The Effects of Metadata Corruption on NFS

Swetha Krishnan, Giridhar Ravipati,
Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Barton P. Miller

Department of Computer Sciences, University of Wisconsin-Madison

{swetha, giri, dusseau, remzi, bart}@cs.wisc.edu

ABSTRACT

Distributed file systems need to be robust in the face of failures. In this work, we study the failure handling and recovery mechanisms of a widely used distributed file system, Linux NFS. We study the behavior of NFS under corruption of important metadata through fault injection. We find that the NFS protocol behaves in unexpected ways in the presence of these corruptions. On some occasions, incorrect errors are communicated to the client application; in others, the system hangs applications or crashes outright; in a few cases, success is falsely reported when an operation has failed. We use the results of our study to draw lessons for future designs and implementations of the NFS protocol.

Categories and Subject Descriptors:
D.4.5 [**Operating Systems**]: Reliability
Subjects: Fault-tolerance

General Terms: Experimentation, Reliability

Keywords: NFS, metadata corruption, fault tolerance, reliability, retry, silent failure, inconsistency

1. INTRODUCTION

Systems fail. Processors compute bad results, despite the vast resources spent on design and test [14]; bits in memory get flipped, despite the presence of ECC protection [13]; disk sectors become inaccessible or corrupt, despite the intricate internal machinery drives contain for reading and writing data [8, 10].

Robust software must be designed to handle failures such as these, and the history of reliable system design and implementation is replete with examples of such systems. Robust storage systems, such as those from EMC and Network Appliance, have many guards built-in against disk failure, including parity, data mirroring, and sophisticated checksums to ensure the integrity of data blocks [7, 21]. Reliable operating systems, such as those built by IBM and Tandem, use process pairs and other techniques to detect and recover from processor or memory faults [2].

Many commodity open-source systems, however, do not have these high levels of paranoia within them. For example, recent work has shown that commodity file systems have difficulty handling partial disk faults [18], leading to unexpected crashes or volume corruption. Similarly, open-source virtual memory systems have been found to exhibit poor behavior in the face of disk faults [1].

Beyond the disk system, the memory subsystem is an increasing source of faultiness in commodity systems, for a variety of reasons. First, inexpensive memory (often found in desktop systems) often forgoes ECC, thus increasing the chance of random bit flips; worse, ECC found in typical commodity memory systems may not be strong enough to mask all errors [13]. Second, buggy software, due to concurrency troubles [9] or poor memory management [6, 15], may overwrite arbitrary memory locations. Third, in a distributed setting, weak network checksums may not provide strong enough protection of packets [11]. Finally, malicious software could surreptitiously alter data fields or network packets, with the hope of causing damage or gaining privilege [4].

In this paper, we examine system robustness in the presence of memory corruption. We focus our study on one important system: the Linux implementation of the Network File System (NFS) protocol [20]. NFS is a widely used distributed protocol, and thus critical to the operation of many computing infrastructures. Further, NFS already has many built-in safeguards against failure; indeed, the original NFS can seamlessly continue operation despite the presence of server crashes [19]. Although it has many known problems [12], we believe studying the behavior of NFS to thus be an interesting endeavor.

To conduct this study, we apply a new memory corruption framework within the Linux client and server NFS stacks. The framework allows great control over the type of corruption we insert; we focus on controlled corruption of metadata values, to see how Linux NFS reacts when these values have unexpected contents.

Our results are as follows. We observe that the Linux NFS client commonly detects errors and invokes retry as a means of recovery; thus, the standard NFS approach of "if it didn't work, try, try again" is often successful in the face of memory corruption. However, in several cases, the system fails silently; the operation fails but success is conveyed to the client application. In another scenario, a corrupted value causes Linux NFS to forgo hard-mount semantics. We also find that the Linux NFS client is particularly sensitive to bad procedure numbers, the presence of which leads to client-side application hangs and kernel panics. We discover a number of inconsistencies in error handling; for example, the same operation returns different errors when attempted multiple times. The lessons from our study may be useful for future designs and implementations of the NFS protocol.

The rest of this paper is organized as follows. In Section 2, we describe our corruption framework, and in Section 3 we present our results. We present the lessons from our study in Section 4, and conclude in Section 5.

StorageSS'07, October 29, 2007, Alexandria, Virginia,USA.

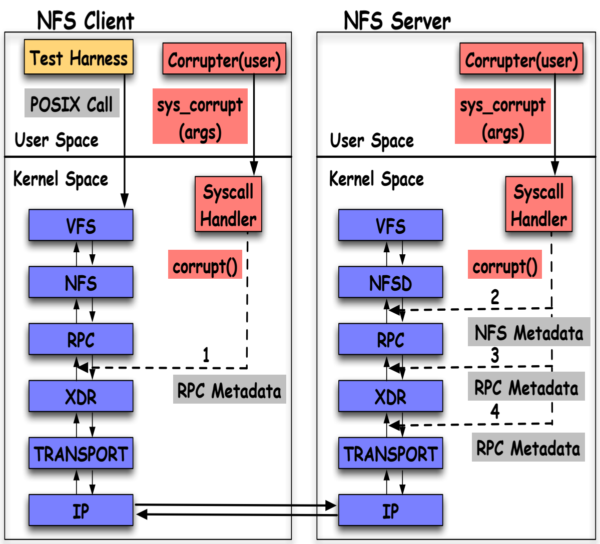

Figure 1: Corruption Framework. ***The diagram shows the four points at which our experiments introduce corruption in the NFS protocol stack.***

POSIX operations	NFS procedures
read()	`lookup`, `read`
write()	`lookup`, `write`, `commit`
access()	`lookup`, `access`
stat()	`lookup`
statfs()	`lookup`, `fsstat`, `fsinfo`
readlink()	`lookup`, `readlink`
chmod()	`lookup`, `setattr`
getdirentries()	`lookup`, `readdir`
create()	`lookup`, `create`
symlink()	`lookup`, `symlink`
link()	`lookup`, `lookup`, `link`
unlink()	`lookup`, `remove`
chdir()	`lookup`
mkdir()	`lookup`, `mkdir`,
rmdir()	`lookup`, `rmdir`
rename()	`lookup`, `lookup`, `rename`

Table 1: Workload for Test Harness. ***The test harness executes the 16 file system POSIX operations shown in the fist column; the second column shows the NFS procedures called by the POSIX operations.***

2. METHODOLOGY

In this section, we discuss our methodology for studying how NFS responds to metadata corruption. We first provide some NFS background, then discuss our corruption model, framework, which metadata fields we corrupt, and finally the experimental environment we employ.

2.1 NFS Background

The NFS protocol provides transparent remote access to shared file systems [5, 19]. NFS is based on the Remote Procedure Call (RPC) protocol [3], which eases the development of client-server applications. The NFS protocol is designed to be machine, operating system, network, and transport-protocol independent. NFS achieves this independence through the use of RPC primitives built on top of the eXternal Data Representation (XDR) layer.

We evaluate the Linux implementation of NFS version 3 [16], as version 3 is still the most widely used. We deploy NFS over UDP in all our experiments.

2.2 Corruption Model

In our corruption model, corruptions are:

- *Carefully targeted.* For each experiment, exactly one NFS or RPC metadata field is corrupted; the corruption is performed for either a single NFS procedure or for all NFS procedures.
- *Persistent.* A retry of the same NFS request experiences the same corrupted value.
- *Not sticky.* Corruption is turned off at the end of each experiment and any NFS calls performed after the experiment do not experience corruption.
- *Isolated.* Corruptions do not affect other parts of the NFS distributed environment, including other NFS clients. Specifically, we inject corruption only for NFS version 3 requests over UDP that originate from our client's IP address.

2.3 Corruption Framework

Figure 1 shows the architecture of our corruption framework. We introduce a *test harness* that exercises NFS through the client-side file system POSIX API and a *corrupter* that injects metadata corruption on both the client and server. We discuss these two modules in more detail.

The *test harness* is a user-level application containing a suite of file system POSIX API calls. As shown in Table 1, the test suite contains 16 POSIX operations, which in turn invoke a total of 18 NFSv3 procedures and exercise the NFS and RPC server and client code in the kernel. Each experiment begins with a newly mounted file system and issues a single POSIX operation (for a single block of a file, when applicable). Note that four NFSv3 procedures not exercised: NFSPROC3_NULL, NFSPROC3_MKNOD, NFSPROC3_READDIRPLUS and NFSPROC3_PATHCONF; we believe these procedures are used relatively infrequently.

The *corrupter* lets us corrupt NFS and RPC metadata at different points in the network stack. Thus, the corrupter allows us to emulate memory errors that could occur due to hardware problems, software bugs, or maliciousness at various layers in the system. In our experiments, we corrupt metadata at four different points in the network stack across the client and server, as shown in Figure 1. In all cases, the corruption is applied as the request or reply is traversing down the stack. The corrupter is implemented by introducing a new system call (*sys_corrupt()*, as shown in Figure 1), that lets us control corruption parameters from user space. Thus, for each experiment, the parameters we control are: corruption state (i.e. turned on or turned off), the metadata field to corrupt, the corrupted value for this metadata field, and the affected NFS procedures. The change of a metadata field value to the value specified through the system call is performed by the *corrupt()* routine shown in Figure 1.

2.4 Corrupted Metadata

Since NFS is layered on top of RPC, the failure policy of NFS interacts with that of RPC as well. Given that NFS and RPC have more than 10 fields of metadata each, exhaustively corrupting each field is beyond the scope of this study. In this paper, we explore the sensitivity of NFS to corruption in five metadata fields: trans-

Metadata	Layer	Point	Values	Description
XID	RPC	4	XID+1, XID-1	Selected by the client to uniquely identify a request-response pair
Direction	RPC	4	Not 1	Value in the header that identifies Call (0) or Reply (1)
Response length	RPC	4	Above and below length	Appended to the RPC reply message buffer
Procedure Number	RPC	1, 3	`read` to `write` `symlink` to `rename`	Procedure number to invoke on the server; found in the request header.
File handle size	NFS	2	Above and below size	Size of the opaque reference to a file or directory

Table 2: Corrupted Metadata. ***The table shows the set of metadata that we have corrupted in our experiments. The columns indicate whether the metadata is applicable to RPC or NFS; the corruption points as defined in Figure 1; the values to which we corrupt the metadata; and finally, a brief description of the purpose of that metadata.***

action id (XID), call/reply direction, response buffer length, procedure number, and file handle size. We focus on this subset of metadata because we believe each is used in critical ways (*e.g.*, for response verification).

Table 2 summarizes where and how each of these five metadata fields is corrupted. For RPC metadata, the corruption is usually performed on the server beneath the XDR layer (*i.e.*, point 4). For NFS metadata, the corruption is performed on the server beneath the NFSD layer (*i.e.*, point 2). For practical reasons, we focus on a subset of the possible values to which the metadata may be corrupted; these values are described in more detail where appropriate.

2.5 Experimental Environment

We conduct all of the experiments described in this paper on Emulab [22]. One machine is configured as an NFS server, with one exported directory tree, while another is set up as an NFS client that performs a hard mount of the exported file system. Each of these nodes run Red Hat Linux 9.0, a 2.4.20 Linux kernel. We use Nfsdump v1.01 to view requests and replies between server and client.

3. EXPERIMENTAL RESULTS

We now describe our results corrupting the metadata fields of XID, direction, response buffer length, procedure number, and file handle size. We present the observed behavior at the NFS client, from both an application and system perspective.

3.1 Transaction ID (XID) Corruption

Our first experiment evaluates the impact of a corrupted XID field in the reply packet sent by the NFS server; this corruption is injected at point 4 in Figure 1. We corrupted the XID to XID+1 and XID-1 independently for each NFS procedure invoked.

Observation 1: *For our workloads, the NFS client detects XID corruption and persistently retries the request.* In our experiments, the NFS client detects that the reply does not correspond to the single request it sent, since the XIDs do not match; hence, the client considers the reply to be invalid, drops the received packet, and retries the original request persistently. We note that the client is not guaranteed to detect XID corruption for all workloads; specifically, for workloads with concurrent NFS requests, if the corrupted XID matches a different request, then the response may be incorrectly matched. In our workload, the client retries until corruption is turned off, at which point it receives a reply with the correct XID and the operation succeeds (we verified the intention of a persistent retry by inspecting the code). Given the procedure-call semantics of the underlying RPC, the policy of persistent retries seems appropriate.

3.2 Call/Reply Direction Corruption

In our second experiment, we corrupted the value of the direction field in the RPC reply header. Under correct operation, the direction field contains a value of '1' to indicate a reply. We corrupted this field both to an invalid value (*i.e.*, 2) and to the value indicating a request (*i.e.*, 0), on the server at point 4 in Figure 1. We tried these experiments for each NFS procedure invoked by our test harness, as well as for a special workload consisting of `mount` (which invokes the NFS procedures `getattr`, `fsstat`, and `fsinfo`).

Observation 2: *The NFS client detects direction field corruption, but only retries either two or four times, despite the fact that the remote file system is hard-mounted.* For some NFS procedure calls, the corrupted direction field in the reply packet causes the client to retry the NFS call two times (`getattr`, `lookup`, `read`, `commit`, `access`, `readlink`, `fsstat`, and `fsinfo`), while for others, the NFS client retries four times (*i.e.*, `write`, `create`, `symlink`, `link`, `remove`, `mkdir`, `rmdir`, `setattr`, `rename`, and `readdir`). The client stops retrying after a few times, which does not match the semantics of the remote file system being hard-mounted.

Observation 3: *If the direction field is corrupted, the error is not always correctly propagated to the application.* In the case of the *statfs()* POSIX call (which invokes the procedures `fsstat` and `fsinfo`), the return code of **success** is incorrectly returned to the client application; however, as expected, if the results of the operation are examined, they are incorrect (*e.g.*, block size = -1, number of blocks = -1, free blocks count= -1). We note that the NFS layer at the client does detect the corruption, since the kernel log shows "nfs_statfs: statfs error = 5". Thus, this is a case when NFS does not propagate the error to the application on the client but *fails silently*. For all other POSIX calls, an error of **EIO** is propagated to the client application.

Observation 4: *During a* `mount` *of the file system, direction field corruption eventually causes the client to switch to NFS version 2 procedures.* Specifically, when the direction field is corrupted during a mount, the client retries the call twice (with the same NFS version 3 procedure, as expected). If these retries fail, the client tries the same call with its version 2 equivalent (*e.g.*, if version 3 `getattr` fails, version 2 `getattr` is issued; if version 3 `fsstat` or `fsinfo` fail, then version 2 `statfs` is tried). Interestingly, the client then permanently switches to only version 2 procedures for all subsequent POSIX calls, even though their version 3 equivalents are untainted. Finally, if the direction field is corrupted for both version 3 and version 2 NFS calls, then the mount fails, after the client retries the version 3 procedure twice and the version 2 procedure five times.

3.3 Response Buffer Length Corruption

The length of the RPC response buffer is maintained in the response buffer structure and is sent to the client. We corrupted the length to values both below and above the original length, at point 4 in Figure 1. We have tested the impact of response buffer length corruption on the *read()*, *write()*, *create()*, *symlink() link()*, *unlink()*, and *stat()* POSIX calls; the results for these cases are all qualitatively similar, so we focus on the *read()* results in our explanation.

Observation 5: *The NFS client only sometimes detects corruption of the response buffer length, depending upon the value to which the length is corrupted. The application sometimes sees success and sometimes failure.* Specifically, the two cases are as follows. First, when the response buffer length is incremented or decremented by only a very small amount, the corruption is never detected by the client and the POSIX operation always succeeds. (Note that the server kernel does detect the corruption before releasing the socket buffer.) Second, when the reported length is decremented more drastically, the corruption is detected and the client performs a number of retries upon failure, depending on what metadata appears truncated. For example, with a *read()* for 1024 bytes (which has an actual response buffer length of 1056 bytes), corruption to values between about 1036 and 8 bytes causes no retries, corruption to values around 5 causes two retries, and to values around 2 causes persistent retries.

Observation 6: *If the POSIX operation is reissued by the client application, the operation may return success (even though the response buffer length remains corrupted). The exact behavior again varies with the corrupted response length.* When the corrupted value is near that of the actual length, then, on the second application attempt of *read()*, the data returned to the application is intact and complete. When the decremented amount is more drastic, the *read()* operation returns success with the number of bytes read as being equal to the number of bytes requested, but in reality, the data is truncated; thus, from the client's perspective there is a *silent failure*. Finally, for *read()* with very small corrupted buffer lengths, the second client attempt fails with an error code of either EPFNOSUPPORT or EIO.

3.4 Procedure Number Corruption

The RPC request contains a field for the procedure number of the NFS procedure whose invocation is being requested. We corrupted this field at two different points in separate experiments. On the client (point 1), we corrupted the procedure number after the RPC layer hands off the request to the XDR layer for encoding the procedure and its arguments; on the server (point 3), we corrupted it after the RPC layer retrieves the procedure number from the header, but before it calls into XDR to decode the arguments corresponding to the procedure. To limit the space of our study, we looked at procedures that have similar arguments since these procedure pairs are more likely to confuse the server when interchanged. We report results from corrupting `read` to `write` and `symlink` to `rename`.

Observation 7: *The NFS client detects but runs into kernel exceptions when the procedure number is corrupted before encoding it in the request.* Corrupting `read` to `write` on the client causes not only the application to terminate with a segmentation fault, but also the kernel to dereference a null pointer. Even if the corruption is turned off and the operation is retried, the application hangs and the remote file system cannot be remounted on this client. Corrupting `symlink` to `rename` makes the system unusable: a kernel paging fault occurs and the kernel continues to panic after a reboot. Thus, corruption from one application can hamper the robustness of the entire client.

Observation 8: *When the same corruption occurs on the server, the effects are less drastic and an error is reported to the client application.* At the server side, the effects are less severe since corruption happens after the procedure number is retrieved but before decoding its arguments. When `read` is corrupted to `write`, the call fails to return a reply to the client, so the client retries the call twice before propagating an error to the application. When `symlink` is corrupted to `rename`, a reply with an error is returned to the `symlink` call; the client does not retry and propagates EBADHANDLE to the application.

3.5 File Handle Size Corruption

The NFS file handle is an opaque reference to a specific file or directory, constructed by the server. We corrupt the size of the file handle returned to the client; the size value is especially important to an NFS version 3 client because version 3 supports variable length file handles. The corruption was injected at the server-side, between the NFSD and RPC layers, at point 2 in Figure 1. The NFS procedures that we test that return a file handle are `lookup`, `create`, `symlink`, and `mkdir`. We changed the file handle size for these four procedures to both lower and higher values from the actual.

Observation 9: *The NFS client detects file handle size corruption and returns an error to the client application in most cases. In a few cases, success is falsely reported to the application, and in few other cases, different error codes are returned on multiple attempts by the application.* Both `lookup` and `create` return an error if the file handle size is less than the actual size. For a *read()* POSIX call (which invokes `lookup`), the error is set to EIO the first time and EBADHANDLE on the second application attempt. For a *write()*(which invokes `create`) if the corruption changes the size by a small amount, then the corruption goes undetected and the *write()* succeeds; however, an error is returned to the application on closing the file. For `symlink` and `mkdir` with a bad file handle length, success is reported to the client application, but subsequent *readlink()* and *getdirentries()* operations fail with EIO and EBADHANDLE respectively.

4. LESSONS LEARNED

From our experimental results and observations, we have identified the following lessons for error handling in NFS.

Lesson 1: *"Giving up" violates hard mounting semantics, and may not be the best way to tackle failure.* For example, we saw that the client retries persistently given XID corruption, but does not do so given direction corruption. Overall, we saw that corrupting different fields led to different numbers of retries. If corruption is transient, operations may succeed if persistently retried.

Lesson 2: *When different versions are available, try them all.* This lesson was illustrated when the mount call invoked NFS version 2 routines when the version 3 routines failed. It would be interesting to try version 2 routines for POSIX calls other than mount.

Lesson 3: *Inconsistency in behavior indicates that the error handling code in NFS is likely diffused.* Inconsistent error handling was seen in many cases, but corrupting the response buffer length to different values led to some of the most varied results; this inconsistency makes the policy hard to understand and predict. To expose and enforce a clear and consistent policy, it might help to consolidate error handling in a central location.

Lesson 4: *Too much trust of the NFS server can be harmful; therefore it would be wiser for the NFS client to do a greater degree of sanity checking.* If the NFS design had well-defined formats for messages or incorporated checksums for the metadata and data in the packet, the client would not have to rely on the length value returned by the server for the reply buffer. In the current design, the client ends up returning failure or repeatedly retrying requests, in spite of the fact that the reply buffer is actually intact and accessible.

Lesson 5: *Inconsistent error codes may confuse client applications.* We saw that slight differences in corruption values led to different error codes being returned to clients. Although EIO is adequately returned in most cases, EPFNOSUPPORT and EBADHANDLE were inappropriately returned when the response length and the procedure number, respectively, were corrupted; these error codes may cause applications to respond incorrectly.

Lesson 6: *Careful sanity checking of pointers between layers of the NFS protocol stack could avoid crashes.* The client-side corruption results indicate that between the RPC and XDR layers, pointers are not checked and dereferencing a NULL pointer drives the kernel into unstable states. Since XDR uses no data typing, there is an implicit assumption that the application parsing an XDR stream knows what type of data to expect [5]. Therefore, NFS should verify all pointers before passing them to XDR.

Lesson 7: *When in doubt, it is better to report "failure" than "success".* In many cases (such as those explained in Observation 6 for response buffer length corruption and Observation 9 for file handle size corruption), we saw that NFS incorrectly returns success to an application when the operation actually failed. A "false" success not only misinforms the user, because subsequent operations will return failure, but may also render important data inaccessible. Hence, it is important that errors are propagated by the NFS client all the way to the application.

5. CONCLUSIONS

We have conducted a study of how the NFS server and client cope with corrupted metadata. Our important observations are that NFS does not always deal well with metadata corruption. We see improper errors, or false notification of success being conveyed to the client application, and also see the client's kernel being adversely affected in a few cases. We feel that the main lessons to be gained from this study are that the NFS protocol should more carefully limit the amount of trust between the server and client, have both ends do more careful sanity checking, have error handling unified to make it consistent, report errors more carefully to applications, and issue retries instead of giving up after a failed operation.

This study is by no means complete. First, our study is limited to UDP as the transport, and it remains to be seen how NFS over TCP reacts to metadata corruption. Another interesting study would be to repeat these experiments for NFS Version 4 [17], which is designed to have better error recovery semantics and improved security. Also, there are other interesting pieces of metadata (such as the write verifier) which we have not stressed; doing so would be of value. Experiments with more than one NFS client may also yield interesting results. However, we believe that our preliminary study has already revealed interesting insights into NFS failure policies; we hope that such insights will help developers take countermeasures to improve the reliability of NFS as a distributed file system.

6. REFERENCES

[1] L. N. Bairavasundaram, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Dependability Analysis of Virtual Memory Systems. In *DSN-2006*, Philadelphia, PA, June 2006.

[2] W. Bartlett and L. Spainhower. Commercial Fault Tolerance: A Tale of Two Systems. *IEEE Transactions on Dependable and Secure Computing*, 1(1):87–96, January 2004.

[3] A. Birrell and B. Nelson. Implementing Remote Procedure Calls. In *SOSP '83*, October 1983.

[4] E. Brewer, P. Gauthier, I. Goldberg, and D. Wagner. Basic Flaws in Internet Security and Commerce. http://www.interesting-people.org/archives/interesting-people/199510/msg00030.html, 1995.

[5] B. Callaghan. *NFS Illustrated*. Addison-Wesley, Inc., 2000.

[6] J. Chapin, M. Rosenblum, S. Devine, T. Lahiri, D. Teodosiu, and A. Gupta. Hive: Fault Containment for Shared-Memory Multiprocessors. In *SOSP '95*, Copper Mountain, CO, December 1995.

[7] EMC Corporation. Symmetrix Enterprise Information Storage Systems. http://www.emc.com, 2002.

[8] G. F. Hughes and J. F. Murray. Reliability and Security of RAID Storage Systems and D2D Archives Using SATA Disk Drives. *ACM Transactions on Storage*, 1(1):95–107, February 2005.

[9] M. Isard and A. Birrell. Automatic Mutual Exclusion. In *HotOS '07*, San Diego, CA, May 2007.

[10] H. H. Kari, H. Saikkonen, and F. Lombardi. Detection of Defective Media in Disks. In *The IEEE International Workshop on Defect and Fault Tolerance in VLSI Systems*, pages 49–55, Venice, Italy, October 1993.

[11] P. Karn, C. Bormann, G. Fairhurst, D. Grossman, R. Ludwig, J. Mahdavi, G. Montenegro, J. Touch, and L. Wood. Advice for Internet Subnetwork Designers. http://www.ietf.org/rfc/rfc3819.txt, July 2004.

[12] O. Kirch. Why NFS Sucks. In *Proceedings of the Linux Symposium Volume Two, Ottawa, Canada*, July 2006.

[13] D. Milojicic, A. Messer, J. Shau, G. Fu, and A. Munoz. Increasing relevance of memory hardware errors: a case for recoverable programming models. In *9th ACM SIGOPS European Workshop*, Kolding, Denmark, Sept. 2000.

[14] T. Nicely. Bug in the pentium fpu. http://www.trnicely.net/pentbug/bugmail1.html, Oct. 1994.

[15] G. Novark, E. D. Berger, and B. G. Zorn. Exterminator: Automatically Correcting Memory Errors with High Probability. In *PLDI '07*, San Diego, CA, June 2007.

[16] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz. NFS Version 3 Design and Implementation. In *Proceedings of the Summer 1994 USENIX Conference, Boston, Massachusetts*, June 1994.

[17] B. Pawlowski, S. Shepler, C. Beame, B. Callaghan, M. Eisler, D. Noveck, D. Robinson, and R. Thurlow. The NFS Version 4 Protocol. http://www.connectathon.org/talks97/index.html, 1997.

[18] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. IRON File Systems. In *SOSP '05*, pages 206–220, Brighton, UK, October 2005.

[19] R. Sandberg. The Design and Implementation of the Sun Network File System. In *Proceedings of the 1985 USENIX Summer Technical Conference*, pages 119–130, Berkeley, CA, June 1985.

[20] C. M. Smith. Linux NFS Overview. http://nfs.sourceforge.net/, 2007.

[21] R. Sundaram. The Private Lives of Disk Drives. http://www.netapp.com/go/techontap/matl/sample/ 0206tot_resiliency.html, February 2006.

[22] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *OSDI '02*, pages 255–270, Boston, MA, December 2002.