must remain valid during storage. Transparency is not just logically motivated: migration is often used to improve system-wide resource utilization.

## 1.1 Overview

When we began work on MojaveFS, our goal was to create a very reliable distributed file system, and we had several related objectives. First, we wanted to provide the full functionality of a local file system that is backward compatible with existing Unix file systems. Second, we wanted to design a system that would be scalable in the sense that the overhead of adding a new machine to a MojaveFS file system would be negligible compared to the benefits of doing so. Finally, we wanted to support distributed transactions and migration.

Figure 1 presents the MojaveFS prototype in relation to the kernel and user processes. The solid lines that connect the components represent normal I/O

operations, while the dashed lines represent special transaction system calls that MojaveFS introduces at the kernel boundary. The transaction calls and their support at the file system level are discussed in Section 3.

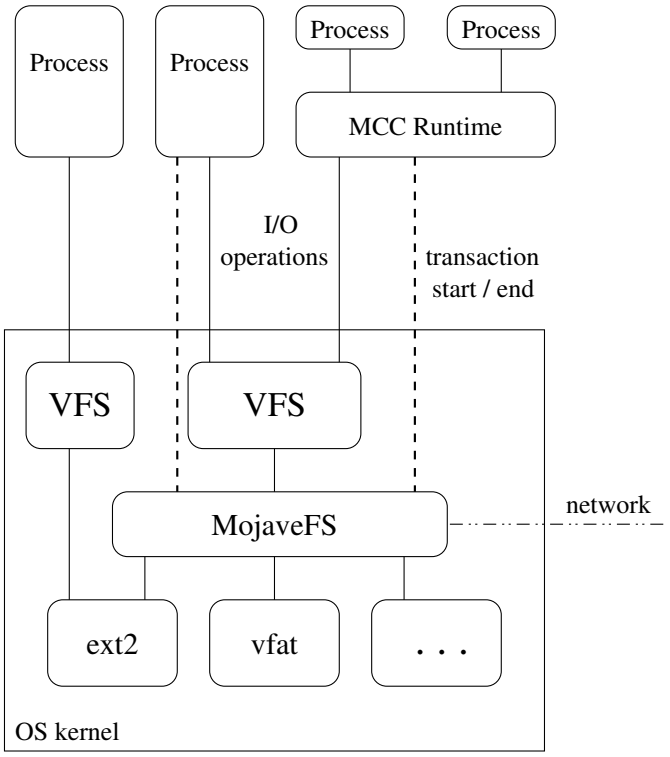


Figure 1: Overview of the Mojave File System

The leftmost two processes in Figure 1 are "legacy" processes that are not compiled with MCC. The first process uses the Linux ext2 file system for its file I/O operations, while the second process uses MojaveFS and explicitly makes use of the transactions system calls provided by MojaveFS. The rightmost two processes are compiled with the MCC compiler and, through MCC, implicitly make use of transactions by using high-level language primitives. The figure also illustrates MojaveFS's networking component, which links it with similar modules on remote machines to form the distributed file system.

Section 2 discusses the design and implementation decisions that were made for the prototype. Section 3 presents the transactional side of the system and the relation between MojaveFS and MCC. Section 4 discusses related work and shows how MojaveFS is different than, and what common characteristics it has with other distributed file systems. Section 5 concludes this paper with a summary of our work and future directions.

## 2 Design issues

The platform we are using for our prototype is the Linux operating system, kernel version 2.4.18. MojaveFS is implemented primarily as a kernel module to minimize changes to the existing kernel. However, minor kernel modifications are required for supporting network communication (for distribution of file system information) and transactions.

The native Linux file systems provide good performance for local low-level file I/O. For this reason, MojaveFS is designed as a layer between the Linux Virtual File System (VFS) and any the supported native file systems (e.g. ext2, vfat, reiserfs). MojaveFS intercepts file system calls at the VFS boundary, interprets them, and if local file I/O is necessary, one or more file operations are passed to the native file system. In order to use MojaveFS the local administrator formats a partition using a native file system, and then mounts it as file system type mojavefs. It should be noted that direct access to the native file system, while permitted for management purposes, is unlikely to be of direct use since the native files do not correspond directly to MojaveFS files. Details on this part of the implementation are discussed in Section 2.1.

MojaveFS is intended to be simple and modular. The system is composed of several small components, each with its own well-defined functionality and API.

This permits the introduction, the removal and the replacement of different components without having to rewrite other parts of the system. An overview of the system is presented in Figure 2.

The system comprises three main components. The *File I/O* component is the main engine of the file system. It interprets the I/O calls initiated by the user, and passed through the VFS. Distribution of the files, group management, and file system consistency are managed by the *Distribution* component. The *Journalling* component provides transactional functionality for the file system. Details about transactions are presented in Section 3.

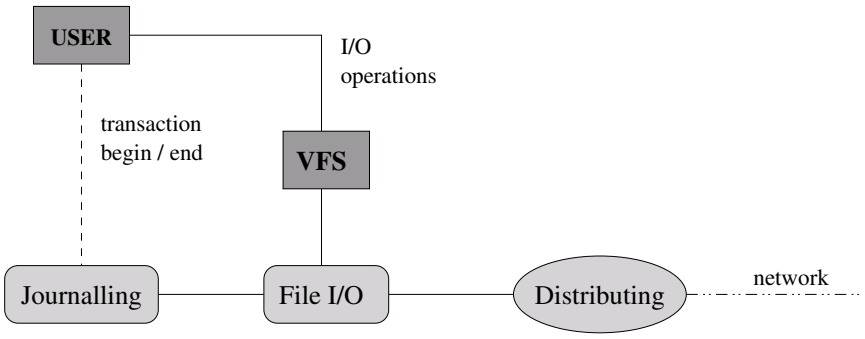


Figure 2: Architecture of the Mojave File System

## 2.1 File I/O

In a Linux system, all normal file system operations are passed through an abstract layer, called the VFS, or Virtual File System. From here, every call is routed to an individual file system. In this manner, Linux is able to provide an abstract interface that supports a large number of file systems. On a MojaveFS partition, the File I/O module acts as such a file system, handling all calls coming from the VFS.

There are many issues related to local file operations, such as block allo-

cation, that currently do not lie within the goals of our system. Instead, we pass the task of low-level block I/O to another file system. MojaveFS acts as a layer that intercepts and interprets file system calls before passing them to the underlying file system, as shown in Figure 3. The low-level file interface is abstract, and nearly any native file system can be used for low-level storage, similar to the mechanism in InterMezzo [2].

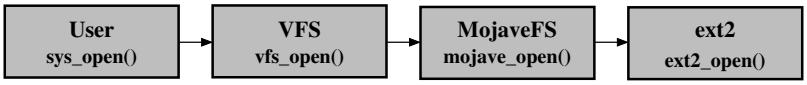


Figure 3: Example of a mkdir() call

### 2.1.1 File Chunks

MojaveFS is a distributed file system, and we need a mechanism that allows files to be replicated and distributed. Normally, only part of a file will be used by any single process at a given time. Other parts of the file may be used by other processes in other locations. In order to improve file utilization, files are split into fixed-size file fragments, called chunks. Only those chunks that are in use need be locally resident.

Each user-visible file corresponds to a directory on the underlying file system containing all of its data, called its *chunk directory*. Each of the chunks is a represented as a file on the underlying file system, but the chunk is not directly visible to the user.

Each MojaveFS file has a *chunk table*, located in its chunk directory, that holds metadata for the file. The chunk table contains a header describing properties of the entire file, and a chunk list containing information about each chunk. The latter section contains information specific to a machine, such as whether a given chunk is actually present. When a user read is intercepted for a file, the

file's chunk table is consulted to determine which chunks are locally available; a CHUNK_FAULT exception is raised for any that are not. The CHUNK_FAULT is managed by the Distribution component, described in the next section, which will retrieve the chunk data before returning control.

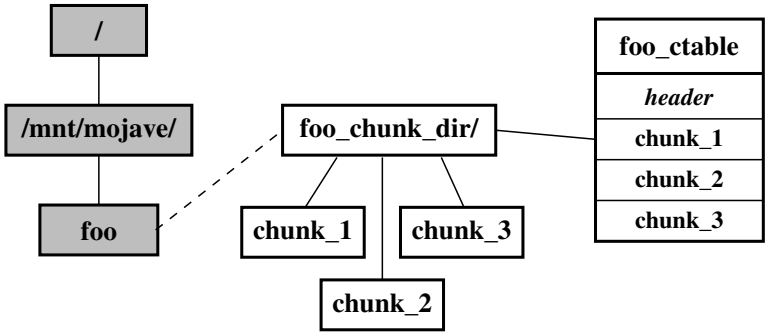


Figure 4: Overview of the chunk system. Only the shaded areas are visible to the user.

In Figure 4, we have one user file, named *foo*. All of *foo*'s chunks are located in the directory *foo_chunk_dir*. The chunk table for *foo*, named *foo_ctable*, lies inside of *foo_chunk_dir*. The figure presents the structure of the file on the underlying file system. To the user, only *foo* and its parent directories are visible.

### 2.1.2 Unique identifiers

In order to support location transparency, MojaveFS uses a global namespace for files, and each file and directory in the file system has a globally-unique identifier. This identifier has two parts: a unique identifier for the machine on which the file was created, and a unique number assigned by that machine. Note that there is no special connection between the machine that creates a file and the file itself after it has been created; the sole purpose of the machine identifier

is in helping to establish a globally unique namespace. Currently, identifiers are 64-bit, with a 32-bit machine identifier and a 32-bit local identifier.

### 2.1.3 Directories

Directories on a MojaveFS partition are represented as normal files on the underlying file system. Each directory has a *directory table* containing entries for each file inside of it. When accessing a file inside a given directory, the actual file can be resolved by looking up the filename inside of this table. This process is shown below in Figure 5. Here we have one directory, named *foo_dir*, containing the three files *file_1*, *file_2*, and *file_3*.

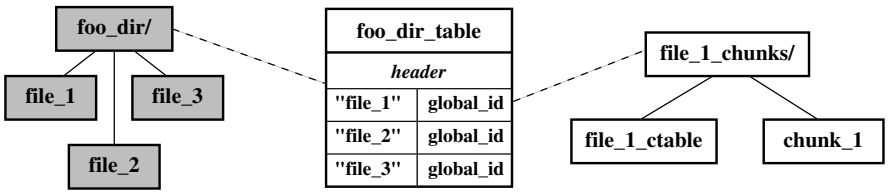


Figure 5: Overview of directory structure. The shaded areas represent the user's point of view.

## 2.2 Distribution Component

The main objectives of the Distribution component are to keep track of the actual locations of file chunks, to deliver them to specific file system instances when requested, and to maintain consistent metadata across the system. The Distribution Component has three sub-components.

The Group Manager supervises the membership of all the systems that compose the distributed system. The Data Replicator keeps track of chunk locations, and methods for delivering them efficiently. The Metadata Service determines metadata updates, broadcasts, and consistency. The three subcomponents, and

their interconnections, are shown in Figure 6.

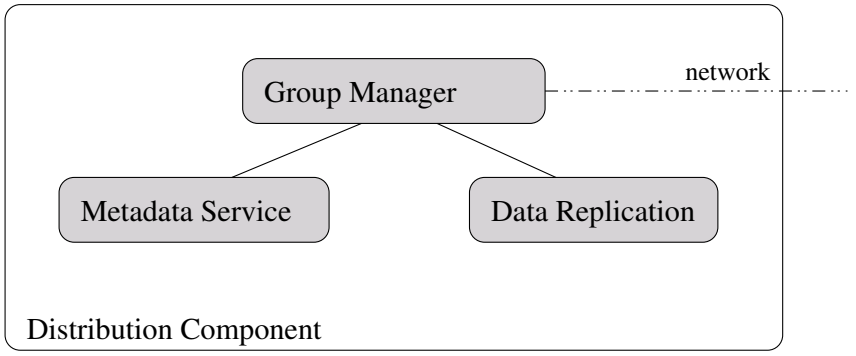


Figure 6: Architecture of the distribution component

### 2.2.1 Group Manager

The Group Manager is the key part of the distribution side of the file system. It determines what computers are part of the distributed MojaveFS system. The Group Manager also provides a form of reliable, totally-ordered broadcast among group members, similar to the Ensemble System [8, 5]. Each node in the file system keeps a list of nodes it can communicate with, called the *view*. The view is changed whenever a node joins or leaves the group; this is called *installing* the view. The properties of group communication are best described through the following invariants.

**Self** If process $p$ installs view $v$, then $p \in v$.

**ViewOrder** Views are totally ordered and they are installed in ascending order.

**NonOverlap** For any two processes $p$ and $q$ that both install view $v$, the previous views of $p$ and $q$ must either be the same, or they must be disjoint.

**MsgView** All delivered messages are delivered in the view in which they were sent.

**FIFO** Messages from one process to another in a view are delivered in FIFO order.

**Sync** Any two processes that install a view $v_2$, both with preceding view $v_1$, deliver the same messages in view $v_1$.

**Total** Any two messages $m_1$ and $m_2$ delivered to more than one process are delivered in the same order to all such processes.

**Causal** Messages are *causally* ordered: if message $m$ is delivered from process $p_1$ to process $p_2$ in view $v$, then all messages that were delivered to $p_1$ in view $v$ before sending $m$ must be delivered to process $p_2$ before delivering $m$ to $p_2$.

The total and causal ordering properties are critical; we use them to ensure consistency of replicated data, as well as provide contention resolution for resource allocation.

### 2.2.2 Metadata Service

As described in Section 2.1, file metadata is represented as a replicated chunk table. The Metadata Service (MS) ensures that all metadata in the system is up to date and that each node in the system has a consistent view of the file system. The MS is responsible for sending initial bootstrap data, such as the root directory, to new machines that join the distributed system. The MS uses the GM to serialize file operations on MojaveFS, providing UNIX semantics. Pending operations are restarted on a view change.

The creation and deletion of entries are similar. When an entry is created, the metadata for the new entry is broadcast. If another node previously created the same entry, the operation fails. Each node receiving the broadcast message

is responsible for locally updating its metadata. When an entry currently in use is deleted, the entry is marked as delete-on-close.

When the metadata attributes are modified, the message is broadcast. Attributes are modified in the order in which they are delivered.

The metadata for every file contains a subview $V_o$ of nodes where the file is open. When a file is opened, the open is broadcast and the node is added to $V_o$. When a file is closed by a node, the node is deleted from $V_o$ in a similar way.

### 2.2.3 Data Replicator

In order to provide high availability in the presence of network and processor failures, data must be replicated. Since files are partitioned into chunks, data replication can be far more efficient than replication based on entire files. The Data Replicator ranks each node on a reliability metric determined from the resources and longevity of the node.

For each file, the Data Replicator (DR) maintains a subview $V_f$ of all nodes that store chunks for that file. Chunks are replicated based on the reliability metric provided by the Data Replicator. On a view change, the DR replicates chunks hosted on nodes that left the system, and restarts any pending operations.

The *get_chunk* function is used to retrieve an existing chunk of a file $f$ from a remote location. This operation involves all members of $V_f$. It is invoked whenever there is no local copy of the chunk. The Data Replicator broadcasts the inquiry and waits for replies from members of $V_f$. If no positive replies are received, an error is raised by the DR and the operation is canceled. The DR *alloc_chunk* call is similar to get_chunk; however, it does not assume that the chunk already exists.

The *lock* operation broadcasts a lock request to $V_f$. If a lock has not already

been granted to another node, the initiator acquires the lock. Otherwise, the initiator blocks until the lock is released. The *unlock* operation broadcasts a release message to $V_f$.

When a chunk is modified by the File I/O component, the updated copy is propagated to all members of $V_f$.

Read and write operations are handled as follows. For a read operation, the chunk that is read is guaranteed to be locally resident. The read operation is processed using the local data. Write operations are blocked if the file is locked. Once the file is unlocked, the write operation is broadcast to the view $V_f$, and the call blocks until the broadcast is delivered. If another node acquires the lock before the write broadcast is delivered, the operation is restarted.

# 3 Integration with Language-level Transactions

MCC implements primitives for distributed computation as part of a generic functional intermediate language (FIR) it uses to represent all source programs. The FIR contains three primitives for atomic transactions and one primitive for capturing the process state for process checkpoints. Process checkpoints may be written to a file on disk, allowing the process to be resumed at a later time, or sent to another machine for execution, enabling process migration. A high-level syntax for accessing these primitives has been defined as an extension in C, and it is straightforward to extend other languages with support for atomic transactions and process checkpoints.

Atomic transactions and process checkpoints require the ability to record the complete state of a process at a given time, including the state of memory, registers, and all active I/O operations. The state must be completely recoverable at a later point, in the event that a transaction must be rolled back or

a process must be resurrected from a checkpoint. MCC provides the ability to rollback a process's memory and registers internally, but it requires assistance from the operating system to record and rollback the current state of any I/O operations for the process. MojaveFS provides transactional support on part of the I/O state by allowing any file operation on a MojaveFS partition to be rolled back. Certain other I/O operations, such as I/O over sockets, pipes, and sequential devices, require additional operating system support that is currently beyond the scope of MojaveFS.

When MCC compiles a program utilizing atomic transactions or checkpoints, no distinction is made between the various parts of the process state in the FIR. The distinction is applied only by the MCC back-end, which issues separate calls to capture, rollback, or commit the "memory state" and "I/O state" of a process. MCC provides a runtime library that handles rollback of the memory state; the runtime also acts as an intermediary between the FIR program and MojaveFS system calls to rollback the I/O state. This is illustrated in Figure 7, which depicts a rollback call as it is compiled by MCC.

## 3.1 MCC atomic transaction primitives

The FIR primitives support a superset of traditional nestable transactions. **atomic** $f(a_1, \ldots, a_n)$ enters a new transaction, then calls function $f$ with arguments $a_1, \ldots, a_n$. A new atomic level is created on each atomic entry, allowing for nested transactions. **rollback** $[a_{level}]$ rolls back to the beginning of the indicated $a_{level}$. If the corresponding level was entered using the call **atomic** $f(a_1, \ldots, a_n)$, then rollback will restore the program state to the state it was in *after* the entry call, and will call $f$ with arguments $a_1, \ldots, a_n$. **commit** $[a_{level}]$ $g(a_1, \ldots, a_n)$ commits the indicated level by folding changes made to $a_{level}$ into its parent level. The commit operation does not commit

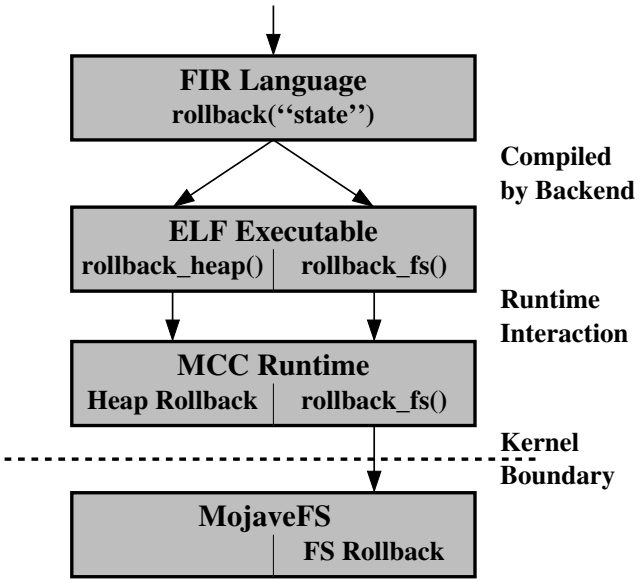


Figure 7: MCC Transactions and MojaveFS

changes made within levels more recent than $a_{level}$; changes in more recent levels remain in the journal and must be explicitly committed with another **commit** call. Once a transaction is committed, it calls function $g$ with arguments $a_1, \ldots, a_n$.

Note that **commit** allows for a superset of traditional atomic transactions. The semantics of traditional atomic transactions require that any nested transaction be committed before its parent transaction can be committed. The MCC **commit** primitive instead folds changes made at a particular level (but not changes made within any nested transactions) into its parent level; this is illustrated in Figure 8. This primitive is designed to accommodate speculative computing in a distributed, faulty environment. In such an environment, a process uses transactions to rollback a computation if another process fails at an earlier part of the computation; the process will commit older transactions once it discovers that all other processes have reached the corresponding step of

the computation. In this model, older transactions are always committed before newer transactions, which is the opposite behavior from traditional transactions.

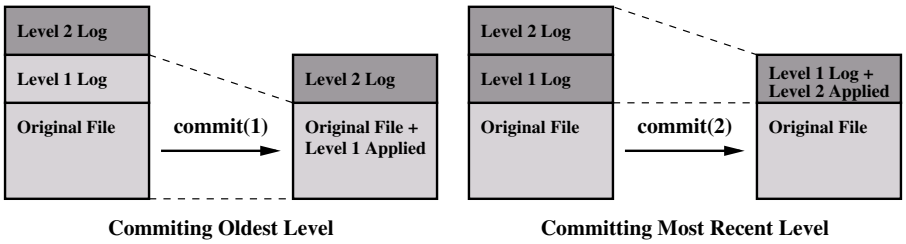


Figure 8: MCC Commit Operations

Note that while a process is involved in at least one transaction, changes made to files by that process should remain invisible to all processes not participating in the transaction. As a result, MojaveFS maintains a log of all I/O operations that mutate the file system state; the changes are not written to the original files on disk until the transaction is committed.

Implementation of these primitives to capture and rollback the memory state is discussed elsewhere [6]; the remainder of this section discusses how the design of MojaveFS accommodates these primitives to capture and rollback the file state. MojaveFS provides three interface functions that correspond to the three FIR primitives. The interface to these functions requires new system calls on the kernel boundary which are described below. The design of the interface is such that MCC does not need to know the current state of any particular file descriptor; MojaveFS will be responsible for identifying the open descriptors for a process, and determining what state needs to be captured and rolled back.

## 3.2 System call interface for transactions

The **atomic** system call signals to MojaveFS that the process is entering a new transaction. On this call, MojaveFS iterates over all file descriptors owned

by the process, and enters an atomic transaction on each file which is on a MojaveFS partition. MojaveFS returns a list of descriptors on which it *cannot* enter a transaction, including all descriptors corresponding to pipes, sockets, and files on non-MojaveFS partitions. It is up to the MCC runtime library to filter I/O on these "uncontrolled" descriptors. MojaveFS will log all actions on the remaining descriptors until the transaction level is committed or rolled back.

Note that until the transaction level created by the **atomic** call is committed or rolled back, MojaveFS must intercept any call to open a new file on MojaveFS partitions and make sure that I/O operations on the file are logged until the transaction is completed. Also, MojaveFS defers any call to close a file descriptor until the transaction is completed. This requires MojaveFS to keep track of some state information specific to the process in addition to the logs maintained for each descriptor.

The **commit** system call signals to MojaveFS that all logs associated with a particular transaction level can be folded into the parent level. The MCC runtime indicates which level should be committed. If the transaction is the outermost transaction, then all logs for that level can be committed to the file system. Otherwise, the changes are applied to the parent level's logs and remain invisible to any process not involved in the transaction. This call should close any descriptors that were closed within the transaction but deferred by MojaveFS. This call does not need to return any information to the MCC runtime, unless an error occurs.

The **rollback** system call signals to MojaveFS that all logs associated with a particular transaction level, and all later levels, should be discarded. The MCC runtime indicates which level should be rolled back. Since MojaveFS does not modify the original files in place, it can simply delete all logs associated with

the indicated level and all later levels. This call does not need to return any information to the MCC runtime, unless an error occurs.

## 3.3 Support for process migration and suspension

MojaveFS supports processes which use the checkpointing mechanism in MCC to migrate to another location or suspend to a file. Before a process migrates, MojaveFS provides the process with a list of all descriptors referring to files on a MojaveFS partition. After the process migrates to a new location, the MCC runtime can use this information to reopen the file descriptors. As with transactions, MojaveFS is unable to migrate file descriptors corresponding to sockets, pipes, and files which do not reside on a MojaveFS partition[1].

When a process suspends itself to a checkpoint on disk, MojaveFS must preserve a reference to every file which is currently open by the process. The checkpoint is written to a MojaveFS partition as a hidden directory. A hard link is created in this directory for each file that is currently open by the process; this prevents MojaveFS from losing the file reference even if the user deletes the original file. The process state is written to a special file within this directory. When the process is resurrected, MojaveFS traverses the hard links and reopens each of the descriptors, restoring the I/O state of the process.

## 3.4 Journalling Component

This section describes the MojaveFS implementation of transactions. MojaveFS maintains one journal per transaction level per process. Once a transaction level is entered, a new journal is created and locks are acquired for all open files. When a nested level is committed, its journal is merged with that of

[1]Migration of these descriptors will require additional operating system support which is beyond the scope of MojaveFS.

the parent transaction; for the outermost level, the changes in the journal are written directly to disk. The locks acquired for a transaction level are released when the level is committed or rolled back.

In order to initiate the locking mechanism, MojaveFS uses features provided by the kernel to obtain the set of open file descriptors associated with the process, and identifies the safe file descriptors[2]. The Group Manager has the authority to revoke locks from off-line computers and is responsible for propagating lock information to new members of the system. Locks are implemented as leases to prevent starvation of concurrent accesses to files involved in a transaction.

The journal contains a list of operations that have been performed on each file since the beginning of the transaction. For efficiency, chunks are treated as copy on write blocks, similar to pages in virtual memory systems. There is an extra level of indirection for chunk operations, as for each access MojaveFS must check the journal for the current location of the chunk. The journal is indexed by file descriptor; for each descriptor we maintain different lists to keep track of addition, modification, and deletion of chunks.

MCC allows a process to open and create files within a transaction. To prevent a newly created file from being visible outside the transaction, we do not propagate the new directory entry to the disk until the transaction is committed. In order to successfully open an existing file, a lock on the file must be acquired. While we can guarantee deadlock free acquisition of locks for files that are open on entry, there is no such guarantee for files opened within a transaction.

When transactions are committed, MojaveFS must ensure changes are propagated from the journal to the physical disk before the locks are released, and that replicated copies of modified chunks are either invalidated or replaced.

[2]Descriptors corresponding to files on a MojaveFS partition are considered safe.

# 4 Related work

The area of distributed system and distributed file systems has been an active research area for about two decades now. We discuss a few of the systems that shaped research in this area. Our work attempts to incorporate many of the ideas of these systems.

One of the first distributed file systems used on a large scale was the Andrew File System (AFS) [7]. Developed at Carnegie Mellon University in the early eighties, AFS was very popular due to its scalability, and its security which was provided by using Kerberos. We are addressing some of the problems that researchers and users alike put forward with respect to AFS. AFS offers session semantics rather than UNIX semantics, which makes real-time concurrent file sharing almost impossible. In our system, we provide the user with UNIX semantics. Another drawback of AFS is availability. In the initial version, entire files were transferred to destinations on the open call. Our system is more selective in sending data. This gives our system higher availability and lower overhead on open calls.

CODA [1] is one AFS's descendants. Developed at Carnegie Mellon as well, CODA was designed as a distributed file system that would provide access to files even while machines were disconnected. The file system was designed for portable machines and did not enforce strict consistency of data, requiring user input to solve conflicts. Another issue is that CODA "hoards" files on the local machine to make them available while off-line, which makes CODA impractical to use on low-end machines with scarce resources. Our system currently does not address the availability of data while a mobile computer is off-line.

The latest file system in the series initiated by AFS is InterMezzo [2]. Although it is similar to MojaveFS in the sense that it was built as a filter between

the VFS and the local file systems, InterMezzo has a client-server architecture and it follows the direction taken by CODA in providing access to data while the computer is off-line.

SPRITE [9] is similar in many ways with MojaveFS. It is part of a distributed operating system that was designed to transparently provide a distributed file system and process migration [3]. SPRITE also provides UNIX semantics. One major difference between the two systems is the fact that SPRITE was designed using a client-server paradigm.

# 5 Conclusions

MojaveFS provides a key part of a reliable distributed operating system. Seamless integration with programming languages provides a new programming model for designing verifiable highly-available distributed systems. Transactions are a central tool for failure isolation that is widely used in the database community but has not yet had a major effect on traditional file systems. MojaveFS provides location transparency and transparent support for transactions on file I/O, enabling effective process mobility. MojaveFS supports these features while retaining backward compatibility with existing file systems.

The implementation of MojaveFS is currently in progress. We are currently investigating support for non-file I/O, including sockets, FIFOs, and devices.

# References

[1] Peter J. Braam. The coda distributed file system. *Linux Journal*, June 1998.

[2] Peter J. Braam, Michael Callahan, and Phil Schwan. The intermezzo filesystem. 1999.

[3] F. Douglis and J. Ousterhout. Transparent process migration: Design alternatives and the sprite implementation. *Software-Practice and Experience*, 21(8), August 1991.

[4] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1994.

[5] Jason Hickey, Nancy Lynch, and Robbert van Renesse. Specifications and proofs for Ensemble layers. In *TACAS '99*, March 1999.

[6] Jason Hickey, Justin D. Smith, Brian Aydemir, Nathaniel Gray, Adam Granicz, and Cristian Tapus. Process migration and transactions using a novel intermediate language. Submitted to *$30^{th}$ ACM Symposium on Principles of Programming Languages (POPL'03)*; see also Technical Report caltechCSTR 2002.007, 2003.

[7] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[8] Xiaoming Liu, Christoph Kreitz, Robbert van Renesse, Jason Hickey, Mark Hayden, Kenneth Birman, and Robert Constable. Building reliable, high-performance communication systems from components. In *$17^{th}$ ACM Symposium on Operating Systems Principles (SOSP'99)*, volume 34 of *Operating Systems Review*, pages 80–92, 1999.

[9] J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson, and B. Welch. The sprite network operating system. *IEEE Computer*, 21(2):23–26, February 1988.