DOI:10.1145/2788401

Article development led by acmqueue
queue.acm.org

Rethinking the fundamental abstractions of the file system.

BY T.S. PILLAI, V. CHIDAMBARAM, R. ALAGAPPAN, S. AL-KISWANY, A.C. ARPACI-DUSSEAU, AND R.H. ARPACI-DUSSEAU

Crash Consistency

THE READING AND writing of data, one of the most fundamental aspects of any von Neumann computer, is surprisingly subtle and full of nuance. For example, consider access to a shared memory in a system with multiple processors. While a simple and intuitive approach known as *strong consistency* is easiest for programmers to understand,[14] many weaker models are in widespread use (for example, x86 total store ordering[22]); such approaches improve system performance, but at the cost of making reasoning about system behavior more complex and error prone. Fortunately, a great deal of time and effort has gone into thinking about such memory models,[24] and, as a result, most multiprocessor applications are not caught unaware.

Similar subtleties exist in local file systems—those systems that manage data stored in your desktop computer, on your cellphone,[13] or that serve as the underlying storage beneath large-scale distributed systems such as Hadoop Distributed File System (HDFS).[23] Specifically, a pressing challenge for developers trying to write portable applications on local file systems is *crash consistency* (that is, ensuring application data can be correctly recovered in the event of a sudden power loss or system crash).

Crash consistency is important. Consider a typical modern photo-management application such as iPhoto, which stores not only the photos a user takes, but also information relevant to a photo library, including labels, events, and other photo metadata. No user wants a system that loses photos or other relevant information simply because a crash occurs while the photo-management application is trying to update its internal database.

Much of the burden today in ensuring crash consistency is placed on the application developer, who must craft an *update protocol* that orchestrates modifications of the persistent state of the file system. Specifically, the developer creates a carefully constructed sequence of *system calls* (such as file writes, renames, and other file-system calls) that updates underlying files and directories in a recoverable way. The correctness of the application, therefore, inherently depends on the semantics of these system calls with respect to a system crash (that is, the *crash behavior* of the file system).

Unfortunately, while the standardized file-system interface has been in widespread use for many years, application-level crash consistency is currently dependent on intricate and subtle details of file-system behavior. Either by design or by accident, many modern applications depend on particular file-system implementation details and thus are vulnerable to unexpected behaviors in response to system crashes or power losses when run on different file systems or with different configurations.

Recent research, including work performed by our group at the University of Wisconsin–Madison,[21] as well as elsewhere,[29] has confirmed that crashes are problematic: many applications

(including some widely used and developed by experienced programmers) can lose or corrupt data on a crash or power loss. The impact of this reality is widespread and painful: users must be prepared to handle data loss or corruption,[15] perhaps via time-consuming and error-prone backup and restore; applications might tailor their code to match subtle file-system internals, a blatant violation of layering and modularization; and adoption of new file systems is slowed because their implementations do not match the crash behavior expected by applications.[6] In essence, the file-system abstraction, one of the basic and oldest components of modern operating systems, is broken.

This article presents a summary of recent research in the systems community that both identifies these crash consistency issues and points the way toward a better future. First a detailed example illustrates the subtleties of the problem. We summarize the state of the art, illustrating the problems we (and others) have found are surprisingly widespread. Some of the promising research in the community aims to remedy these issues, bringing new thinking and new techniques to transform the state of the art.

An Example

Let's look at an example demonstrating the complexity of crash consistency: a simple database management system (DBMS) that stores its data in a single file. To maintain transactional atomicity across a system crash, the DBMS can use an update protocol called *undo logging*: before updating the file, the DBMS simply records those portions of the file that are about to be updated in a separate log file.[11] The pseudocode is shown in Figure 1; `offset` and `size` correspond to the portion of the `dbfile` that should be modified, and whenever the DBMS is started, the DBMS rolls back the transaction if the log file exists and is fully written (determined using the size field). The pseudocode in Figure 1 uses POSIX system calls (POSIX is the standard file-system interface used in Unix-like operating systems). In an ideal world, one would expect the pseudocode to work on all file systems implementing the POSIX interface. Unfortunately, the pseudocode does not work on *any* widely used file-system configuration; in fact, it requires a different set of measures to make it work on each configuration.

Figure 1. Incorrect undo-logging pseudocode.

```
creat(log);
# Making a backup in the log file
write(log, "<offset>,<size>,<data>");
write(dbfile, offset, data); # Actual Update
unlink(log); # Deleting the log file
```

Figure 2. Undo-logging pseudocode that works correctly in Linux file systems.

```
creat(log);
write(log, "<offset>,<chksum>,
        <size>,<data>");
fsync(log);
fsync(./);
write(dbfile, offset, data);
fsync(dbfile);
unlink(log);
fsync(./);
```

- `<chksum>`: Log file can end up with garbage, in ext2, ext3-wb, ext4-wb
- `fsync(log);`: write(log) and write(dbfile) can re-order in all considered configurations
- `fsync(./);`: creat(log) can be re-ordered after write (dbfile), according to warnings in Linux manpage. Occurs on ext2.
- `fsync(dbfile);`: write(dbfile) can re-order after unlink(log) in all considered configurations except ext3's default mode
- `fsync(./);` (last): If durability is desired, in all considered configurations

Because file systems buffer writes in memory and send them to disk later, from the perspective of an application most file systems can *reorder* the effects of system calls before persisting them on disk. For example, with some file systems (ext2, ext4, xfs, and btrfs in their default configurations, but not ext3), the deletion of the log file can be reordered before the write to the database file. On a system crash in these file systems, the log file might be found already deleted from the disk, while the database has been updated partially. Other file systems can persist a system call partially in seemingly nonsensical ways: in ext2 and nondefault configurations of ext3 and ext4, while writing (appending) to the log file, a crash might leave garbage data in the newly appended portions of the file; in such file systems, during recovery, one cannot differentiate whether the log file contains garbage or undo information.

Figure 2 shows the measures needed for undo logging to work on Linux file-system configurations ("./" refers to the current directory); the red parts are the additional measures needed. Comments in the figure explain which measures are required by different file systems: we considered the default configurations of ext2, ext3, ext4, xfs, and btrfs, and the data=writeback configuration of ext3/4 (denoted as ext3-wb and ext4-wb). Almost all measures simply resort to using the `fsync()` system call, which flushes a given file (or directory) from the buffer cache to the disk and is used to prevent the file system from reordering updates. The `fsync()` calls can be arbitrarily costly, depending on how the file system implements them; an efficient application will thus try to avoid `fsync()` calls when possible. With only a subset of the `fsync()` calls, however, an implementation will be consistent only on some file-system configurations.

Note that it is not practical to use a verified implementation of a single update protocol across all applications; the update protocols found in real applications vary widely and can be more complex than in Figure 2. The choice can depend on performance characteristics; some applications might aim for sequential disk I/O and prefer an update protocol that does not involve seeking to different portions of a file. The choice can also depend on usability characteristics. For example, the presence of a separate log file unduly complicates common workflows, shifting the burden of recovery to include user involvement. The choice of update protocol is also inherently tied to the application's concurrency mechanism and the format used for its data structures.

Current State of Affairs

Given the sheer complexity of achieving crash consistency among different file systems, most developers write incorrect code. Some applications (for example, Mercurial) do not even try to handle crashes, instead assuming that users will manually recover any data lost or corrupted as a result of a crash. While application correctness depends on the intricate crash behavior of file systems, there has been little formal discussion on this topic.

Two recent studies investigate the correctness of application-level crash consistency: one at the University of Wisconsin–Madison[21] and the other at Ohio State University and HP Labs.[29] The applications analyzed include distributed systems, version-control systems, databases, and virtualization software; many are widely used applications written by experienced developers, such as Google's LevelDB and Linus Torvalds's Git. Our study at the University of Wisconsin–Madison found more than 30 vulnerabilities exposed under widely used file-system configurations; among the 11 applications studied, seven were affected by data loss, while two were affected by silent errors. The study from Ohio State University and HP Labs had similar results: they studied eight widely used databases and found erroneous behavior in all eight.

For example, we found that if a file system decides to reorder two `rename()` system calls in HDFS, the HDFS namenode does not boot[2] and results in unavailability. Therefore, for portable crash consistency, `fsync()` calls are required on the directory where the `rename()` calls occur. Presumably, however, because widely used file-system configurations rarely reorder the `rename()` calls, and Java (in which HDFS is written) does not directly allow calling `fsync()` on a directory, the issue is currently ignored by HDFS developers.

As another example, consider LevelDB, a key-value store that adds any inserted key-value pairs to the end of a log file. Periodically, LevelDB switches to a new log file and compacts the previous log file for faster record retrieval. We found that, during this switching, an `fsync()` is required on the old log file that is about to be compacted;[19] otherwise, a crash might result in some inserted key-value pairs disappearing.

Try It Yourself!

Many application-level crash-consistency problems are exposed only under uncommon timing conditions or specific file-system configurations, but some are easily reproduced. As an example, on a default installation of Fedora or Ubuntu with a Git repository, execute a git-commit, wait for five seconds, and then pull the power plug; after rebooting the machine, you will likely find the repository corrupted. Fortunately, this particular vulnerability is not devastating: if you have a clone of the repository, you likely can recover from it with a little bit of work. (Note: do not do this unless you are truly curious and will be able to recover from any problems you cause.)

The Unspoken Agreement

What *can* applications rely on? File-system developers seem to agree on two rules that govern what information is preserved across system crashes. The first is subtle: information already on disk (file data, directory entries, file attributes, among others) is preserved across a system crash, unless one explicitly issues an operation affecting it.

The second rule deals with `fsync()` and similar constructs (`msync()`, `O _ SYNC`, and so on) in Unix-like operating systems. An `fsync()` on a file guarantees the file's data and attributes are on the storage device when the call returns, but with some subtleties. A major subtlety with `fsync()` is the definition of *storage device*: after information is sent to the disk by `fsync ()`, it can reside in an on-disk cache and hence can be lost during a system crash (except in some special disks). Operating systems provide ad hoc solutions to flush the disk cache *to the best of their ability*; since you might be running atop a fake hard drive,[8] nothing is promised. Another subtlety relates broadly to directories: directory entries of a file and the file itself are separate entities and can each be sent separately to the disk; an `fsync()` on one does not imply the persistence of others.

Best Practices for Application Developers

Developers can alleviate the problem of crash consistency within their applications by following these recommended practices:

***Use a library.* Implementing consistency directly atop the file-system interface is like pleading insanity in court: you do it only if you have no other choice. A wiser strategy is to use a library, such as SQLite, that implements crash consistency below your application whenever possible.**

***Document guarantees and requirements.* Consistency guarantees provided by applications can be confusing; some developers can be unclear about the guarantees provided by their own applications. Documenting file-system behaviors that the application requires to maintain consistency is more complicated, since both application developers and users are often unclear about file-system behavior. The best documentation is a list of supported file-system configurations.**

***Test your applications.* Because of the confusing crash behavior exhibited by file systems, it is important to test applications. Among the tools publicly available for finding application crash vulnerabilities, ALICE[21] has been used successfully for testing eleven applications; ALICE also clearly shows which program lines lead to a vulnerability. The public version of ALICE, however, does not work with `mmap()` memory and some rare system calls. There is another tool designed for testing file systems[9] that works with any application that runs on Linux, but it is less effective.**

Many vulnerabilities arise because application developers rely on a set of popular beliefs to implement crash consistency. Unfortunately, much of what seems to be believed about file-system crash behavior is not true. Consider the following two myths:

▸ **Myth 1: POSIX defines crash behavior.** POSIX[17] defines the standard file-system interface (`open`, `close`, `read`, and `write`) exported by Unix-like operating systems and has been essential for building portable applications. Given this, one might believe that POSIX requires file systems to have a reasonable and clearly defined response to crashes, such as requiring that directory operations be sent to the disk in order.[18] Unfortunately, there is little clarity as to what exactly POSIX defines with regard to crashes,[3,4] leading to much debate and little consensus.

▸ **Myth 2: Modern file systems require and implement in-order metadata updates.** Journaling, a common technique for maintaining file-system metadata consistency, commits different sets of file-system metadata updates (such as directory operations) as atomic transactions. Journaling is popular among modern file systems and has traditionally committed metadata updates in order;[12] hence, it is tempting to assume modern file systems guarantee in-order metadata updates. Application developers should not assume such guarantees, however. Journaling is an internal file-system technique; some modern file systems, such as btrfs, employ techniques other than journaling and commonly reorder directory operations. Furthermore, even file systems that actually use journaling have progressively reordered more operations while maintaining internal consistency. Consider ext3/4: ext3 reorders only overwrites of file data, while ext4 also reorders file appends; according to Theodore Ts'o, a maintainer of ext4, future journaling file systems might reorder more (though unlikely with ext4).

Should file-system developers be blamed for designing complicated file systems that are unfavorable for implementing crash consistency? Some complex file-system behaviors can (and should) be fixed. Most behaviors that make application consistency difficult, however, are essential for general-purpose file systems.

Recent research has confirmed that crashes are problematic: many applications (including some widely used and developed by experienced programmers) can lose or corrupt data on a crash or power loss.

To illustrate, consider reordering, the behavior that is arguably the least intuitive and causes the most crash-consistency vulnerabilities. In our study, a file system that provided in-order operations (and some minimal atomicity) exposed only 10 vulnerabilities, all of minor consequences; in comparison, 31 were exposed in btrfs and 17 in ext4. In current environments with multiple applications running simultaneously, however, a file system requires reordering for good performance. If there is no reordering, `fsync()` calls from important applications will be made to wait for writes from nonessential tasks to complete. Indeed, ext3 in its default configuration provides an (almost) in-order behavior, but has been criticized for unpredictably slow `fsync()` calls.[7]

Moving Forward

Fortunately, not all is bleak in the world of crash consistency, and recent research points toward a number of interesting and plausible solutions to the problems outlined in this article. One approach is to help developers build correct update protocols. At least two new open source tools are available publicly for consistency testing (though neither is mature yet): ALICE,[20] the tool created for our research study at the University of Wisconsin–Madison, and a tool designed by Linux kernel developers[9] for testing file-system implementations. ALICE is more effective for testing applications since it verifies correctness on a variety of simulated system crashes for a given application test case. In contrast, the kernel tool verifies correctness only on system crashes that occur with the particular execution path traversed by the file system during a run of the given test case.

Two other testing tools are part of recent research but are not yet publicly available: BOB[21] from our study, and the framework used by researchers from Ohio State University and HP Labs.[29] Both of these are similar to the kernel tool.

A second approach for better application crash consistency is for file systems themselves to provide better, more easily understood abstractions that enable both correctness and high performance for applications. One solution would be to extend and improve the current file-system interface (in the Unix world or in Windows); however, the interface has been built upon many years of experience and standardization, and is hence resistant to change.[16] The best solution would provide better crash behavior with the current file-system interface. As previously explained, however, in-order updates (that is, better crash behavior) are not practical in multitasking environments with multiple applications. Without reordering in these environments, the performance of an application depends significantly on the data written by other applications in the background and will thus be unpredictable.

There is a solution. Our research group is working on a file system that maintains order only within an application. Constructing such a file system is not straightforward; traditional file systems enforce some order between metadata updates[10] and therefore might enforce order also between different applications (if they update related metadata). Another possible approach, from HP Labs,[26] does change the file-system interface but keeps the new interface simple, while being supported on a production-ready file system.

A third avenue for improving the crash consistency of applications goes beyond testing and seeks a way of formally modeling file systems. Our study introduces a method of modeling file systems that completely expresses their crash behavior via abstract persistence models. We modeled five file-system configurations and used the models to discover application vulnerabilities exposed in each of the modeled file systems. Researchers from MIT[5] have more broadly considered different formal approaches for modeling a file system and found Hoare logic to be the best.

Beyond local file systems, application crash consistency is an interesting problem in proposed storage stacks that will be constructed on the fly, mixing and matching different layers such as block remappers, logical volume managers, and file systems.[27,28] An expressive language is required for specifying the complex storage guarantees and requirements of the different layers in such storage stacks. Our group is also working on such a language, along with methods to prove the overall correctness of the entire storage stack.[1]

Conclusion

This article aims to convince readers that application-level crash consistency is a real and important problem. Similar problems have been faced before in other areas of computer systems, in the domains of multiprocessor shared memory and distributed systems. Those problems have been overcome by creating new abstractions, understanding various trade-offs, and even thinking about the problem with analogies to baseball.[25] Similar solutions are possible for application crash consistency, too, but only with the involvement of the wider systems community.

Related articles on queue.acm.org

Abstraction in Hardware System Design
Rishiyur S. Nikhil
http://queue.acm.org/detail.cfm?id=2020861

Storage Systems: Not Just a Bunch of Disks Anymore
Erik Riedel
http://queue.acm.org/detail.cfm?id=864059

Keeping Bits Safe: How Hard Can It Be?
David S. H. Rosenthal
http://queue.acm.org/detail.cfm?id=1866298

References

1. Alagappan, R., Chidambaram, V., Sankaranarayana Pillai, T., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H. Beyond storage APIs: Provable semantics for storage stacks. In *Proceedings of the 15th Workshop on Hot Topics in Operating Systems* (Kartause Ittingen, Switzerland, May 2015).
2. Al-Kiswany, S. Namenode fails to boot if the file system reorders `rename` operations, 2014; http://issues.apache.org/jira/browse/HDFS-6820.
3. Aurora, V. POSIX v. reality: A position on O PONIES, 2009; http://lwn.net/Articles/351422/.
4. Austin Group Defect Tracker. 0000672: Necessary step(s) to synchronize filename operations on disk, 2013; http://austingroupbugs.net/view.php?id=672.
5. Chen, H., Ziegler, D., Chlipala, A., Kaashoek, M. F., Kohler, E., Zeldovich, N. Specifying crash safety for storage systems. In *Proceedings of the 15th Workshop on Hot Topics in Operating Systems* (Kartause Ittingen, Switzerland, May 2015).
6. Corbet, J. Ext4 and data loss, 2009; https://lwn.net/Articles/322823/.
7. Corbet, J. That massive filesystem thread, 2009; http://lwn.net/Articles/326471/.
8. Davies, C. Fake hard drive has short-term memory not 500GB. SlashGear, 2011; http://www.slashgear.com/fake-hard-drive-has-short-term-memory-not-500gb-08145144/.
9. Edge, J. Testing power failures, 2015; https://lwn.net/Articles/637079/.
10. Ganger, G.R., Patt, Y.N. 1994. Metadata update performance in file systems. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation.* (Monterey, CA, Nov. 1994), 49–60.
11. Garcia-Molina, H., Ullman, J.D., Widom, J. *Database Systems: The Complete Book.* Prentice Hall Press, 2008.
12. Hagmann, R. Reimplementing the Cedar file system using logging and group commit. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, (Austin, TX, Nov. 1987).
13. Kim, H., Agrawal, N., Ungureanu, C. Revisiting storage for smartphones. In *Proceedings of the 10th Usenix Symposium on File and Storage Technologies* (San Jose, CA, Feb. 2012).
14. Lamport, L. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers 28*, 9 (1979), 690–691.
15. Mercurial. Dealing with repository and dirstate corruption, 2014; http://mercurial.selenic.com/wiki/RepositoryCorruption.
16. Microsoft. Alternatives to using transactional NTFS; https://msdn.microsoft.com/en-us/library/windows/desktop/hh802690(v=vs.85).aspx.
17. Open Group Base Specifications. POSIX.1-2008 IEEE Std 1003.1, 2013; http://pubs.opengroup.org/onlinepubs/9699919799/.
18. Sankaranarayana Pillai, T. Possible bug: `fsync()` required after calling `rename()`, 2013; https://code.google.com/p/leveldb/issues/detail?id=189.
19. Sankaranarayana Pillai, T. Possible bug: Missing a `fsync()` on the log file before compaction, 2013; https://code.google.com/p/leveldb/issues/detail?id=187.
20. Sankaranarayana Pillai, T., Chidambaram, V. Alagappan, R., Al-Kiswany, S., Arpaci-Dusseau, A.C. and Arpaci-Dusseau, R.H. ALICE: Application-Level Intelligent Crash Explorer; http://research.cs.wisc.edu/adsl/Software/alice/.
21. Sankaranarayana Pillai, T., Chidambaram, V., Alagappan, R., Al-Kiswany, S., Arpaci-Dusseau, A.C. and Arpaci-Dusseau, R.H. 2014. All file systems are not created equal: on the complexity of crafting crash-consistent applications. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation* (Broomfield, CO, Oct. 2014).
22. Sewell, P., Sarkar, S., Owens, S., Nardelli, F.Z. and Myreen, M.O. x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM 53*, 7 (July 2010): 89–97.
23. Shvachko, K., Kuang, H., Radia, S. and Chansler, R. The Hadoop Distributed File System. In *Proceedings of the 26th IEEE Symposium on Mass Storage Systems and Technologies* (Incline Village, NV, May 2010).
24. Sorin, D.J., Hill, M.D., Wood, D.A. A Primer on Memory Consistency and Cache Coherence. Morgan & Claypool Publishers, 2011.
25. Terry, D. Replicated data consistency explained through baseball. *MSR Technical Report* (Oct. 2011).
26. Verma, R., Mendez, A.A., Park, S., Mannarswamy, S.S., Kelly, T.P., and Morrey III, C.B. Failure-atomic updates of application data in a Linux file system. In *Proceedings of the 13th Usenix Symposium on File and Storage Technologies* (Santa Clara, CA, Feb. 2015).
27. VMWare. Software-defined storage (SDS) and storage virtualization; http://www.vmware.com/software-defined-datacenter/storage.
28. VMWare. The VMware perspective on software-defined storage; http://www.vmware.com/files/pdf/solutions/VMware-Perspective-on-software-defined-storage-white-paper.pdf.
29. Zheng, M., Tucek, J., Huang, D., Qin, F., Lillibridge, M., Yang, E. S., Zhao, B. W., Singh, S. Torturing databases for fun and profit. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation* (Broomfield, CA, Oct. 2014).

T. Sankaranarayana Pillai, **Vijay Chidambaram**, and **Ramnatthan Alagappan** (madthanu, vijayc, ra @cs.wisc.edu) are Ph.D. candidates in the Department of Computer Science at the University of Wisconsin–Madison. Chidambaram is joining the faculty at the University of Texas at Austin.

Samer Al-Kiswany (samera@cs.wisc.edu) is a postdoctoral fellow in the Department of Computer Science at the University of Wisconsin–Madison.

Andrea Arpaci-Dusseau and **Remzi Arpaci-Dusseau** (dusseau, remzi @cs.wisc.edu) are professors of computer science at the University of Wisconsin–Madison.