*-Box: Towards Reliability and Consistency in Dropbox-like File Synchronization Services

Yupu Zhang, Charlotte Dragga, Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau
Computer Sciences Department, University of Wisconsin-Madison

Abstract

Cloud-based file synchronization services, such as Dropbox, have never been more popular. They provide excellent reliability and durability in their server-side storage, and can provide a consistent view of their synchronized files across multiple clients. However, the loose coupling of these services and the local file system may, in some cases, turn these benefits into drawbacks. In this paper, we show that these services can silently propagate both local data corruption and the results of inconsistent crash recovery, and cannot guarantee that the data they store reflects the actual state of the disk. We propose techniques to prevent and recover from these problems by reducing the separation between local file systems and synchronization clients, providing clients with deeper knowledge of file system activity and allowing the file system to take advantage of the correct data stored remotely.

1 Introduction

Cloud-based file synchronization services have exploded in popularity in recent years. Dropbox is foremost among them, having surpassed 100 million users as of November 2012 [5], and services like Google Drive and Microsoft SkyDrive are also popular.

File synchronization services occupy a unique design point between distributed file systems, like NFS or Coda, and file backup services, like Mozy or Data Domain. Like the former, file synchronization services provide a means for users to access their files on any machine connected to the service. Like the latter, however, file synchronization services propagate local changes asynchronously, and often provide a means to restore previous versions of files. Furthermore, they are only loosely integrated with the file system, allowing them to be portable across a wide range of devices.

While the automatic propagation of files as they are modified is no doubt key to these services' success, the perceived reliability and consistency they provide is also instrumental to their appeal. The Dropbox tour goes as far as to state that "none of your stuff will ever be lost" [1]. Unfortunately, the loose coupling of cloud synchronization services with the underlying file system gives the lie to this claim. While the data stored remotely is generally robust, local client software is unable to distinguish between deliberate modifications and unintentional errors, potentially causing corruption to automatically propagate to all machines associated with a user. Thus, despite the presence of multiple redundant copies, synchronization destroys the user's data.

In this paper, we outline the underpinnings for *-*Box*[1] (pronounced "star-box"), our project to analyze this phenomenon in detail and correct it. We first examine how these services can silently propagate data corruption and then how they cannot guarantee data consistency with the underlying file system after a crash. For both issues, we investigate how the separation of the client from the file system causes these problems, and then propose ways to better integrate the two components to allow both prevention and recovery. Due to its dominant market share, we focus primarily on Dropbox, but our proposals are equally applicable to similar services. We conclude with a discussion of the current status of the *-Box project and our future work on it.

2 Background

A cloud-based file synchronization service generally consists of two parts: server software that stores user data and a client-side daemon that actively synchronizes one or more directories with the server. The server software is usually managed by the service provider, such as Dropbox or Google, though it may run on a public cloud, like Amazon's EC2. While there are a wide variety of design and implementation choices on the server side, the operation of the server is opaque to the user, and service providers almost never publish technical details. Thus, we focus our attention on the client-side daemon.

Though clients are usually closed-source, we can observe their behavior on the user's computer and infer many of their design decisions. As an example, we now use Dropbox to examine the two primary operations of a synchronization service: detection and transmission.

Detection: The client must be able to automatically detect changes, both remotely at the server and in the local file system. Dropbox uses push-based notifications to detect remote changes [8], while relying on a file-system monitoring service, such as Linux's inotify, to detect local changes. To detect changes that occur while it is offline, Dropbox keeps detailed metadata for each file, in-

[1] "*" is a wild-card character that can be replaced with many desired properties such as reliable, consistent, etc.

cluding last modification and attribute change times, in a local database. When booting, Dropbox scans its monitored files, uploading any whose metadata (other than access time) has changed in any way from their last known value. Thus, Dropbox will upload any file with a different modification time than the one it has recorded, even if the new time precedes the recorded one.

Transmission: When sending changed files to and from the server, Dropbox uses deduplication to reduce network traffic, calculating a SHA-1 hash for each 4 MB chunk in the file. It then sends the hash for that chunk to the destination first; if the chunk is present, it is not sent. For partially modified chunks, Dropbox uses rsync to transmit only the changed portions. Finally, to achieve atomicity at the client, Dropbox downloads chunks of files to a staging area, assembles them, and then renames them to their final location.

3 Data Reliability

One might think that a local file system synchronized with Dropbox will provide better data reliability, due to the redundant copies of data on the cloud and the excellent protection offered by the service provider. However, focusing on the back-end risks neglecting the other part of the system: the local file system, which is especially crucial in Dropbox's case.

Because the Dropbox client has only minimal knowledge of file system activity, it may propagate undetected data corruption to the server, which will then pollute the copies on all machines connected to the account. While Dropbox allows the user to revert to a clean copy, it only keeps revisions for thirty days ordinarily; thus, if the user fails to notice the corruption, the data could be permanently lost. Even if the file system detects corruption, the user still has to manually revert the file to the clean copy in Dropbox, a process that should ideally be automated.

In this section, we first introduce the problem of corruption propagation in Dropbox-like file synchronization services. Then we discuss approaches to prevent such propagation and recover from the corruption.

3.1 The Corruption Problem

Data corruption is not uncommon and can occur due to defects on disk media, bugs in drive firmware and disk controllers, and even software bugs in operating systems [3, 12]. Without a file synchronization service, the corruption remains local. However, a synchronization service running atop a local file system may propagate corruption to every copy synchronized with the service.

To determine how a disk corruption could be propagated to the cloud, we perform a series of fault injection experiments with a variety of file synchronization services, as listed in Table 1. We first inject disk corruption to a block in a file that is synchronized with the cloud. Then

FS	Service	Data write	Metadata mtime	Metadata ctime	Metadata atime
ext4 (Linux)	Dropbox	×	×	×	
	ownCloud	×	×		
	FileRock	×	×		
HFS+ (Mac OS X)	Dropbox	×	×		
	ownCloud	×	×		
	GoogleDrive	×	×		
	SugarSync	×			
	Syncplicity	×	×		

Table 1: **Corruption Propagation.** *"×": corrupt data uploaded to server. Blank: no changes detected, so corruption is not uploaded.*

we perform several test workloads and see if the corrupted block is propagated to the cloud.

Our test workloads fall into two categories: metadata-only operations, which do not alter file data, and data operations, which change the uncorrupted portion of the file. For metadata, we focus on the timestamps stored in the inode: access time (*touch -a*), modification time (*touch -m*) and change time (*chown* and *chmod*). For data operations, we execute regular appends and in-place updates both close to and far from the corrupted block. These data operations always update the modification time of the file.

We perform the experiments on ext4 [9] in Linux (kernel 3.6.11) and HFS+ [2] in Mac OS X (10.5 Lion). The results are shown in Table 1. We can see that the corruption gets propagated to the cloud whenever the client detects a change to the file. Clients also propagate corruption when the modification time is changed without writing the file; they detect the changes in the file (due to the injected corruption) and upload the corrupted version. SugarSync is an exception, waiting until the file's contents change or until it restarts to upload the file.

3.2 Solution Space

We believe that the key to solving this problem is to distinguish legitimate changes (actual updates) from "unauthorized" changes (corruption). We investigate two different solutions: data checksumming and fine-grained file-system monitoring.

3.2.1 Data Checksumming

Usually, detecting corruption requires checksums from the file system, and recovering from corruption requires redundant copies. Only a few file systems, such as ZFS [4] and btrfs [11], provide such detection and recovery mechanisms, while many other popular file systems, such as ext4, omit them. Therefore, we can either use ZFS or btrfs directly, or enhance file systems such as ext4 with data block checksumming. We should note that ZFS, as well as btrfs, can only detect corruption on disk, not in memory. More aggressive checksumming techniques, such as

page-cache level checksums, are required to detect memory corruption [13, 14].

Recovering from detected corruption usually requires additional space or extra disks to store redundant copies. This is often not desired or affordable by regular users. However, in this case, the Dropbox server already provides these copies; we simply need to enable the file system to take advantage of them.

3.2.2 Fine-grained File-System Monitoring

If checksumming is not an option, we can still prevent the propagation of corrupted data by implementing a more fine-grained file-system monitoring mechanism and adapting the client to use it. For example, Linux's inotify framework only provides file-level monitoring and cannot tell which part of a file has been modified. When a Dropbox client detects a change, it uses the rsync algorithm to determine which parts of the file were modified, uploading any deltas to the cloud. Rsync usually reads the whole file from disk and is unable to distinguish corruption from legitimate changes. Therefore, adding the range of the update to each modify event will allow the client to upload only the data in this range, thus preventing the corruption on disk from being uploaded to the server.

The client can then detect corruption by checksumming the file and verifying it against the remote copy. This method will not, however, detect corruption that was read by the application and written out without being modified, as might occur with an insertion in a document or when a file is written to a temporary location and then renamed over the original file.

4 Crash Consistency

While cloud back-end storage is often only eventually consistent, as in Dynamo [6] and similar systems, a substantial body of work exists on masking that inconsistency from the user. These techniques provide a solid foundation for securing user data in the cloud but fail to address inconsistencies that may arise at the client, especially those resulting from crash recovery. While Dropbox could provide substantial assistance in recovering from crashes, it often does not. At best, it preserves the status quo; at worst, it actively propagates inconsistent data resulting from partial recovery. Dropbox does so because it lacks a full view of the events that occur in its monitored directories. This leads to a broader, application-level inconsistency problem, in which Dropbox is unable to guarantee that its remote contents reflect the complete state of the local disk at any point in time.

In this section, we first discuss the twin problems of crash consistency and application-level inconsistency present in Dropbox and similar synchronization services. We then analyze the solution space and the trade-offs in performance and freshness that it contains.

4.1 The Consistency Problem

The consistency problems in Dropbox and similar services primarily manifest themselves when crashes occur. The first problem, that of crash recovery, arises from Dropbox's interactions with the underlying file system recovery mechanisms. The second, an inconsistent application-level image of the underlying file system, will affect a wider variety of file systems and must be addressed in order for Dropbox to provide complete recovery in the event of catastrophic failure, like that of a disk. As the crash recovery problem is the more specific of the two, we analyze it first; from there, we examine broader application-level inconsistency.

4.1.1 Crash Recovery

Recovering gracefully from crashes has been a key goal of file system developers since the introduction of fsck, if not earlier [10]. Today, two primary mechanisms exist for achieving this: copy-on-write, used in btrfs and ZFS, and journaling, used in Linux's ext4 and Mac OS X's HFS+, among others. As their name implies, copy-on-write (COW) file systems never overwrite data when issuing an update, ensuring they can roll back to a consistent version of the file system after a crash. In contrast, journaling file systems issue their writes to a log (or journal) before committing them to a fixed location, allowing for replay or rollback, as appropriate. If all data and metadata are written to the journal, journaling will provide nearly the same consistency as COW. However, because doing so requires all blocks to be written twice, it is more common to only journal metadata and simply ensure that the data blocks are written first, as in ext4's ordered mode. This method provides complete metadata consistency but may result in data reflecting only part of an update.

To explore how Dropbox deals with crash recovery in each type of file system, we initialize Dropbox with a single file present on disk and in the cloud at version v_0. We then write a new version, v_1, and inject a crash. As the system recovers from the crash, we observe Dropbox's behavior. We perform this experiment on ZFS and on ext4 using ordered mode (without delayed allocation).

In ZFS, Dropbox always synchronizes the cloud to the state on the disk, even if v_0 is on disk and v_1 is in the cloud. This case occurs if Dropbox synchronizes v_1 before ZFS commits it to disk. When ZFS reboots, it still has the original version, v_0, which contains the old modification time. This local timestamp differs from the one recorded in Dropbox's database, so Dropbox treats it as a local update and uploads the file, correctly propagating the file system state.

Dropbox's interactions with ext4's ordered mode journaling are less predictable, depending on the precise state of the file system and Dropbox's databases at the time of

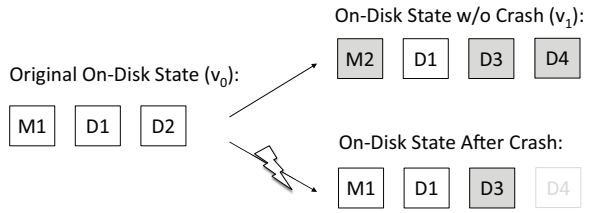

Figure 1: **A Crash in Ext4 Ordered Mode.** *This image shows a file whose data is rendered inconsistent by a crash in ext4 ordered mode. The original state of the file on disk (v_0) is shown at left, with its inode block, M1, pointing to two data blocks, D1 and D2. An application then overwrites the last block (D2) with D3 and appends a new block, D4, resulting in an updated inode block, M2. If there is no crash, the final state of the file will be v_1, as shown in the upper right. All newly written blocks are marked with gray. If a crash occurs before M2 is journaled but after D3 and D4 are written, the file will be in an inconsistent state after recovery, as shown in the lower right.*

the crash. Again, Dropbox updates the cloud to the state on disk; this works well for cases where both the data and metadata for v_1 are either completely written (committed) or completely unwritten before the crash. However, if the crash occurs before v_1's transaction commits but after some of v_1's data is written to disk, the local file will have inconsistent data. As shown in Figure 1, after reboot, the metadata of the file is consistent on disk at v_0 but the file's data is an inconsistent mixture of v_0 and v_1. Such data inconsistency cannot be detected and corrected by fsck. In this case, if Dropbox has already updated its database with v_1's modification time, which differs from the local timestamp (v_0), it will immediately propagate the inconsistent file.

4.1.2 Application-level Inconsistency

Because it only communicates with the file system via a notification system, such as Linux's inotify, Dropbox has a weak sense of what changes are occurring in the file system. In particular, it cannot determine when data and metadata become persistent on disk, leading to its problems during crash recovery. In addition, because Dropbox uploads asynchronously, generally at a much slower rate than the local disk [8], there is no guarantee that the current version of the files stored in Dropbox reflect a consistent application-level image of the local file system.

To illustrate how this inconsistency can cause problems, consider synchronizing a photo-editing suite that keeps photos and their thumbnail previews in separate files. As Dropbox uploads small files first, the thumbnail is likely to reach its servers before the photo. In the event of a crash after editing a photo and updating its thumbnail, Dropbox may have the most recent version of the thumbnail but only an old copy of the photo. The resulting mismatch may lead to user confusion or, depending on the nature of the edits, potential embarrassment when looking through photos at a later date.

This problem is exacerbated by Dropbox's inability to upload a file that is being actively modified, as doing so may combine incomplete portions of several writes. The resultant delay increases the likelihood that frequently modified files will experience data loss. Conversely, this strategy cannot detect when updates between files depend on each other, and, as in the previous example, may result in an uploaded state that would never occur on disk.

4.2 Solution Space

Recovering gracefully from a crash and preventing application-level inconsistency requires Dropbox to have a strong sense of what data is persistent on disk at specific points in time. While delaying uploads of actively modified files provides some of this knowledge during normal operation, obtaining it in the event of failure is more challenging. In this section, we propose a variety of approaches for obtaining application-consistent disk images in Dropbox and then using these images to recover from crashes. As ZFS handles basic recovery correctly, we target journaling file systems; however, our proposals for application-level consistency are also applicable to COW file systems.

4.2.1 Establishing Application-level Consistency

One basic approach is to ensure that files are uploaded in the order in which their transactions commit to disk. This requires the underlying file system to notify Dropbox when it commits a transaction, informing Dropbox of the files involved. While this will work well for workloads that modify only a few files at a time, involving minimal amounts of bookkeeping, it may encounter difficulties in workloads featuring complex access patterns to a variety of files. Because Dropbox cannot upload files atomically, it may need to delay uploading a set of files until all I/O to them has quiesced completely.

An alternative to this approach is to employ snapshots, gathered either implicitly or explicitly. Implicit snapshots use a similar technique to the previous consistency method; rather than imposing a specific ordering, however, they simply link a file's newest revision in Dropbox to the last transaction in which the file was modified. Explicit snapshots, on the other hand, upload in-memory copies of the write data associated with each transaction. Implicit snapshots will likely require fewer changes to both the operating system and the client than explicit, but may lead to staler remote copies. Conversely, explicit snapshots are likely to impose higher overheads.

Finally, if only certain data need to be consistent, we could employ an `fsync`-like primitive that blocks an application until the data reaches the cloud. This strictly enforces consistency, but it also constrains applications to Dropbox's available bandwidth, which will generally be lower than that of the disk [8].

4.2.2 Recovering From Crashes

Given knowledge of consistent disk states, recovery becomes relatively simple. A straightforward approach could simply choose the most recent point known to be consistent, and restore all files to that point. However, this requires verifying the contents of all the files that Dropbox monitors to ensure that each has consistent data.

Ideally, we should restore only those files affected by the crash; most journaling file systems, however, do not provide this information. One way to determine this is to record in the journal the names of all files with pending modifications in the current committing transaction before writing any of their data blocks. This is similar to the declared-mode journaling proposed by Denehy et al. [7] and will likely exhibit similar performance.

5 Status and Future Work

Currently, *-Box, our project to remedy consistency and reliability problems in cloud-based synchronization services, focuses on two file systems: ZFS and ext4. While we have diagnosed both sets of problems, we have thus far only implemented solutions for reliability.

Since ZFS already provides data checksumming, we enhance ZFS with the ability to recover detected corruption using Dropbox. We modify ZFS's recovery policy, so that, when ZFS fails to recover from local redundant copies, it sends a recovery request to a user-level daemon. The daemon then fetches the requested block from the Dropbox server through the REST API and returns it to ZFS. ZFS re-verifies the integrity of the block before it uses the data to perform recovery.

For ext4, we avoid adding checksums and instead add update ranges to inotify modification events. We then modify an open-source file synchronization service, ownCloud, to upload only the ranges specified in these events to the server, via the HTTP PATCH command. Once the file is uploaded, the client compares its local checksum of the file with the checksum from the server; if these differ, the client assumes its version is corrupt and downloads the server's version to a safe location.

We next plan to implement solutions for consistency and crash recovery using implicit and explicit snapshots. Because ZFS recovers from crashes automatically, we will focus our efforts on ext4, using the declared-mode journaling described earlier to determine which files need to be recovered. As there seem to be substantial trade-offs between implicit and explicit snapshots, we will pay close attention to their performance overheads under a variety of workloads.

Ultimately, though, all of these approaches are somewhat stop-gap in nature, attempting to maximize benefit for a minimum of infrastructure changes. Removing the separation between the cloud synchronization service and the underlying file system is a powerful concept, however, and a system that fully integrates the two could realize even greater improvements. For instance, our proposed recovery system could allow for substantially relaxed journaling requirements; we can also combine metadata between the synchronization service and file system, improving time and space overheads during basic operations. Eventually, we seek to create such an integrated design, building a cohesive system that provides capabilities beyond those that synchronization services and file systems can provide in isolation.

6 Acknowledgments

We thank the reviewers and our shepherd Kaushik Veeraraghavan for their feedback and comments. This material is based upon work supported by the National Science Foundation (NSF) under CCF-0811657 and CNS-0834392 as well as generous donations from EMC, Facebook, Fusion-io, Google, Huawei, Microsoft, NetApp, Samsung, and VMware. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or other institutions.

References

[1] What is dropbox? https://www.dropbox.com/tour/1.

[2] Apple. Technical Note TN1150. http://dubeiko.com/development/FileSystems/HFSPLUS/tn1150.html, March 2004.

[3] L. N. Bairavasundaram, G. R. Goodson, B. Schroeder, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. An Analysis of Data Corruption in the Storage Stack. In *FAST '08*, San Jose, CA, February 2008.

[4] J. Bonwick and B. Moore. ZFS: The Last Word in File Systems. http://opensolaris.org/os/community/zfs/docs/zfs_last.pdf.

[5] J. Constine. Dropbox is now the data fabric tying together devices for 100m registered users who save 1b files a day. http://techcrunch.com/2012/11/13/dropbox-100-million/, November 2012.

[6] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-Value Store. In *SOSP '07*, Stevenson, WA, October 2007.

[7] T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Journal-guided Resynchronization for Software RAID. In *FAST '05*, pages 87–100, San Francisco, CA, December 2005.

[8] I. Drago, M. Mellia, M. M. Munafò, A. Sperotto, R. Sadre, and A. Pras. Inside dropbox: understanding personal cloud storage services. In *IMC'12*, pages 481–494, Boston, MA, 2012.

[9] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier. The new ext4 filesystem: Current status and future plans. *Linux Symposium*, 2007.

[10] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. Fsck - The UNIX File System Check Program. Unix System Manager's Manual - 4.3 BSD Virtual VAX-11 Version, April 1986.

[11] Oracle Corporation. Btrfs: A Checksumming Copy on Write Filesystem. http://oss.oracle.com/projects/btrfs/.

[12] 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.

[13] Y. Zhang, D. Myers, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Zettabyte Reliability with Flexible End-to-end Data Integrity. In *MSST'13*, Long Beach, CA, May 2013.

[14] Y. Zhang, A. Rajimwale, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. End-to-end Data Integrity for File Systems: A ZFS Case Study. In *FAST'10*, San Jose, CA, February 2010.