Transaction-based Process Crash Recovery of File System Namespace Modules

David C. van Moolenbroek, Raja Appuswamy, Andrew S. Tanenbaum
Dept. of Computer Science, Vrije Universiteit Amsterdam, The Netherlands
{dcvmoole, raja, ast}@cs.vu.nl

***Abstract*—In this paper, we describe the emerging concept of namespace modules: operating system components that are responsible for constructing a hierarchical file system namespace based on one or more individual underlying file objects. We show that the likely presence of software bugs in such modules calls for the ability to recover from crashes, but that the current state of the art falls short of the desired behavior. We then introduce a crash recovery solution that is based on transactions, and detail the requirements for a system to implement this solution. We apply our solution to two different use cases: the primary namespace module for a storage stack, and an extension module that exposes the contents of scientific data files. Our evaluation shows that the transaction system has low overhead and significantly adds to the robustness of the namespace modules.**

***Index Terms*—Operating systems, File systems, Data storage systems, Software reliability, Fault tolerance**

I. Introduction

Relatively recent developments have brought about a new concept in operating system storage stacks, namely *namespace modules*: software components that construct and manage a hierarchical file system namespace, using one or more file objects managed by an *object storage layer* below. Traditionally, namespace management has been an integral part of file systems, but there are two developments that have resulted in the isolation of such functionality into a separate module.

First, after the success of distributed storage stacks that separate metadata (and thus namespace) management from object storage for scalability [1], such architectures are now being introduced on single-system storage stacks as well, mainly to make up for inherent reliability problems with block-level RAID in traditional storage stacks [2].

Second, with user space file system frameworks such as FUSE [3], it has become relatively easy to write namespace modules that expose the inner structure of individual files using a hierarchical model. While such modules can be used for internal maintenance of such files, a recent case study [4] has shown by that current-day file formats can be highly complex, and suggests that important information for the storage stack is lost by storing these files as a single blob. Thus, there is a case to be made for use these namespace modules as the primary means of access to such complex individual files.

Due to the complexity resulting from (in particular) multithreading for high performance, and the large amount of code typically required for proper parsing and manipulation of complex file structures, both these types of namespace modules are likely to contain software bugs. If triggered, these bugs cause a runtime failure within the namespace module, typically with severely damaging consequences for both the running applications and the underlying storage. While namespace modules are often already isolated in user space processes (e.g., [5], [3]), this is only a first step toward dealing with failures, and we claim that current situation leaves several properties to be desired: 1) application-transparent recovery from transient crashes, 2) fine-grained request failure in the case of repeated failures, and 3) preservation of integrity of the underlying file objects.

In this paper, we present a crash recovery solution for namespace modules, based primarily on atomic transactions. In particular, we show that by using transactions between the namespace module and the object storage layer, we can not only provide recovery from fail-stop crashes with the three aforementioned properties, but also use semantic information in the transactions to minimize the runtime overhead.

We implement the transaction framework in our own storage stack, and apply it to two example namespace modules: the stack's primary POSIX namespace module, and a new extension module that allows the internal namespace of HDF5 scientific data files [6] to be mounted into the standard file system hierarchy. Our evaluation confirms that our prototype implementation has relatively low runtime overhead and allows for full recovery from large numbers of injected faults.

The rest of the paper is structured as follows. In Sec. II, we describe the motivation for this work, elaborating on both the emergence of namespace modules as a concept, and the need to improve on the current situation with respect to crash recovery of such modules. In Sec. III, we present the design of our transaction-based solution. In Sec. IV, describe the storage stack we use as test platform, the implementation of the transaction framework on this platform, the changes made to the original POSIX namespace module to make it recoverable, and the new HDF5 namespace module with recovery support. Sec. V presents the evaluation of our implementation, assessing both the resulting reliability improvement and the performance impact. In Sec. VI, we list related work. Finally, we conclude with future work in Sec. VII.

II. Motivation

In this section, we describe the motivation of our work: the emerging concept of namespace modules (Sec. II-A) and

the need for a better solution for recovery from bug-induced crashes of such namespace modules (Sec. II-B).

A. Namespace modules as an emerging concept

Within the storage stack part of the operating system, one layer of functionality that appears to be increasingly common is what we call the *namespace layer*: a layer in the storage stack that, based on one or more underlying individual file objects and their contents, constructs a namespace view for use by applications. This namespace layer can typically be found directly under the operating system's Virtual File System (VFS) layer, where it exposes a hierarchical file system namespace that is accessible through a standardized interface. This interface is most commonly the POSIX application programming interface (API) [7], and thus, the hierarchy is made up of files, directories, links, and so on. Instances of the namespace layer, which we call **namespace modules**, thus translate file system requests coming from VFS, into individual file object operations on the lower layers of the storage stack. We identify two main types of namespace modules.

1) Primary namespace modules: The first type of namespace modules is emerging as a direct result of a shift in storage stack architectures. The traditional storage stack has a single file system layer which converts file system operations to block operations. The lower layers thus operate on a block basis, and this block interface mixes both file data and namespace metadata. Redundant storage across devices (RAID) is performed at the block level. This model is depicted in Fig. 1a. More recently, storage architectures have started to separate the management of the namespace and related file metadata, from the storage of actual file data. In this model, depicted in Fig. 1b, the lower layer exposes an abstraction of individual *file objects*. We refer to this lower layer as the *object storage layer*. It typically implements redundant storage at the object level, making the translation to block operations only at the lowest (device) level. On top of the object storage layer, the namespace layer is responsible for constructing a file system namespace out of the collection of individual objects. We call instances of this layer **primary** namespace modules, as they are the primary managers of the underlying storage.

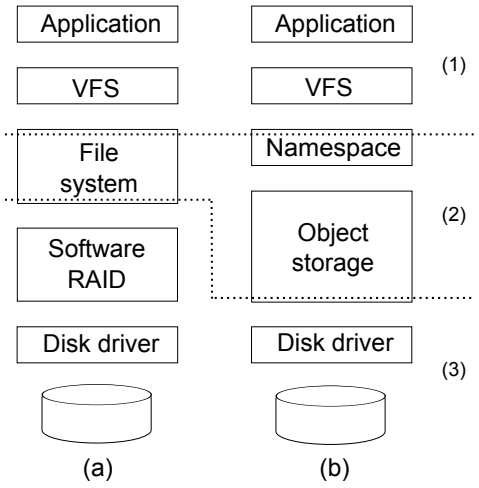

Fig. 1. The figure shows the layers of (a) the traditional storage stack and (b) the object-based storage arrangement. The dotted lines delineate interface abstractions between layers: (1) the file system interface, (2) the object interface, not found in the traditional stack, and (3) the block interface.

This general concept was introduced in the distributed storage world [1]. It has since been widely adopted in distributed storage systems (e.g., [8], [9], [10]), and has sparked the development and standardization of object-based storage devices [11]. The main reason for this architecture is scalability, as the decoupling management of metadata from the storage of data allows the object storage layer to be distributed across a large number of fully independent data storage nodes.

However, the object-based architecture is now also being introduced in storage stack designs for single systems, both in research (e.g., hFAD [12] and our own Loris storage stack [5]) and in the real world (e.g., ZFS [13]). Even though scalability is not a major concern on a single system, there are several other advantages of this new architecture. Most importantly, since cross-device redundancy is now implemented at the object level rather than the block level, this redundancy functionality can now make use of object information. This allows the storage stack to not only solve fundamental reliability problems present in block-level RAID, but also recover more files in case of a large number of concurrent device failures, and even store individual files with user-specified levels of redundancy [2]. In addition, the separation of the namespace layer allows it to be replaced easily and without affecting the underlying storage [14]. Furthermore, it allows for the introduction of several additional namespace modules that together manage the primary object storage space, thereby offering a variety of rich interfaces to the application in addition to the standard POSIX API [12].

Because of these advantages, we expect that the object-based storage architecture will see even more widespread adoption on single systems, and thus, that primary namespace modules will become more prevalent as well.

2) Extension namespace modules: At the same time, new user space file system frameworks such as FUSE [3] and PUFFS [15] have resulted in the emergence of a second type: **extension** namespace modules that expose the internal structure of a single file, by mapping its contents to the standard hierarchical file system model, allowing the file to be mounted into the system's file hierarchy. These modules thus "break open" the underlying file in a way that makes the file contents available to the end user and moreover, to any tool that uses the standard POSIX API.

Such modules are additions to the normal file system hierarchy, and typically loaded on demand. They can be written for any files of which the contents can be mapped to the standard namespace interface; for example, archive files, document files, and scientific data files. They can be used for inspection and maintenance of files generated by other applications, but also as the primary means of creation, manipulation, and usage of those files. As such, these namespace modules can serve to standardize access to these files between applications. In

addition, a recent case study of complex file formats [4] suggests that storing complex files as binary blobs prevents the application from expressing its desires to the storage stack, for example to guarantee atomicity of operations on the file. This then forces the application to perform relatively expensive file system operations (such as full file copies and frequent `fsync` calls) to maintain such guarantees. Exposing the internal hierarchy of complex files through a namespace module would allow applications to better express their intentions to the operating system.

Even though stackable file systems [16] and Hurd translators [17] have been around for a long time, the new frameworks make development of namespace modules of this type accessible to a wide range of developers (and not just kernel developers), also on commonly used operating systems. In systems like FUSE however, the provider of the underlying file object is the entire storage stack, and the POSIX API is used for access to the file. We will show in Sec. IV that differently structured storage stacks can allow extensions to talk directly to the object storage layer.

3) Hybrid cases: However, we believe that the two aforementioned types of namespace modules are in fact not so different, especially when considering virtualization. We are working toward a lightweight virtualization system that allows multiple namespace modules–one per virtual environment– to share a single object storage layer, thereby eliminating much of the redundancy found in contemporary virtualization systems [18]. These virtual environments contain individual applications, and thus have private sets of files and are created and destroyed on demand. The namespace modules in such virtual environments can thus be classified as either type.

B. The reliability problem

All these namespace modules can represent a substantial amount of error-prone code in the operating system layer. The primary namespace modules make up a crucial part of the storage stack. Unlike most parts of the object storage layer, the namespace layer has to process many application requests immediately, before any operation caching comes into play. The desire for high performance typically translates into extensive use of multithreading, which is a well-known source of reliability problems, especially so in file system code [19].

In contrast, extension namespace modules are not crucial to the storage stack itself, nor are they performance critical; however, they are expected to interpret complex file formats. Worse yet, since these namespace modules must be able to handles files they have not created themselves, they have to deal correctly with arbitrary contents in the underlying file. This requires either substantial amounts of newly written code, or the inclusion of an external parser library of which the quality is not always known. Previous studies have suggested that the number of bugs in a piece of software is largely a function of its source code size, reporting numbers in the range of 0.5–6 bugs per 1,000 lines of code [20], [21]. Thus, neither type of namespace module can be expected to be bug free.

With the possibility that software bugs can cause namespace modules to crash as a given, the first concern is then the stability of the operating system in general. This issue can be addressed by placing the namespace module in an isolated user-space process, as is done in microkernel operating systems as well as by user space frameworks such as FUSE. However, even in such systems, no attempts are made to recover the namespace module after it has crashed. At best, applications receive I/O errors for all requests that involve the crashed module. While this protects the operating system from damage extending beyond the boundaries of the module, we believe that this is insufficient, for three reasons.

First, the application is always exposed to the crash, even when the cause of the crash was transient. As indicated, such crashes could be the result of race conditions in a multithreaded environment. If the namespace module could be restarted after a crash, and the calls could be reissued, the transient crash would not occur again. This would allow for crash recovery that is fully transparent to the applications.

Second, the entire namespace module is shut down after the crash, even if the crash was the result of a software bug in a specific code path of a request handler. For example, many file system bugs are found in error handling code [19], which triggers only exceptional conditions. In that case, repeating the call after recovery would thus crash the namespace module in the same way. However, if the system could recognize this case and fail only the corresponding application call for that particular request, all other request (and thus, applications) can continue to make use of the namespace module.

Third, namespace modules typically translate single incoming application requests into multiple related object operations sent down to the object storage layer. For example, creation of a new file typically involves two steps: the creation of the new file object, and addition of the name record to its containing directory object. To maintain consistency of the underlying storage, such a set of operations is supposed to be atomic. However, it is possible that a namespace module sends down an incomplete subset of the operations, and then crashes. It will then leave the underlying storage in an inconsistent state. This also prevents subsequent recovery.

Thus, we argue that a proper crash recovery system for namespace modules can and should 1) recover the modules in an application-transparent way from transient failures, 2) fail only specific system calls in the case of repeated failures, and 3) never let fail-stop [22] crashes introduce corruption in the underlying storage. At the same time, namespace modules can be assumed to crash only in exceptional situations, and thus, such a system should have low overhead during normal runtime. In the rest of the paper, we will describe our approach to meeting these requirements.

III. Design

We will now present the design of a solution that is based on **transactions**. We start by stating our assumptions about the operating environment and failures (Sec. III-A). We then show how transactions form the basis of our recovery system

(Sec. III-B), and the changes needed to the object storage layer (Sec. III-C) and the VFS layer (Sec. III-D). Finally, we list the requirements for namespace modules themselves (Sec. III-E).

A. Assumptions

We assume that the namespace layer is positioned below the VFS layer and on top of an object storage layer, indeed as depicted in Fig. 1b. As such, the main purpose of a namespace module is to process typical requests coming in from the VFS layer [23]. The module handles each request by performing a number of operations on individual objects the storage layer below. Typical object operations are creation and deletion of an object; read, write, and truncate operations on the object; and, retrieval and modification of attributes associated with the object. While the namespace module may use input from outside the storage stack (e.g., the current time of day), any modifications it makes as a result of a VFS request must involve stored objects only. In the case of extension namespace modules, only one underlying object is involved, and thus, no object creation or deletion will take place. We believe that these assumptions are sufficiently generic that they cover any single-system object-based storage architecture, as well as namespace modules in frameworks such as FUSE.

Software bugs may cause arbitrary behavior, including undetected propagation of corrupted results to the application or storage. Thus, we have to limit our solution to a subset of all possible failures. For failure isolation, we assume that the namespace module runs in a separate user space process, and that a basic infrastructure is in place to restart this process cleanly if it has crashed. Furthermore, we assume the following failure model. First, all failures are detectable, either by the module itself or the outside system. Previous research has suggested that silent failures are rare [24]. Second, failures are detected before corrupted results are propagated outside the process. Similarly, previous studies suggest that fault propagation is rare [24], [25]. This failure model covers more cases than fail-stop [22], because it also allows for wild memory overwrites as long as the overwritten memory is either not accessed by another process or (for example) protected by a checksum. In addition, the process isolation limits the impact of resource leaks. The failure model matches that of other contemporary work in this area [26], [27], [28], [29].

B. Transactions and recovery

Given these assumptions, it is easy to see how one can introduce a system of atomic transactions. In the course of processing a VFS request, all operations that modify underlying objects can be bundled into a single transaction, which is sent down to the object storage layer at the end of that request. Thus, each VFS request that spawns any object modifications at all, results in one such transaction. The request will only finish once either the transaction has been fully committed in the object storage layer, or it has been aborted due to an error, with no changes made to the underlying storage.

Such a transaction system has two effects for crash recovery. First, each transaction is atomic, and thus the object storage layer commits either the entire transaction or no part of it. In addition, each transaction makes a transition from one consistent state to another. Thus, there is no possibility that a crash of the namespace module causes inconsistency in the underlying storage. This covers the third point from Sec. II-B, and paves the way for covering the first two points. Note that we do not discuss isolation between transactions in this work: while it is crucial that proper isolation is provided if multiple concurrent transactions can exist, a similar requirement exists in the original situation, and we believe that solutions for this are too implementation specific to be generalized.

Second, since the namespace layer must commit the transaction before the end of processing the VFS request, it can not have any pending changes in its memory that do not pertain to VFS requests that are currently ongoing. Thus, a crash of the namespace module will at most result in loss of pending state for the ongoing requests only. As a result, if the namespace module crashes and is restarted, the underlying storage layer is guaranteed to reflect all the changes made by previously completed VFS requests. For each currently ongoing VFS request, either none or all of its changes have been committed already–after all, it is possible that the namespace module crashes either before or after each corresponding transaction has been committed. As we will show in the next subsections, these conditions are sufficient to allow the namespace module to guarantee correct recovery from a crash, as long as some additional requirements are met.

C. Support in the object storage layer

The transactions are sent as atomic units to the layer below the namespace module: the object storage layer. In general, this layer is expected to provide a cache for object data and operations. Thus, while it must process each transaction right away, the transactions need not be persisted on a device immediately–a successful transaction requires only that the modifications have been taken in by the cache. In rare cases, the transaction may result in a failure at the object storage layer. For example, this layer may encounter a general out-of-space condition, or an integrity problem with data that needs to be read in to perform the transaction (e.g., for a partial write to a block). In that case, the entire transaction must be aborted, and an error must be returned to the namespace module.

We argue that due to the semantic information available in the transactions, the object storage layer need not implement full support for rollback of the operations in the transaction. All actions required to guarantee the successful execution of the transaction, such as checking available resources and reading in data blocks for partial writes, can be done before the actual transaction is executed. For example, write operations need not make expensive memory copies for rollback after a later failure. In multithreaded environments, some minimal rollback support may be needed, such as reserving space beforehand and canceling the reservation on failure. However, the actual operations will be deferred for performance reasons (e.g., delayed writes), and hence any device failure will be detected long after the corresponding operation has been

acknowledged–this is the same in the traditional situation. Thus, by avoiding the need for rollback, the transaction support in the object storage layer has minimal overhead.

There is one exception to the above: the application may request *direct I/O*, in which case the object storage layer must complete the transaction only once the changes have been persisted on a device. The transaction must then fail atomically if any of its operations cannot be committed to the device immediately. In that case, full rollback support is required. However, the rollback overhead is likely to be masked by the cost of the immediate device I/O. It should be noted that abstractions within the object storage layer may prevent rollback of some combinations of operations: while object create, write, and set-attribute operations can be undone, truncate and delete operations can not. For transactions that contain only one operation of the latter category, this operation can be performed last. We have not found any scenario where multiple such operations would be sent down in a single transaction by a namespace module.

D. Support in the VFS layer

The VFS layer must implement the necessary support for recovery after a namespace module has crashed and restarted. The first step is resynchronization of the namespace module to the current state of VFS, which reflects the result of all completed requests, and of none of the requests that were still ongoing to that module at the time of the crash. As part of this step, the recovery procedure in VFS must issue requests to reopen any other previously opened files and restore any mount points, for example.

As the second step, VFS must reissue all requests that were ongoing at the time of the crash. In particular, VFS should reissue them one by one, for two reasons: 1) for transient failures caused by race conditions, serial execution prevents such race conditions from happening again, and 2) for repeated failures caused by a bad implementation of a request handler, serial reexecution allows VFS to pinpoint the request that causes another crash, and abort just that request. This cleanly covers the first and second points from Sec. II-B.

Especially in multithreaded environments, for some requests, the corresponding transaction may have already have been committed in the object storage layer before the crash. Thus, the namespace module needs a way to recognize whether a request has already been completed or not. To this end, at the very minimum, each VFS request must have a unique identifier that stays the same upon repetition of the request.

E. Requirements for namespace modules

With this infrastructure in place, a namespace module can recover from all failures in our failure model, as long as it adheres to the following four requirements.

First, the namespace module may not defer any modifications to objects until after finishing the corresponding VFS request. All operations must be part of the request's transaction. We believe that for namespace modules, this is not a prohibitive requirement. Unlike file systems, namespace modules generally sit on top of the storage stack's main cache, and thus, this will incur little to no extra device I/O.

Second, the namespace module must deal correctly with transaction failures reported by the object storage layer. In particular, if the module maintains its own read caches (e.g., for file attributes or directory data), then either these caches must be updated (pessimistically) only after the transaction succeeds, or any (optimistic) modification before transaction commit must be rolled back. Since transaction failures in the object storage layer are expected to be highly exceptional (see Sec. III-C), even more drastic approaches can be taken in this case, as we will show in Sec. IV-D.

Third, the namespace module must not expect that its modifications are visible to the lower layers before the transaction is committed. For example, it must not expect that an object read operation will reflect changes made by an earlier object write operation within the same request, as the write operation will be deferred as part of the transaction. This requirement can be fulfilled by the transaction framework, or by optimistic modification of local caches.

Fourth, the namespace module must handle repeated requests correctly. If a request's transaction was committed before a crash, but VFS never received the request's reply, then VFS will reissue the request after the restart. Idempotent requests can simply be performed again, but the same does not apply to nonidempotent requests. For example, a "create file" request from VFS that succeeded the first time, would fail with a file-exists error the second time. Thus, after a restart, the namespace module must recognize any such requests for which the transactions have already been committed. For atomicity reasons, this information must be saved as part of the transaction, and thus be maintained by the object storage layer. In our case studies (Sec. IV) we present an approach that requires no further extensions to the object storage layer.

In some cases, more than just the request identifier must be saved. For example, any results that had not yet been copied to VFS or the application, must be copied upon repeat. In addition, recall that VFS will restore a restarted namespace module to the state *before* the ongoing requests. Thus, any internal state changes resulting from repeated requests (e.g., adjusting open counts of files) must be replayed as well.

IV. Implementation

We now describe the implementation of our design. We introduce our previously developed Loris storage stack (Sec. IV-A), the infrastructure changes we made to it to support transactions (Sec. IV-B), the modifications we made to make its primary namespace module restartable (Sec. IV-C), and a new restartable extension namespace module developed for this work (Sec. IV-D).

A. Background: the Loris storage stack

In previous work, we have come up with a new object-based operating system storage stack [5], by applying two modifications to the traditional storage stack. First, we split up the traditional file system into three separate layers: a

namespace layer, a cache layer, and a device layout (*physical*) layer. Second, we swapped the physical layer with the traditional RAID layer (now the *logical* layer). We call this new arrangement the Loris storage stack; it is shown in Fig. 2b. The VFS and device driver layers are unchanged. The four new layers communicate in terms of *objects*, which are file-like storage containers that consist of a unique identifier, a variable amount of byte data, and a set of associated attributes. The lower three layers can together be viewed as an object store. We will now briefly describe the four layers.

Analogous to what we described in Sec. II-A, the **namespace layer** is responsible for the translation of VFS file system requests to file object operations. We currently have a single namespace module: the primary implementation of the POSIX namespace for our storage stack. This module stores both files and directories as objects; lower layers are not aware of the concept of directories. It uses object attributes to store POSIX attributes of files. It maintains two extra objects for its metadata: a bitmap to track in-use object identifiers, and a list of open deleted files needed for system crash recovery.

Below the namespace layer is the **cache layer**, where object data and attributes are cached, and various operations are deferred, for performance. This layer uses a large amount of system memory for this purpose, staging and evicting data as necessary. Below it, the **logical layer** implements the equivalent of the traditional RAID layer, but on a per-object basis. Each object has an associated policy that determines its RAID-like policy. For example, an object can be mirrored or striped across a number of devices. The logical layer constructs single "logical" objects out of one or more "physical" objects stored in the **physical layer**. This layer consists of physical modules that each manage the layout of one underlying device, thus translating object operations to block operations.

As we already sketched in Sec. II-A, the new arrangement has several structural advantages, mainly for reliability and flexibility, as a result of the logical layer now operating at the object level. We have implemented our prototype of the Loris

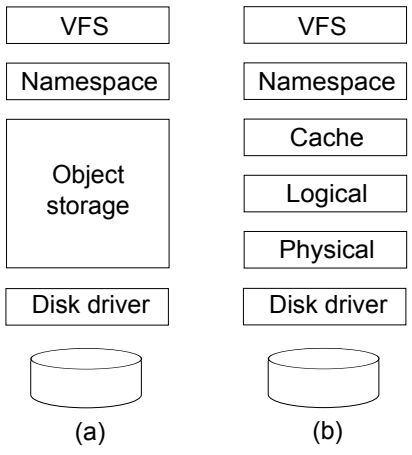

Fig. 2. The figure shows (a) the generic object-based storage architecture from Fig. 1b, and (b) the layers of the Loris storage stack.

stack on the MINIX 3 microkernel operating system. In this environment, each module of each layer is a separate process running in user space.

B. Infrastructure changes

We implemented the transaction infrastructure described in Sec. III-C in the cache layer. Layers below the cache layer need not be aware of transactions. The transaction is fully maintained within the namespace module until it is committed, at which point it is sent down to the cache layer in its entirety. This saves on interprocess communication and hides local transaction aborts from the lower layer. We added a new "commit transaction" operation type to the communication protocol between the namespace layer and the cache layer, consisting of a set of one or more operations on objects. The cache ensures atomic execution of the transaction. Since our storage stack prototype does not yet support direct I/O, we did not add rollback support.

We also implemented the necessary changes in the VFS component, as per III-D. We added identifiers to all VFS requests. Since VFS internally uses threads that block on requests to file systems, these identifiers consist of the combination of a thread number and a per-thread counter. Furthermore, MINIX 3 already supports crash detection and stateless restarts of system operating processes [30]. Thus, we only had to implement a recovery procedure in VFS: if it detects that a namespace module has been restarted, it first issues a number of requests that restore the previous state in the namespace module with respect to open files and mount points. It then instructs each thread waiting for a reply from that namespace module, to resend its request. This is done serially; if a repeated call triggers a new crash, VFS fails the corresponding application call with an I/O error. After reissuing all requests, normal operation resumes.

C. Case study: the POSIX namespace module

As our first case study, we modified the original POSIX namespace module of the Loris stack to incorporate proper transaction support. Internally, this namespace module consists of three sublayers: 1) a *dispatch sublayer*, which receives VFS requests, dispatches worker threads for them, and sends replies; 2) a *logic sublayer*, which implements the handlers for each of the VFS request types; and, 3) a *caching sublayer*, which consists of caches for its stored metadata (directory data, file attributes, in-use object identifiers, open deleted files). The adaptation of this module to the transaction system allowed us to assess the development effort of applying the transaction system to a module that has not been designed with transactions in mind. In particular, we wanted to know to which extent the "heart" of the module was affected: the logic sublayer. In addition, this use case allowed us to measure the performance impact on a primary namespace module.

We started by adding basic support for transactions. Each worker thread now starts a transaction before invoking a request handler. All object modification operations performed by the handler are added to the current thread's transaction.

Upon return from the handler, the transaction is committed, unless the request failed or the transaction is empty. The POSIX namespace module will never make any changes to objects as the result of a failing VFS request, and thus, if the request failed, the transaction is aborted.

We then changed the module to meet the additional requirements listed in Sec. III-E. To satisfy the first requirement, we made the caching sublayer write-through, thus making its caches flush their changes immediately as part of the current transaction. To cover the second and third requirements, we decided to update these caches optimistically during the execution of request handlers. This way, we avoided changing the request handlers to track their own uncommitted changes. The only code change to the logic sublayer here was to allow rollback of changes to file attributes, since these were simply assignments to fields in a structure. We wrapped such assignments in macros that create a shadow record of the original value upon the first assignment in each request.

For the fourth requirement, we changed the namespace module to maintain records that contain the necessary information to skip repeated requests. We call such records *request recovery records*. Each record corresponds to a particular VFS request, and contains the request identifier, the reply fields expected by VFS for the request (e.g., an updated file size in the case of a "file write" request), and any file open count change that resulted from the request. We had to add a small number of calls to the logic sublayer to track the latter.

In order to store these records across requests, we changed the namespace module to create an extra object in the object store whenever it is started. The object is destroyed upon clean shutdown. Each request writes the corresponding request recovery record to this object as part of its transaction.

After a crash, the namespace module is restarted, and it will find that the object still exists. It then reloads the records from the object. Subsequently, while processing VFS requests, it compares each incoming request identifier to its records in memory. If there is match, the request is skipped: any file open count change in the record is reapplied, and a success code is sent back to VFS along with the stored reply fields in the record. Any memory copies to VFS or the application (e.g., the data for a "file read" request) are performed before the transaction is committed[1]; these need not be saved or repeated.

Maintaining request recovery records incurs little extra overhead. By using the VFS thread number embedded in the request identifier, we can aggressively recycle record entries in the object. As a result, the object is small in size and updated frequently, and thus expected to remain in the cache layer's (RAM) cache at all times. Therefore, updating a record as part of an existing transaction is cheap. Requests that end up in failure will not commit their transaction, and read-only operations will not create a transaction at all. Thus, no record needs to be saved in these two cases: the requests can safely be repeated. Therefore, maintaining these records never requires extra transactions.

In summary, fulfilling the first three requirements required changes to the caching sublayer only, except for small changes in the logic sublayer to be able to roll back updates. Fulfilling the fourth requirement required changes to the dispatch sublayer only, except for a few added calls to track file open count changes in the logic sublayer. Overall we did indeed not have to make substantial changes to the logic sublayer.

D. Case study: the HDF5 namespace module

For the second case study, we developed a new extension namespace module from scratch. This namespace module allows one to explore and manage the contents of a scientific data file, by mounting a representation of its internal hierarchy in the local system's general file hierarchy. While we do not expect our implementation to be used for the *generation* of such files, we believe that offering access to the contents of such a file through the normal POSIX API will help users in *maintenance* of such files, allowing them to use existing and familiar POSIX tools to not only explore the file, but also to make corrections to the file's internal organization.

For this module, we chose HDF5, a commonly used file format for scientific data [6]. HDF5 files are internally structured as filesystem-like hierarchies, with support for regular data files (*data spaces*), directories (*groups*), soft and hard links, an object attribute system much like POSIX extended attributes, and so on. The contents of the data spaces do not always map well to the standard byte-oriented POSIX `read` and `write` calls, but we expect such an extension module to be used for managing the hierarchy rather than the actual data.

We based our implementation on the official HDF5 open-source library implementation [6]. This library is single threaded, and thus, so is our namespace module. We added a relatively thin layer that maps the VFS requests to HDF5 operations. In addition, the namespace module intercepts all I/O calls made by the library, and converts them into Loris operations on the underlying object. The namespace module can be mounted by specifying a HDF5 file and a mount point, and then operates below VFS and on top of the cache layer, next to any other namespace module in the namespace layer.

After completing an initial version, we added basic transaction support, and made the module recoverable by implementing the requirements from Sec. III-E. Satisfying the first requirement was not as straightforward as we hoped. While the library provides a function to flush its caches to the underlying file (`H5Fflush`), as it turned out, this function does not guarantee that the file can be reopened in read-write mode afterwards. Thus, we had to resort to making the library close and reopen the underlying file between each two requests. To lessen the performance impact of the library's resulting constant file data reloads, we added a small read cache to the I/O handling sublayer of the namespace module. Again, optimistic updating of this cache fulfills the third requirement.

Meeting the second requirement was problematic at a more fundamental level. Since all intercepted I/O calls have to be deferred until transaction commit time, the I/O interception

[1]The transaction may still fail, but the POSIX standard does not require that application memory buffer contents be preserved upon call failure.

code has to report preliminary success to the library. If the transaction then fails in the cache layer, there is no longer a way to revert internal state changes in the library that resulted from the earlier perceived success. Given that transaction failures in the cache layer are rare and hard to deal with sensibly, we opted for a drastic solution that works in all cases: upon getting a transaction failure from the cache layer, the namespace module sends an error back to VFS, and then purposely crashes. As a result, the library will reload the pre-request state from the underlying object upon restart, and VFS will not repeat the request; operation can then continue.

We covered the fourth requirement with a similar request recovery record system as for the POSIX namespace module. We avoid creating a separate object in the lower storage layer, instead storing the information in a temporary data space within the HDF5 file. For this namespace module, the records contain some additional information. For example, the library may make changes to the underlying file even when it fails a particular call. Therefore, we also have to create a request recovery record for failing requests, and thus, each record has to contain the resulting error code as well.

In summary, this case study shows that it is feasible for an extension namespace module to meet the requirements for recovery, even when embedding a library of which the internals are largely unknown. We expect that our approach can be reused in many other extension namespace modules.

V. Evaluation

In this section, we evaluate our work. In Sec. V-A, we evaluate the performance of both the the POSIX namespace module and the HDF5 module. In Sec. V-B, we perform fault injection experiments to evaluate the robustness of both modules.

A. Performance

1) The POSIX *namespace module:* We started by subjecting the POSIX namespace module to microbenchmarks (omitted due to lack of space). Initial performance results were not impressive; this turned out to be the effect of file access time updates. These updates resulted in generation of a transaction for each (otherwise read-only) "file read" request, and this caused high interprocess communication (IPC) overheads. We believe that maintaining accurate access times is not essential in most environments, and in all experiments we turned file access times updates off. With this change, all microbenchmarks showed low overheads.

We then ran a number of macrobenchmarks, on four different versions of the POSIX namespace module. First, the unmodified version forms our baseline for the benchmarks (*Original*). Second, we changed the module's caching sublayer to flush all object modifications down to the cache layer within the scope of the request (*No writeback*). Thus, this version shows the overhead of the extra operations resulting from the immediate flushing. Third, we added transaction tracking and rollback support to the namespace module (*Transactions*). These changes are expected to add some more overhead. However, the transaction system now bundles all modifying operations of each request into a single transaction, which is sent down as a unit to the cache layer. As a result, this version should have reduced IPC overhead. Fourth and finally, we added support for request recovery records, thus fulfilling all requirements to allow for crash recovery (*Recoverable*).

We used the following benchmarks and configurations: a MINIX 3 source compilation in a chroot environment; an OpenSSH build test which unpacks, configures, and builds OpenSSH, also in a chroot environment; PostMark, with 80K transactions on 40K files in 10 directories, 4–28KB file sizes, and 4K I/O sizes; FileBench File Server, with its default configuration, run for 30 minutes at once; and, FileBench Web Server, modified to access its files using a Zipf distribution in order to be more realistic, also run for 30 minutes.

The experiments were conducted on an Intel Core 2 Duo E8600 PC, with 4GB of RAM, and a 500GB 7200RPM Western Digital Caviar Blue (WD5000AKS) SATA hard drive, running MINIX 3.2.1. The tests were run with 1GB of cache memory in the cache layer, and on the first 32GB of the disk. For the PostMark and File Bench tests, we consider the run phase only. We report the average of at least ten runs.

The results are shown in Table I. Compared to the original namespace module, the recoverable version has an overhead of 0–2% across the benchmarks, with some benchmarks even showing a small performance improvement. Removing writeback caching had the biggest impact on PostMark, but the transactions effectively canceled this out by reducing IPC overhead. Reduced IPC overheads also explain the other small performance improvements; MINIX 3 is not particularly optimized in this regard. As predicted, maintenance of request recovery records added no significant overhead in any of the benchmarks. Overall, we believe that the runtime overheads are sufficiently low for a process crash recovery system for the primary namespace module of storage stack.

2) The HDF5 namespace module: The HDF5 namespace module was written from scratch and primarily intended for human-driven maintenance. However, our basic support for reading and writing data spaces proved sufficient to run PostMark on it. Thus, we were able to get some performance measurements here as well. We tested five versions: the initial version (*Original*); a version that calls `H5Fflush` to flush the library caches to the underlying file after each request (*Flush*), which as we noted is not sufficient to retain read-write consistency; a version that reopens the underlying file between requests to flush all changes (*Reopen*); a version that adds transaction support to that (*Transactions*), and the final version that also adds request recovery records (*Recoverable*).

The results are shown in Table II. As can be seen, constantly reopening the underlying file results in a serious performance penalty. Our tests show that almost all this time is spent by the library on creating and destroying internal data structures– it is simply not optimized for this kind of usage. If calling `H5Fflush` had been sufficient, this overhead would have been limited to about 17%. Thus, we conclude that while it is possible to meet recovery requirements in an extension

TABLE I
Macrobenchmark performance of the POSIX namespace module, comparing four versions using five benchmarks, and showing both absolute and relative numbers.

Benchmark	Unit	Better if..	Original	No writeback	Transactions	Recoverable
MINIX 3 build	seconds	lower	703 (1.00)	701 (1.00)	698 (0.99)	698 (0.99)
OpenSSH build	seconds	lower	532 (1.00)	536 (1.01)	541 (1.02)	545 (1.02)
PostMark	trans./sec.	higher	825 (1.00)	776 (0.94)	823 (1.00)	825 (1.00)
File Server	IOPS	higher	1265 (1.00)	1244 (0.98)	1243 (0.98)	1279 (1.01)
Web Server (Zipf)	IOPS	higher	12915 (1.00)	12973 (1.00)	13044 (1.01)	13067 (1.01)

namespace module even if it uses a preexisting library as is, if this library has not been optimized for these requirements, the resulting performance may suffer. We maintain that performance is not critical for an extension namespace module, unless it is intended as primary means of access to the file–in that case, it is worth optimizing any included library as well.

B. Reliability

We assessed the reliability of our implementation by performing a number of fault injection experiments on the POSIX and the HDF5 namespace modules. We injected faults in each module while a benchmark was running in a continuous loop.

For the POSIX namespace module, we used two benchmarks. The first is PostMark, which we modified to verify the results of all its calls. As part of this, we made it write known patterns in its `write` calls, and verify the data returned from its `read` calls accordingly. The second is the OpenSSH benchmark, with an added verification step for the compiled binaries at the end. For the HDF5 module, we used the same modified version of PostMark. In addition, since the OpenSSH benchmark expects more from the file system than the HDF5 module can offer (e.g., device nodes), we instead wrote and used a custom benchmark which performs a number of hierarchy manipulation operations in a loop.

We limited ourselves to fail-stop fault injection, as there is no easy way to inject faults that match exactly the failure model from Sec. III-A. In order to maximize fault injection coverage, we injected fail-stop faults into the namespace module process in two different ways: 1) killing the process at random times by sending it a fatal signal ("kill"); 2) using a software fault injection tool to overwrite a limited, random set of CPU instructions in the process with instructions that generate an exception ("swifi"). While the latter eliminates skew introduced by process scheduling, the former adds more multithreading-like coverage of these (single-threaded) benchmark runs, as the module may now also be killed while it is in the middle of performing an operation. We note that without our changes, all fail-stop failures would be fatal to the namespace module.

TABLE II
Macrobenchmark performance of the HDF5 namespace module, comparing five version using PostMark, showing both absolute and relative transactions-per-second numbers.

Original	Flush	Reopen	Transactions	Recoverable
631 (1.00)	530 (0.83)	38 (0.06)	37 (0.06)	36 (0.06)

For each of the combinations, we injected faults 1500 times. For the "swifi" fault injection, we injected 100 faults each time. The results are shown in Table III. In all cases, *all* injections caused the namespace module to crash. More importantly, all crashes were followed by successful recovery, and none of the benchmarks were affected by the fault injection in any way. We believe that this is a good indication that our implementation works and can indeed achieve the intended reliability improvement.

VI. Related work

The closest to our work is Re-FUSE [29], which provides recovery from fail-stop process crashes in FUSE file systems. It logs the system calls (and their results) made by the FUSE file system while processing each file system request. After a crash, the file system is restarted and the pending request is repeated. The file system is then expected to perform exactly the same system calls, for which Re-FUSE replays the original results–after completion, normal operation can resume. Like our solution, Re-FUSE requires that the file system defer no operations across requests. In contrast to our work, Re-FUSE also allows use of nonstorage resources like network connections. However, it requires strict determinism from the file system, and offers no guarantees in multithreaded environments. As stated, we believe that multithreading in particular is not only a requirement for high performance, but also one of the main sources of transient failures. In addition, Re-FUSE can not cleanly recover the underlying resources in the case of a repeating crash; under the same failure assumptions, our system prevents inconsistencies. Compared to both our work and Re-FUSE, other process recovery solutions for file systems either require more resources and processing (e.g., Membrane [31]), or provide more invasive recovery (e.g., CuriOS [26]).

The use of transactions as the basis for recovery from operating system failures is not new (e.g., [27], [28]). While these approaches can recover from failures in larger parts

TABLE III
Fault injection results, showing for each namespace module, benchmark, and types of fault injection, the number of times fault injection took place (I), the number of crashes (C), and the number of successful recoveries (R).

Module	Benchm.	"kill" injection			"swifi" injection		
		I	C	R	I	C	R
POSIX	PostMark	1500	1500	**1500**	1500	1500	**1500**
POSIX	OpenSSH	1500	1500	**1500**	1500	1500	**1500**
HDF5	PostMark	1500	1500	**1500**	1500	1500	**1500**
HDF5	Custom	1500	1500	**1500**	1500	1500	**1500**

of the operating system, we believe they have two main disadvantages: 1) they require extensive changes to the entire operating system, and 2) the generality of the solution makes it harder to apply domain-specific optimizations. For example, we stipulate that the high overheads on file write operations in Akeso [27] are due to extra memory copies necessary for its rollback system. Our system can mostly avoid this.

Other work has explored exposing transactions to applications (e.g., [32], [33]). While such systems add more overhead, it should be possible to combine them with our solution.

Previous work suggested *microrebooting* of isolated components to recover from fail-stop crashes [34]. Our work could be seen as an instance of this concept, although by focusing on the storage stack, we address different aspects of the problem.

ZFS implements a namespace module in the form of its ZFS Posix Layer (ZPL) [13]. It uses per-request transactions from this layer to ensure atomicity of its modifications in the lower layers. However, the ZPL is not a separate process, and we are not aware of work on process crash recovery of ZFS.

VII. Conclusion and future work

In this work, we have described and evaluated a way to improve the reliability of a key component in the next generation of operating system storage stacks. There is however more work to be done in this context.

So far, we have taken the POSIX interface and thus the presence of a VFS layer as a given. In future work, we intend to focus on namespace modules that bypass VFS and expose an API directly to user applications, exploring the requirements for making such modules recoverable as well.

As sketched in Sec. II-A, another future goal is to have multiple primary namespace modules in virtual environments, on top of a single shared object storage layer. In order to deal with hostile modules, we intend to implement on-the-fly verification that transactions retain the overall integrity of the storage system. This is similar to other recent work [35]. However, by working at the object level, we have more semantic information available to achieve this.

Acknowledgment

This research was supported in part by European Research Council Advanced Grant 227874.

References

[1] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka, "A cost-effective, high-bandwidth storage architecture," in *ASPLOS*, 1998, pp. 92–103.

[2] R. Appuswamy, D. C. van Moolenbroek, and A. S. Tanenbaum, "Block-level RAID is dead," in *HotStorage*, 2010.

[3] "FUSE: Filesystem in Userspace," http://fuse.sourceforge.net/.

[4] T. Harter, C. Dragga, M. Vaughn, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "A file is not a file: understanding the I/O behavior of Apple desktop applications," in *SOSP*, 2011.

[5] R. Appuswamy, D. C. van Moolenbroek, and A. S. Tanenbaum, "Loris - A Dependable, Modular File-Based Storage Stack," in *PRDC*, 2010.

[6] The HDF Group, "Hierarchical data format version 5, 2000–2010," http://www.hdfgroup.org/HDF5.

[7] *System Application Program Interface (API) [C Language]: IEEE Std 1003.1-1990 (Revision of IEEE Std 1003.1-1988)*, ser. Information Technology - Portable Operating System Interface. IEEE, 1990.

[8] Cluster File Systems, Inc., "Lustre: A Scalable, High Performance File System," 2002.

[9] A. Azagury, V. Dreizin, M. Factor, E. Henis, D. Naor, N. Rinetzky, O. Rodeh, J. Satran, A. Tavory, and L. Yerushalmi, "Towards an Object Store," in *MSST*, 2003, pp. 165–176.

[10] B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou, "Scalable performance of the Panasas parallel file system," in *FAST*, 2008, pp. 1–17.

[11] "Object-Based Storage Device Commands, ANSI standard INCITS 400-2004," 2004.

[12] M. Seltzer and N. Murphy, "Hierarchical file systems are dead," in *HotOS*, 2009.

[13] "Sun Microsystems, Solaris ZFS file storage solution. Solaris 10 Data Sheets," 2004.

[14] R. van Heuven van Staereling, R. Appuswamy, D. C. van Moolenbroek, and A. S. Tanenbaum, "Efficient, Modular Metadata Management with Loris," in *NAS*, 2011, pp. 278–287.

[15] "Filesystems in userspace: puffs, refuse, FUSE, and more," http://www.netbsd.org/docs/puffs/.

[16] J. S. Heidemann and G. J. Popek, "File-system development with stackable layers," *ACM Trans. Comput. Syst.*, vol. 12, no. 1, pp. 58–89, 1994.

[17] T. Bushnell, "Towards a New Strategy of OS Design," *GNU's Bulletin*, vol. 1, no. 16, 1994.

[18] D. C. van Moolenbroek, R. Appuswamy, and A. S. Tanenbaum, "Integrated End-to-End Dependability in the Loris Storage Stack," in *HotDep*, 2011.

[19] L. Lu, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and S. Lu, "A study of linux file system evolution," in *FAST*, 2013.

[20] T. J. Ostrand and E. J. Weyuker, "The distribution of faults in a large industrial software system," in *ISSTA*, 2002.

[21] L. Hatton, "Reexamining the Fault Density-Component Size Connection," *IEEE Software*, vol. 14, no. 2, pp. 89–97, 1997.

[22] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing," *IEEE Trans. Dependable Secur. Comput.*, vol. 1, no. 1, pp. 11–33, Jan. 2004.

[23] S. R. Kleiman, "Vnodes: An architecture for multiple file system types in Sun UNIX," in *USENIX Summer*, vol. 86, 1986, pp. 238–247.

[24] W. Gu, Z. Kalbarczyk, R. K. Iyer, and Z. Yang, "Characterization of Linux Kernel Behavior under Errors," in *DSN*, 2003.

[25] W.-L. Kao, R. K. Iyer, and D. Tang, "FINE: A Fault Injection and Monitoring Environment for Tracing the UNIX System Behavior under Faults," *IEEE Trans. Software Engineering*, vol. 19, no. 11, pp. 1105–1118, 1993.

[26] F. M. David, E. M. Chan, J. C. Carlyle, and R. H. Campbell, "CuriOS: Improving Reliability through Operating System Structure," in *OSDI*, 2008.

[27] A. Lenharth, V. S. Adve, and S. T. King, "Recovery domains: an organizing principle for recoverable operating systems," in *ASPLOS*, 2009.

[28] C. Giuffrida, L. Cavallaro, and A. S. Tanenbaum, "We crashed, now what?" in *HotDep*, 2010.

[29] S. Sundararaman, L. Visampalli, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Refuse to Crash with Re-FUSE," in *EuroSys*, 2011.

[30] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum, "Failure Resilience for Device Drivers," in *DSN*, 2007, pp. 41–50.

[31] S. Sundararaman, S. Subramanian, A. Rajimwale, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and M. M. Swift, "Membrane: operating system support for restartable file systems," in *FAST*, 2010, pp. 21–21.

[32] R. P. Spillane, S. Gaikwad, M. Chinni, E. Zadok, and C. P. Wright, "Enabling transactional file access via lightweight kernel extensions," in *FAST*, 2009, pp. 29–42.

[33] D. E. Porter, O. S. Hofmann, C. J. Rossbach, A. Benn, and E. Witchel, "Operating system transactions," in *SOSP*, 2009, pp. 161–176.

[34] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox, "Microreboot – A technique for cheap recovery," in *OSDI*, 2004.

[35] D. Fryer, K. Sun, R. Mahmood, T. Cheng, S. Benjamin, A. Goel, and A. D. Brown, "Recon: verifying file system consistency at runtime," in *FAST*, 2012.