ViewBox: Integrating Local File Systems with Cloud Storage Services

Yupu Zhang[†], Chris Dragga[†*], Andrea C. Arpaci-Dusseau[†], Remzi H. Arpaci-Dusseau[†]

[†] *University of Wisconsin-Madison,* [*] *NetApp, Inc.*

Abstract

Cloud-based file synchronization services have become enormously popular in recent years, both for their ability to synchronize files across multiple clients and for the automatic cloud backups they provide. However, despite the excellent reliability that the cloud back-end provides, the loose coupling of these services and the local file system makes synchronized data more vulnerable than users might believe. Local corruption may be propagated to the cloud, polluting all copies on other devices, and a crash or untimely shutdown may lead to inconsistency between a local file and its cloud copy. Even without these failures, these services cannot provide causal consistency.

To address these problems, we present ViewBox, an integrated synchronization service and local file system that provides freedom from data corruption and inconsistency. ViewBox detects these problems using ext4-cksum, a modified version of ext4, and recovers from them using a user-level daemon, cloud helper, to fetch correct data from the cloud. To provide a stable basis for recovery, ViewBox employs the view manager on top of ext4-cksum. The view manager creates and exposes views, consistent in-memory snapshots of the file system, which the synchronization client then uploads. Our experiments show that ViewBox detects and recovers from both corruption and inconsistency, while incurring minimal overhead.

1 Introduction

Cloud-based file synchronization services, such as Dropbox [11], SkyDrive [28], and Google Drive [13], provide a convenient means both to synchronize data across a user's devices and to back up data in the cloud. While automatic synchronization of files is a key feature of these services, the reliable cloud storage they offer is fundamental to their success. Generally, the cloud backend will checksum and replicate its data to provide integrity [3] and will retain old versions of files to offer recovery from mistakes or inadvertent deletion [11]. The robustness of these data protection features, along with the inherent replication that synchronization provides, can provide the user with a strong sense of data safety.

Unfortunately, this is merely a sense, not a reality; the loose coupling of these services and the local file system endangers data even as these services strive to protect it. Because the client has no means of determining whether file changes are intentional or the result of corruption, it may send both to the cloud, ultimately spreading corrupt data to all of a user's devices. Crashes compound this problem; the client may upload inconsistent data to the cloud, download potentially inconsistent files from the cloud, or fail to synchronize changed files. Finally, even in the absence of failure, the client cannot normally preserve causal dependencies between files, since it lacks stable point-in-time images of files as it uploads them. This can lead to an inconsistent cloud image, which may in turn lead to unexpected application behavior.

In this paper, we present ViewBox, a system that integrates the local file system with cloud-based synchronization services to solve the problems above. Instead of synchronizing individual files, ViewBox synchronizes views, in-memory snapshots of the local synchronized folder that provide data integrity, crash consistency, and causal consistency. Because the synchronization client only uploads views in their entirety, ViewBox guarantees the correctness and consistency of the cloud image, which it then uses to correctly recover from local failures. Furthermore, by making the server aware of views, ViewBox can synchronize views across clients and properly handle conflicts without losing data.

ViewBox contains three primary components. Ext4-cksum, a variant of ext4 that detects corrupt and inconsistent data through data checksumming, provides ViewBox's foundation. Atop ext4-cksum, we place the view manager, a file-system extension that creates and exposes views to the synchronization client. The view manager provides consistency through *cloud journaling* by creating views at file-system epochs and uploading views to the cloud. To reduce the overhead of maintaining views, the view manager employs *incremental snapshotting* by keeping only deltas (changed data) in memory since the last view. Finally, ViewBox handles recovery of damaged data through a user-space daemon, cloud helper, that interacts with the server-backend independently of the client.

We build ViewBox with two file synchronization services: Dropbox, a highly popular synchronization service, and Seafile, an open source synchronization service based on GIT. Through reliability experiments, we demonstrate that ViewBox detects and recovers from local data corruption, thus preventing the corruption's propagation. We also show that upon a crash, ViewBox successfully rolls back the local file system state to a previously uploaded view, restoring it to a causally consistent image. By comparing ViewBox to Dropbox or Seafile running atop ext4, we find that ViewBox incurs less than 5% overhead across a set of workloads. In some cases, ViewBox even improves the synchronization time by 30%.

The rest of the paper is organized as follows. We first show in Section 2 that the aforementioned problems exist through experiments and identify the root causes of those problems in the synchronization service and the local file system. Then, we present the overall architecture of ViewBox in Section 3, describe the techniques used in our prototype system in Section 4, and evaluate ViewBox in Section 5. Finally, we discuss related work in Section 6 and conclude in Section 7.

2 Motivation

As discussed previously, the loosely-coupled design of cloud-based file synchronization services and file systems creates an insurmountable semantic gap that not only limits the capabilities of both systems, but leads to incorrect behavior in certain circumstances. In this section, we demonstrate the consequences of this gap, first exploring several case studies wherein synchronization services propagate file system errors and spread inconsistency. We then analyze how the limitations of file synchronization services and file systems directly cause these problems.

2.1 Synchronization Failures

We now present three case studies to show different failures caused by the semantic gap between local file systems and synchronization services. The first two of these failures, the propagation of corruption and inconsistency, result from the client's inability to distinguish between legitimate changes and failures of the file system. While these problems can be warded off by using more advanced file systems, the third, causal inconsistency, is a fundamental result of current file-system semantics.

2.1.1 Data Corruption

Data corruption is not uncommon and can result from a variety of causes, ranging from disk faults to operating system bugs [5, 8, 12, 22]. Corruption can be disastrous, and one might hope that the automatic backups that synchronization services provide would offer some protection from it. These backups, however, make them likely to propagate this corruption; as clients cannot detect corruption, they simply spread it to all of a user's copies, potentially leading to irrevocable data loss.

To investigate what might cause disk corruption to propagate to the cloud, we first inject a disk corruption to a block in a file synchronized with the cloud (by flipping bits through the device file of the underlying disk). We then manipulate the file in several different ways, and observe which modifications cause the corruption to be uploaded. We repeat this experiment for Dropbox, ownCloud, and Seafile atop ext4 (both ordered and data journaling modes) and ZFS [2] in Linux (kernel 3.6.11) and Dropbox, ownCloud, Google Drive, SugarSync, and Syncplicity atop HFS+ in Mac OS X (10.5 Lion).

		Data	Metadata		
FS	**Service**	**write**	**mtime**	**ctime**	**atime**
ext4 (Linux)	Dropbox	*LG*	*LG*	*LG*	*L*
	ownCloud	*LG*	*LG*	*L*	*L*
	Seafile	*LG*	*LG*	*LG*	*LG*
ZFS (Linux)	Dropbox	*L*	*L*	*L*	*L*
	ownCloud	*L*	*L*	*L*	*L*
	Seafile	*L*	*L*	*L*	*L*
HFS+ (Mac OS X)	Dropbox	*LG*	*LG*	*L*	*L*
	ownCloud	*LG*	*LG*	*L*	*L*
	GoogleDrive	*LG*	*LG*	*L*	*L*
	SugarSync	*LG*	*L*	*L*	*L*
	Syncplicity	*LG*	*LG*	*L*	*L*

Table 1: **Data Corruption Results.** *"L": corruption remains local. "G": corruption is propagated (global).*

We execute both data operations and metadata-only operations on the corrupt file. Data operations consist of both appends and in-place updates at varying distances from the corrupt block, updating both the modification and access times; these operations never overwrite the corruption. Metadata operations change only the timestamps of the file. We use *touch -a* to set the access time, *touch -m* to set the modification time, and *chown* and *chmod* to set the attribute-change time.

Table 1 displays our results for each combination of file system and service. Since ZFS is able to detect local corruption, none of the synchronization clients propagate corruption. However, on ext4 and HFS+, all clients propagate corruption to the cloud whenever they detect a change to file data and most do so when the modification time is changed, even if the file is otherwise unmodified. In both cases, clients interpret the corrupted block as a legitimate change and upload it. Seafile uploads the corruption whenever any of the timestamps change. SugarSync is the only service that does not propagate corruption when the modification time changes, doing so only once it explicitly observes a write to the file or it restarts.

2.1.2 Crash Inconsistency

The inability of synchronization services to identify legitimate changes also leads them to propagate inconsistent data after crash recovery. To demonstrate this behavior, we initialize a synchronized file on disk and in the cloud at version v_0. We then write a new version, v_1, and inject a crash which may result in an inconsistent version v_1' on disk, with mixed data from v_0 and v_1, but the metadata remains v_0. We observe the client's behavior as the system recovers. We perform this experiment with Dropbox, ownCloud, and Seafile on ZFS and ext4.

Table 2 shows our results. Running the synchroniza-

FS	Service	Upload local ver.	Download cloud ver.	OOS
ext4 (ordered)	Dropbox	√	×	√
	ownCloud	√	√	√
	Seafile	N/A	N/A	N/A
ext4 (data)	Dropbox	√	×	×
	ownCloud	√	√	×
	Seafile	√	×	×
ZFS	Dropbox	√	×	×
	ownCloud	√	√	×
	Seafile	√	×	×

Table 2: **Crash Consistency Results.** *There are three outcomes: uploading the local (possibly inconsistent) version to cloud, downloading the cloud version, and OOS (out-of-sync), in which the local version and the cloud version differ but are not synchronized. "×" means the outcome does not occur and "√" means the outcome occurs. Because in some cases the Seafile client fails to run after the crash, its results are labeled "N/A".*

tion service on top of ext4 with ordered journaling produces erratic and inconsistent behavior for both Dropbox and ownCloud. Dropbox may either upload the local, inconsistent version of the file or simply fail to synchronize it, depending on whether it had noticed and recorded the update in its internal structures before the crash. In addition to these outcomes, ownCloud may also download the version of the file stored in the cloud if it successfully synchronized the file prior to the crash. Seafile arguably exhibits the best behavior. After recovering from the crash, the client refuses to run, as it detects that its internal metadata is corrupted. Manually clearing the client's metadata and resynchronizing the folder allows the client to run again; at this point, it detects a conflict between the local file and the cloud version.

All three services behave correctly on ZFS and ext4 with data journaling. Since the local file system provides strong crash consistency, after crash recovery, the local version of the file is always consistent (either v_0 or v_1). Regardless of the version of the local file, both Dropbox and Seafile always upload the local version to the cloud when it differs from the cloud version. OwnCloud, however, will download the cloud version if the local version is v_0 and the cloud version is v_1. This behavior is correct for crash consistency, but it may violate causal consistency, as we will discuss.

2.1.3 Causal Inconsistency

The previous problems occur primarily because the file system fails to ensure a key property—either data integrity or consistency—and does not expose this failure to the file synchronization client. In contrast, causal inconsistency derives not from a specific failing on the file system's part, but from a direct consequence of traditional file system semantics. Because the client is unable to obtain a unified view of the file system at a single point in time, the client has to upload files as they change in piecemeal fashion, and the order in which it uploads files may not correspond to the order in which they were changed. Thus, file synchronization services can only guarantee eventual consistency: given time, the image stored in the cloud will match the disk image. However, if the client is interrupted—for instance, by a crash, or even a deliberate powerdown—the image stored remotely may not capture the causal ordering between writes in the file system enforced by primitives like POSIX's `sync` and `fsync`, resulting in a state that could not occur during normal operation.

To investigate this problem, we run a simple experiment in which a series of files are written to a synchronization folder in a specified order (enforced by fsync). During multiple runs, we vary the size of each file, as well as the time between file writes, and check if these files are uploaded to the cloud in the correct order. We perform this experiment with Dropbox, ownCloud, and Seafile on ext4 and ZFS, and find that for all setups, there are always cases in which the cloud state does not preserve the causal ordering of file writes.

While causal inconsistency is unlikely to directly cause data loss, it may lead to unexpected application behavior or failure. For instance, suppose the user employs a file synchronization service to store the library of a photo-editing suite that stores photos as both full images and thumbnails, using separate files for each. When the user edits a photo, and thus, the corresponding thumbnail as well, it is entirely possible that the synchronization service will upload the smaller thumbnail file first. If a fatal crash, such as a hard-drive failure, then occurs before the client can finish uploading the photo, the service will still retain the thumbnail in its cloud storage, along with the original version of the photo, and will propagate this thumbnail to the other devices linked to the account. The user, accessing one of these devices and browsing through their thumbnail gallery to determine whether their data was preserved, is likely to see the new thumbnail and assume that the file was safely backed up before the crash. The resultant mismatch will likely lead to confusion when the user fully reopens the file later.

2.2 Where Synchronization Services Fail

Our experiments demonstrate genuine problems with file synchronization services; in many cases, they not only fail to prevent corruption and inconsistency, but actively spread them. To better explain these failures, we present a brief case-study of Dropbox's local client and its interactions with the file system. While Dropbox is merely one service among many, it is well-respected and established, with a broad user-base; thus, any of its flaws are likely to be endemic to synchronization services as a whole and not merely isolated bugs.

Like many synchronization services, Dropbox actively monitors its synchronization folder for changes using a file-system notification service, such as Linux's inotify or Mac OS X's Events API. While these services inform Dropbox of both namespace changes and changes to file content, they provide this information at a fairly coarse granularity—per file, for inotify, and per directory for the Events API, for instance. In the event that these services fail, or that Dropbox itself fails or is closed for a time, Dropbox detects changes in local files by examining their statistics, including size and modification timestamps

Once Dropbox has detected that a file has changed, it reads the file, using a combination of rsync and file chunking to determine which portions of the file have changed and transmits them accordingly [10]. If Dropbox detects that the file has changed while being read, it backs off until the file's state stabilizes, ensuring that it does not upload a partial combination of several separate writes. If it detects that multiple files have changed in close temporal proximity, it uploads the files from smallest to largest.

Throughout the entirety of the scanning and upload process, Dropbox records information about its progress and the current state of its monitored files in a local SQLite database. In the event that Dropbox is interrupted by a crash or deliberate shut-down, it can then use this private metadata to resume where it left off.

Given this behavior, the causes of Dropbox's inability to handle corruption and inconsistency become apparent. As file-system notification services provide no information on what file contents have changed, Dropbox must read files in their entirety and assume that any changes that it detects result from legitimate user action; it has no means of distinguishing unintentional changes, like corruption and inconsistent crash recovery. Inconsistent crash recovery is further complicated by Dropbox's internal metadata tracking. If the system crashes during an upload and restores the file to an inconsistent state, Dropbox will recognize that it needs to resume uploading the file, but it cannot detect that the contents are no longer consistent. Conversely, if Dropbox had finished uploading and updated its internal timestamps, but the crash recovery reverted the file's metadata to an older version, Dropbox must upload the file, since the differing timestamp could potentially indicate a legitimate change.

2.3 Where Local File Systems Fail

Responsibility for preventing corruption and inconsistency hardly rests with synchronization services alone; much of the blame can be placed on local file systems, as well. File systems frequently fail to take the preventative measures necessary to avoid these failures and, in addition, fail to expose adequate interfaces to allow synchronization services to deal with them. As summarized in Table 3, neither a traditional file system, ext4, nor a modern file system, ZFS, is able to avoid all failures.

FS	Corruption	Crash	Causal
ext4 (ordered)	×	×	×
ext4 (data)	×	√	×
ZFS	√	√	×

Table 3: **Summary of File System Capabilities.** *This table shows the synchronization failures each file system is able to handle correctly. There are three types of failures: Corruption (data corruption), Crash (crash inconsistency), and Causal (causal inconsistency). "√" means the failure does not occur and "×" means the failure may occur.*

File systems primarily prevent corruption via checksums. When writing a data or metadata item to disk, the file system stores a checksum over the item as well. Then, when it reads that item back in, it reads the checksum and uses that to validate the item's contents. While this technique correctly detects corruption, file system support for it is limited. ZFS [6] and btrfs [23] are some of the few widely available file systems that employ checksums over the whole file system; ext4 uses checksums, but only over metadata [9]. Even with checksums, however, the file system can only detect corruption, requiring other mechanisms to repair it.

Recovering from crashes without exposing inconsistency to the user is a problem that has dogged file systems since their earliest days and has been addressed with a variety of solutions. The most common of these is journaling, which provides consistency by grouping updates into transactions, which are first written to a log and then later checkpointed to their fixed location. While journaling is quite popular, seeing use in ext3 [26], ext4 [20], XFS [25], HFS+ [4], and NTFS [21], among others, writing all data to the log is often expensive, as doing so doubles all write traffic in the system. Thus, normally, these file systems only log metadata, which can lead to inconsistencies in file data upon recovery, even if the file system carefully orders its data and metadata writes (as in ext4's ordered mode, for instance). These inconsistencies, in turn, cause the erratic behavior observed in Section 2.1.2.

Crash inconsistency can be avoided entirely using copy-on-write, but, as with file-system checksums, this is an infrequently used solution. Copy-on-write never overwrites data or metadata in place; thus, if a crash occurs mid-update, the original state will still exist on disk, providing a consistent point for recovery. Implementing copy-on-write involves substantial complexity, however, and only recent file systems, like ZFS and btrfs, support it for personal use.

Finally, avoiding causal inconsistency requires access to stable views of the file system at specific points in time. File-system snapshots, such as those provided by ZFS or Linux's LVM [1], are currently the only means of obtaining such views. However, snapshot support is relatively uncommon, and when implemented, tends not to be designed for the fine granularity at which synchronization services capture changes.

2.4 Summary

As our observations have shown, the sense of safety provided by synchronization services is largely illusory. The limited interface between clients and the file system, as well as the failure of many file systems to implement key features, can lead to corruption and flawed crash recovery polluting all available copies, and causal inconsistency may cause bizarre or unexpected behavior. Thus, naively assuming that these services will provide complete data protection can lead instead to data loss, especially on some of the most commonly-used file systems.

Even for file systems capable of detecting errors and preventing their propagation, such as ZFS and btrfs, the separation of synchronization services and the file system incurs an opportunity cost. Despite the presence of correct copies of data in the cloud, the file system has no means to employ them to facilitate recovery. Tighter integration between the service and the file system can remedy this, allowing the file system to automatically repair damaged files. However, this makes avoiding causal inconsistency even more important, as naive techniques, such as simply restoring the most recent version of each damaged file, are likely to directly cause it.

3 Design

To remedy the problems outlined in the previous section, we propose ViewBox, an integrated solution in which the local file system and the synchronization service cooperate to detect and recover from these issues. Instead of a clean-slate design, we structure ViewBox around ext4 (ordered journaling mode), Dropbox, and Seafile, in the hope of solving these problems with as few changes to existing systems as possible.

Ext4 provides a stable, open-source, and widely-used solution on which to base our framework. While both btrfs and ZFS already provide some of the functionality we desire, they lack the broad deployment of ext4. Additionally, as it is a journaling file system, ext4 also bears some resemblance to NTFS and HFS+, the Windows and Mac OS X file systems; thus, many of our solutions may be applicable in these domains as well.

Similarly, we employ Dropbox because of its reputation as one of the most popular, as well as one of the most robust and reliable, synchronization services. Unlike ext4, it is entirely closed source, making it impossible to modify directly. Despite this limitation, we are still able to make significant improvements to the consistency and integrity guarantees that both Dropbox and ext4 provide. However, certain functionalities are unattainable without modifying the synchronization service. Therefore, we take advantage of an open source synchronization service, Seafile, to show the capabilities that a fully integrated file system and synchronization service can provide. Although we only implement ViewBox with Dropbox and Seafile, we believe that the techniques we introduce apply more generally to other synchronization services.

In this section, we first outline the fundamental goals driving ViewBox. We then provide a high-level overview of the architecture with which we hope to achieve these goals. Our architecture performs three primary functions: detection, synchronization, and recovery; we discuss each of these in turn.

3.1 Goals

In designing ViewBox, we focus on four primary goals, based on both resolving the problems we have identified and on maintaining the features that make users appreciate file-synchronization services in the first place.

Integrity: Most importantly, ViewBox must be able to detect local corruption and prevent its propagation to the rest of the system. Users frequently depend on the synchronization service to back up and preserve their data; thus, the file system should never pass faulty data along to the cloud.

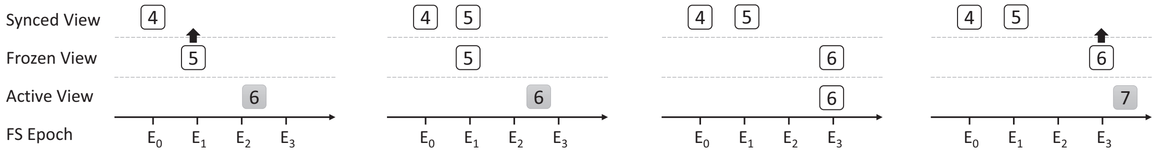
Consistency: When there is a single client, ViewBox should maintain causal consistency between the client's local file system and the cloud and prevent the synchronization service from uploading inconsistent data. Furthermore, if the synchronization service provides the necessary functionality, ViewBox must provide multi-client consistency: file-system states on multiple clients should be synchronized properly with well-defined conflict resolution.

Recoverability: While the previous properties focus on containing faults, containment is most useful if the user can subsequently repair the faults. ViewBox should be able to use the previous versions of the files on the cloud to recover automatically. At the same time, it should maintain causal consistency when necessary, ideally restoring the file system to an image that previously existed.

Performance: Improvements in data protection cannot come at the expense of performance. ViewBox must perform competitively with current solutions even when running on the low-end systems employed by many of the users of file synchronization services. Thus, naive solutions, like synchronous replication [17], are not acceptable.

3.2 Fault Detection

The ability to detect faults is essential to prevent them from propagating and, ultimately, to recover from them as well. In particular, we focus on detecting corruption and data inconsistency. While ext4 provides some ability to detect corruption through its metadata checksums, these



(a) Uploading E_1 as View 5 (b) View 5 is synchronized (c) Freezing E_3 as View 6 (d) Uploading View 6

Figure 1: **Synchronizing Frozen Views.** *This figure shows how view-based synchronization works, focusing on how to upload frozen views to the cloud. The x-axis represents a series of file-system epochs. Squares represent various views in the system, with a view number as ID. A shaded active view means that the view is not at an epoch boundary and cannot be frozen.*

do not protect the data itself. Thus, to correctly detect all corruption, we add checksums to ext4's data as well, storing them separately so that we may detect misplaced writes [6, 18], as well as bit flips. Once it detects corruption, ViewBox then prevents the file from being uploaded until it can employ its recovery mechanisms.

In addition to allowing detection of corruption resulting from bit-flips or bad disk behavior, checksums also allow the file system to detect the inconsistent crash recovery that could result from ext4's journal. Because checksums are updated independently of their corresponding blocks, an inconsistently recovered data block will not match its checksum. As inconsistent recovery is semantically identical to data corruption for our purposes—both comprise unintended changes to the file system—checksums prevent the spread of inconsistent data, as well. However, they only partially address our goal of correctly restoring data, which requires stronger functionality.

3.3 View-based Synchronization

Ensuring that recovery proceeds correctly requires us to eliminate causal inconsistency from the synchronization service. Doing so is not a simple task, however. It requires the client to have an isolated view of all data that has changed since the last synchronization; otherwise, user activity could cause the remote image to span several file system images but reflect none of them.

While file-system snapshots provide consistent, static images [16], they are too heavyweight for our purposes. Because the synchronization service stores all file data remotely, there is no reason to persist a snapshot on disk. Instead, we propose a system of in-memory, ephemeral snapshots, or *views*.

3.3.1 View Basics

Views represent the state of the file system at specific points in time, or epochs, associated with quiescent points in the file system. We distinguish between three types of views: active views, frozen views, and synchronized views. The active view represents the current state of the local file system as the user modifies it. Periodically, the file system takes a snapshot of the active view; this becomes the current frozen view. Once a frozen view is uploaded to the cloud, it then becomes a synchronized view, and can be used for restoration. At any time, there is only one active view and one frozen view in the local system, while there are multiple synchronized views on the cloud.

To provide an example of how views work in practice, Figure 1 depicts the state of a typical ViewBox system. In the initial state, (a), the system has one synchronized view in the cloud, representing the file system state at epoch 0, and is in the process of uploading the current frozen view, which contains the state at epoch 1. While this occurs, the user can make changes to the active view, which is currently in the middle of epoch 2 and epoch 3.

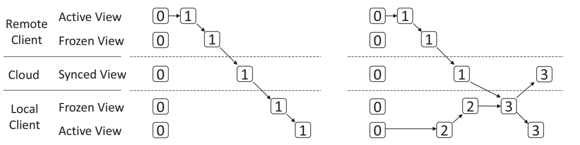
Once ViewBox has completely uploaded the frozen view to the cloud, it becomes a synchronized view, as shown in (b). ViewBox refrains from creating a new frozen view until the active view arrives at an epoch boundary, such as a journal commit, as shown in (c). At this point, it discards the previous frozen view and creates a new one from the active view, at epoch 3. Finally, as seen in (d), ViewBox begins uploading the new frozen view, beginning the cycle anew.

Because frozen views are created at file-system epochs and the state of frozen views is always static, synchronizing frozen views to the cloud provides both crash consistency and causal consistency, given that there is only one client actively synchronizing with the cloud. We call this *single-client consistency*.

3.3.2 Multi-client Consistency

When multiple clients are synchronized with the cloud, the server must propagate the latest synchronized view from one client to other clients, to make all clients' state synchronized. Critically, the server must propagate views in their entirety; partially uploaded views are inherently inconsistent and thus should not be visible. However, because synchronized views necessarily lag behind the active views in each file system, the current active file system may have dependencies that would be invalidated by a remote synchronized view. Thus, remote changes must be applied to the active view in a way that preserves local causal consistency.

To achieve this, ViewBox handles remote changes in two phases. In the first phase, ViewBox applies remote changes to the frozen view. If a changed file does not exist in the frozen view, ViewBox adds it directly; otherwise, it adds the file under a new name that indicates a conflict (e.g., "foo.txt" becomes "remote.foo.txt"). In the second



(a) Directly applying remote updates (b) Merging and handling potential conflicts

Figure 2: **Handling Remote Updates.** *This figure demonstrates two different scenarios where remote updates are handled. While case (a) has no conflicts, case (b) may, because it contains concurrent updates.*

phase, ViewBox merges the newly created frozen view with the active view. ViewBox propagates all changes from the new frozen view to the active view, using the same conflict handling procedure. At the same time, it uploads the newly merged frozen view. Once the second phase completes, the active view is fully updated; only after this occurs can it be frozen and uploaded.

To correctly handle conflicts and ensure no data is lost, we follow the same policy as GIT [14]. This can be summarized by the following three guidelines:

- Preserve any local or remote change; a change could be the addition, modification, or deletion of a file.
- When there is a conflict between a local change and a remote change, always keep the local copy untouched, but rename and save the remote copy.
- Synchronize and propagate both the local copy and the renamed remote copy.

Figure 2 illustrates how ViewBox handles remote changes. In case (a), both the remote and local clients are synchronized with the cloud, at view 0. The remote client makes changes to the active view, and subsequently freezes and uploads it to the cloud as view 1. The local client is then informed of view 1, and downloads it. Since there are no local updates, the client directly applies the changes in view 1 to its frozen view and propagates those changes to the active view.

In case (b), both the local client and the remote client perform updates concurrently, so conflicts may exist. Assuming the remote client synchronizes view 1 to the cloud first, the local client will refrain from uploading its frozen view, view 2, and download view 1 first. It then merges the two views, resolving conflicts as described above, to create a new frozen view, view 3. Finally, the local client uploads view 3 while simultaneously propagating the changes in view 3 to the active view.

In the presence of simultaneous updates, as seen in case (b), this synchronization procedure results in a cloud state that reflects a combination of the disk states of all clients, rather than the state of any one client. Eventually, the different client and cloud states will converge, providing *multi-client consistency*. This model is weaker than our single-client model; thus, ViewBox may not be able to provide causal consistency for each individual client under all circumstances.

Unlike single-client consistency, multi-client consistency requires the cloud server to be aware of views, not just the client. Thus, ViewBox can only provide multi-client consistency for open source services, like Seafile; providing it for closed-source services, like Dropbox, will require explicit cooperation from the service provider.

3.4 Cloud-aided Recovery

With the ability to detect faults and to upload consistent views of the file system state, ViewBox is now capable of performing correct recovery. There are effectively two types of recovery to handle: recovery of corrupt files, and recovery of inconsistent files at the time of a crash.

In the event of corruption, if the file is clean in both the active view and the frozen view, we can simply recover the corrupt block by fetching the copy from the cloud. If the file is dirty, the file may not have been synchronized to the cloud, making direct recovery impossible, as the block fetched from cloud will not match the checksum. If recovering a single block is not possible, the entire file must be rolled back to a previous synchronized version, which may lead to causal inconsistency.

Recovering causally-consistent images of files that were present in the active view at the time of a crash faces the same difficulties as restoring corrupt files in the active view. Restoring each individual file to its most recent synchronized version is not correct, as other files may have been written after the now-corrupted file and, thus, depend on it; to ensure these dependencies are not broken, these files also need to be reverted. Thus, naive restoration can lead to causal inconsistency, even with views.

Instead, we present users with the choice of individually rolling back damaged files, potentially risking causal inconsistency, or reverting to the most recent synchronized view, ensuring correctness but risking data loss. As we anticipate that the detrimental effects of causal inconsistency will be relatively rare, the former option will be usable in many cases to recover, with the latter available in the event of bizarre or unexpected application behavior.

4 Implementation

Now that we have provided a broad overview of ViewBox's architecture, we delve more deeply into the specifics of our implementation. As with Section 3, we divide our discussion based on the three primary components of our architecture: detection, as implemented with our new *ext4-cksum* file system; view-based synchronization using our *view manager*, a file-system agnostic extension to ext4-cksum; and recovery, using a user-space recovery daemon called *cloud helper*.

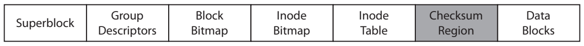


Figure 3: **Ext4-cksum Disk Layout.** *This graph shows the layout of a block group in ext4-cksum. The shaded checksum region contains data checksums for blocks in the block group.*

4.1 Ext4-cksum

Like most file systems that update data in place, ext4 provides minimal facilities for detecting corruption and ensuring data consistency. While it offers experimental metadata checksums, these do not protect data; similarly, its default ordered journaling mode only protects the consistency of metadata, while providing minimal guarantees about data. Thus, it requires changes to meet our requirements for integrity and consistency. We now present ext4-cksum, a variant of ext4 that supports data checksums to protect against data corruption and to detect data inconsistency after a crash without the high cost of data journaling.

4.1.1 Checksum Region

Ext4-cksum stores data checksums in a fixed-sized *checksum region* immediately after the inode table in each block group, as shown in Figure 3. All checksums of data blocks in a block group are preallocated in the checksum region. This region acts similarly to a bitmap, except that it stores checksums instead of bits, with each checksum mapping directly to a data block in the group. Since the region starts at a fixed location in a block group, the location of the corresponding checksum can be easily calculated, given the physical (disk) block number of a data block.

The size of the region depends solely on the total number of blocks in a block group and the length of a checksum, both of which are determined and fixed during file system creation. Currently, ext4-cksum uses the built-in crc32c checksum, which is 32 bits. Therefore, it reserves a 32-bit checksum for every 4KB block, imposing a space overhead of 1/1024; for a regular 128MB block group, the size of the checksum region is 128KB.

4.1.2 Checksum Handling for Reads and Writes

When a data block is read from disk, the corresponding checksum must be verified. Before the file system issues a read of a data block from disk, it gets the corresponding checksum by reading the checksum block. After the file system reads the data block into memory, it verifies the block against the checksum. If the initial verification fails, ext4-cksum will retry. If the retry also fails, ext4-cksum will report an error to the application. Note that in this case, if ext4-cksum is running with the cloud helper daemon, ext4-cksum will try to get the remote copy from cloud and use that for recovery. The read part of a read-modify-write is handled in the same way.

A read of a data block from disk always incurs an additional read for the checksum, but not every checksum read will cause high latency. First, the checksum read can be served from the page cache, because the checksum blocks are considered metadata blocks by ext4-cksum and are kept in the page cache like other metadata structures. Second, even if the checksum read does incur a disk I/O, because the checksum is always in the same block group as the data block, the seek latency will be minimal. Third, to avoid checksum reads as much as possible, ext4-cksum employs a simple prefetching policy: always read 8 checksum blocks (within a block group) at a time. Advanced prefetching heuristics, such as those used for data prefetching, are applicable here.

Ext4-cksum does not update the checksum for a dirty data block until the data block is written back to disk. Before issuing the disk write for the data block, ext4-cksum reads in the checksum block and updates the corresponding checksum. This applies to all data write-backs, caused by a background flush, fsync, or a journal commit.

Since ext4-cksum treats checksum blocks as metadata blocks, with journaling enabled, ext4-cksum logs all dirty checksum blocks in the journal. In ordered journaling mode, this also allows the checksum to detect inconsistent data caused by a crash. In ordered mode, dirty data blocks are flushed to disk before metadata blocks are logged in the journal. If a crash occurs before the transaction commits, data blocks that have been flushed to disk may become inconsistent, because the metadata that points to them still remains unchanged after recovery. As the checksum blocks are metadata, they will not have been updated, causing a mismatch with the inconsistent data block. Therefore, if such a block is later read from disk, ext4-cksum will detect the checksum mismatch.

To ensure consistency between a dirty data block and its checksum, data write-backs triggered by a background flush and fsync can no longer simultaneously occur with a journal commit. In ext4 with ordered journaling, before a transaction has committed, data write-backs may start and overwrite a data block that was just written by the committing transaction. This behavior, if allowed in ext4-cksum, would cause a mismatch between the already logged checksum block and the newly written data block on disk, thus making the committing transaction inconsistent. To avoid this scenario, ext4-cksum ensures that data write-backs due to a background flush and fsync always occur before or after a journal commit.

4.2 View Manager

To provide consistency, ViewBox requires file synchronization services to upload frozen views of the local file system, which it implements through an in-memory file-system extension, the view manager. In this section, we detail the implementation of the view manager, beginning with an overview. Next, we introduce two techniques, cloud journaling and incremental snapshotting, which are key to the consistency and performance provided by the view manager. Then, we provide an example that describes the synchronization process that uploads a frozen view to the cloud. Finally, we briefly discuss how to integrate the synchronization client with the view manager to handle remote changes and conflicts.

4.2.1 View Manager Overview

The view manager is a light-weight kernel module that creates views on top of a local file system. Since it only needs to maintain two local views at any time (one frozen view and one active view), the view manager does not modify the disk layout or data structures of the underlying file system. Instead, it relies on a modified tmpfs to present the frozen view in memory and support all the basic file system operations to files and directories in it. Therefore, a synchronization client now monitors the exposed frozen view (rather than the actual folder in the local file system) and uploads changes from the frozen view to the cloud. All regular file system operations from other applications are still directly handled by ext4-cksum. The view manager uses the active view to track the on-going changes and then reflects them to the frozen view. Note that the current implementation of the view manager is tailored to our ext4-cksum and it is not stackable [29]. We believe that a stackable implementation would make our view manager compatible with more file systems.

4.2.2 Consistency through Cloud Journaling

As we discussed in Section 3.3.1, to preserve consistency, frozen views must be created at file-system epochs. Therefore, the view manager freezes the current active view at the beginning of a journal commit in ext4-cksum, which serves as a boundary between two file-system epochs. At the beginning of a commit, the current running transaction becomes the committing transaction. When a new running transaction is created, all operations belonging to the old running transaction will have completed, and operations belonging to the new running transaction will not have started yet. The view manager freezes the active view at this point, ensuring that no in-flight operation spans multiple views. All changes since the last frozen view are preserved in the new frozen view, which is then uploaded to the cloud, becoming the latest synchronized view.

To ext4-cksum, the cloud acts as an external journaling device. Every synchronized view on the cloud matches a consistent state of the local file system at a specific point in time. Although ext4-cksum still runs in ordered journaling mode, when a crash occurs, the file system now has the chance to roll back to a consistent state stored on cloud. We call this approach cloud journaling.

4.2.3 Low-overhead via Incremental Snapshotting

During cloud journaling, the view manager achieves better performance and lower overhead through a technique called incremental snapshotting. The view manager always keeps the frozen view in memory and the frozen view only contains the data that changed from the previous view. The active view is thus responsible for tracking all the files and directories that have changed since it last was frozen. When the view manager creates a new frozen view, it marks all changed files copy-on-write, which preserves the data at that point. The new frozen view is then constructed by applying the changes associated with the active view to the previous frozen view.

The view manager uses several in-memory and on-cloud structures to support this incremental snapshotting approach. First, the view manager maintains an *inode mapping table* to connect files and directories in the frozen view to their corresponding ones in the active view. The view manager represents the namespace of a frozen view by creating *frozen inodes* for files and directories in tmpfs (their counterparts in the active view are thus called *active inodes*), but no data is usually stored under frozen inodes (unless the data is copied over from the active view due to copy-on-write). When a file in the frozen view is read, the view manager finds the active inode and fetches data blocks from it. The inode mapping table thus serves as a translator between a frozen inode and its active inode.

Second, the view manager tracks namespace changes in the active view by using an *operation log*, which records all successful namespace operations (e.g., create, mkdir, unlink, rmdir, and rename) in the active view. When the active view is frozen, the log is replayed onto the previous frozen view to bring it up-to-date, reflecting the new state.

Third, the view manager uses a *dirty table* to track what files and directories are modified in the active view. Once the active view becomes frozen, all these files are marked copy-on-write. Then, by generating inotify events based on the operation log and the dirty table, the view manager is able to make the synchronization client check and upload these local changes to the cloud.

Finally, the view manager keeps *view metadata* on the server for every synchronized view, which is used to identify what files and directories are contained in a synchronized view. For services such as Seafile, which internally keeps the modification history of a folder as a series of snapshots [24], the view manager is able to use its snapshot ID (called commit ID by Seafile) as the view metadata. For services like Dropbox, which only provides file-level versioning, the view manager creates a view metadata file for every synchronized view, consisting of a list of pathnames and revision numbers of files in that view. The information is obtained by querying the Dropbox server. The view manager stores these metadata files in a hidden folder on the cloud, so the correctness of these files is not affected by disk corruption or crashes.

4.2.4 Uploading Views to the Cloud

Now, we walk through an example in Figure 4 to explain how the view manager uploads views to the cloud. In the

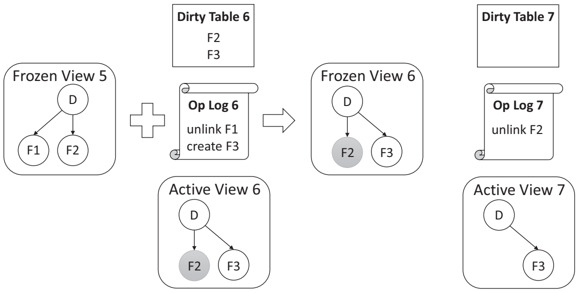


Figure 4: **Incremental Snapshotting.** *This figure illustrates how the view manager creates active and frozen views.*

example, the synchronization service is Dropbox.

Initially, the synchronization folder (D) contains two files (F1 and F2). While frozen view 5 is being synchronized, in active view 6, F1 is deleted, F2 is modified, and F3 is created. The view manager records the two namespace operations (unlink and create) in the operation log, and adds F2 and F3 to the dirty table. When frozen view 5 is completely uploaded to the cloud, the view manager creates a view metadata file and uploads it to the server.

Next, the view manager waits for the next journal commit and freezes active view 6. The view manager first marks F2 and F3 in the dirty table copy-on-write, preserving new updates in the frozen view. Then, it creates active view 7 with a new operation log and a new dirty table, allowing the file system to operate without any interruption. After that, the view manager replays the operation log onto frozen view 5 such that the namespace reflects the state of frozen view 6.

Finally, the view manager generates inotify events based on the dirty table and the operation log, thus causing the Dropbox client to synchronize the changes to the cloud. Since F3 is not changed in active view 7, the client reading its data from the frozen view would cause the view manager to consult the inode mapping table (not shown in the figure) and fetch requested data directly from the active view. Note that F2 is deleted in active view 7. If the deletion occurs before the Dropbox client is able to upload F2, all data blocks of F2 are copied over and attached to the copy of F2 in the frozen view. If Dropbox reads the file before deletion occurs, the view manager fetches those blocks from active view 7 directly, without making extra copies. After frozen view 6 is synchronized to the cloud, the view manager repeats the steps above, constantly uploading views from the local system.

4.2.5 Handling Remote Changes

All the techniques we have introduced so far focus on how to provide single-client consistency and do not require modifications to the synchronization client or the server. They work well with proprietary synchronization services such as Dropbox. However, when there are multiple clients running ViewBox and performing updates at the same time, the synchronization service itself must be view-aware. To handle remote updates correctly, we modify the Seafile client to perform the two-phase synchronization described in Section 3.3.2. We choose Seafile to implement multi-client consistency, because both its client and server are open-source. More importantly, its data model and synchronization algorithm is similar to GIT, which fits our view-based synchronization well.

4.3 Cloud Helper

When corruption or a crash occurs, ViewBox performs recovery using backup data on the cloud. Recovery is performed through a user-level daemon, cloud helper. The daemon is implemented in Python, which acts as a bridge between the local file system and the cloud. It interacts with the local file system using ioctl calls and communicates with the cloud through the service's web API.

For data corruption, when ext4-cksum detects a checksum mismatch, it sends a block recovery request to the cloud helper. The request includes the pathname of the corrupted file, the offset of the block inside the file, and the block size. The cloud helper then fetches the requested block from the server and returns the block to ext4-cksum. Ext4-cksum reverifies the integrity of the block against the data checksum in the file system and returns the block to the application. If the verification still fails, it is possibly because the block has not been synchronized or because the block is fetched from a different file in the synchronized view on the server with the same pathname as the corrupted file.

When a crash occurs, the cloud helper performs a scan of the ext4-cksum file system to find potentially inconsistent files. If the user chooses to only roll back those inconsistent files, the cloud helper will download them from the latest synchronized view. If the user chooses to roll back the whole file system, the cloud helper will identify the latest synchronized view on the server, and download files and construct directories in the view. The former approach is able to keep most of the latest data but may cause causal inconsistency. The latter guarantees causal consistency, but at the cost of losing updates that took place during the frozen view and the active view when the crash occurred.

5 Evaluation

We now present the evaluation results of our ViewBox prototype. We first show that our system is able to recover from data corruption and crashes correctly and provide causal consistency. Then, we evaluate the underlying ext4-cksum and view manager components separately, without synchronization services. Finally we study the overall synchronization performance of ViewBox with Dropbox and Seafile.

We implemented ViewBox in the Linux 3.6.11 kernel, with Dropbox client 1.6.0, and Seafile client and server

Service	**Data**	**Metadata**		
ViewBox w/	**write**	**mtime**	**ctime**	**atime**
Dropbox	*DR*	*DR*	*DR*	*DR*
Seafile	*DR*	*DR*	*DR*	*DR*

Table 4: **Data Corruption Results of ViewBox.** *In all cases, the local corruption is detected (D) and recovered (R) using data on the cloud.*

Service ViewBox w/	**Upload** **local ver.**	**Download** **cloud ver.**	**Out-of-sync** **(no sync)**
Dropbox	×	√	×
Seafile	×	√	×

Table 5: **Crash Consistency Results of ViewBox.** *The local version is inconsistent and rolled back to the previous version on the cloud.*

Workload	**ext4** **(MB/s)**	**ext4-cksum** **(MB/s)**	**Slowdown**
Seq. write	103.69	99.07	4.46%
Seq. read	112.91	108.58	3.83%
Rand. write	0.70	0.69	1.42%
Rand. read	5.82	5.74	1.37%

Table 6: **Microbenchmarks on ext4-cksum.** *This figure compares the throughput of several micro benchmarks on ext4 and ext4-cksum. Sequential write/read are writing/reading a 1GB file in 4KB requests. Random write/read are writing/reading 128MB of a 1GB file in 4KB requests. For sequential read workload, ext4-cksum prefetches 8 checksum blocks for every disk read of a checksum block.*

Workload	**ext4** **(MB/s)**	**ext4-cksum** **(MB/s)**	**Slowdown**
Fileserver	79.58	66.28	16.71%
Varmail	2.90	3.96	-36.55%
Webserver	150.28	150.12	0.11%

Table 7: **Macrobenchmarks on ext4-cksum.** *This table shows the throughput of three workloads on ext4 and ext4-cksum. Fileserver is configured with 50 threads performing creates, deletes, appends, and whole-file reads and writes. Varmail emulates a multi-threaded mail server in which each thread performs a set of create-append-sync, read-append-sync, read, and delete operations. Webserver is a multi-threaded read-intensive workload.*

1.8.0. All experiments are performed on machines with a 3.3GHz Intel Quad Core CPU, 16GB memory, and a 1TB Hitachi Deskstar hard drive. For all experiments, we reserve 512MB of memory for the view manager.

5.1 Cloud Helper

We first perform the same set of fault injection experiments as in Section 2. The corruption and crash test results are shown in Table 4 and Table 5. Because the local state is initially synchronized with the cloud, the cloud helper is able to fetch the redundant copy from cloud and recover from corruption and crashes. We also confirm that ViewBox is able to preserve causal consistency.

5.2 Ext4-cksum

We now evaluate the performance of standalone ext4-cksum, focusing on the overhead caused by data checksumming. Table 6 shows the throughput of several microbenchmarks on ext4 and ext4-cksum. From the table, one can see that the performance overhead is quite minimal. Note that checksum prefeteching is important for sequential reads; if it is disabled, the slowdown of the workload increases to 15%.

We perform a series of macrobenchmarks using Filebench on both ext4 and ext4-cksum with checksum prefetching enabled. The results are shown in Table 7. For the fileserver workload, the overhead of ext4-cksum is quite high, because there are 50 threads reading and writing concurrently and the negative effect of the extra seek for checksum blocks accumulates. The webserver workload, on the other hand, experiences little overhead, because it is dominated by warm reads.

It is surprising to notice that ext4-cksum greatly outperforms ext4 in varmail. This is actually a side effect of the ordering of data write-backs and journal commit, as discussed in Section 4.1.2. Note that because ext4 and ext4-cksum are not mounted with "journal_async_commit", the commit record is written to disk with a cache flush and the FUA (force unit access) flag, which ensures that when the commit record reaches disk, all previous dirty data (including metadata logged in the journal) have already been forced to disk. When running varmail in ext4, data blocks written by fsyncs from other threads during the journal commit are also flushed to disk at the same time, which causes high latency. In contrast, since ext4-cksum does not allow data write-back from fsync to run simultaneously with the journal commit, the amount of data flushed is much smaller, which improves the overall throughput of the workload.

5.3 View Manager

We now study the performance of various file system operations in an active view when a frozen view exists. The view manager runs on top of ext4-cksum.

We first evaluate the performance of various operations that do not cause copy-on-write (COW) in an active view. These operations are create, unlink, mkdir, rmdir, rename, utime, chmod, chown, truncate and stat. We run a workload that involves creating 1000 8KB files across 100 directories and exercising these operations on those files and directories. We prevent the active view from being frozen so that all these operations do not incur a COW. We see a

Operation	Normalized Response Time Before COW	 After COW
unlink (cold)	484.49	1.07
unlink (warm)	6.43	0.97
truncate (cold)	561.18	1.02
truncate (warm)	5.98	0.93
rename (cold)	469.02	1.10
rename (warm)	6.84	1.02
overwrite (cold)	1.56	1.10
overwrite (warm)	1.07	0.97

Table 8: **Copy-on-write Operations in the View Manager.** *This table shows the normalized response time (against ext4) of various operations on a frozen file (10MB) that trigger copy-on-write of data blocks. "Before COW"/"After COW" indicates the operation is performed before/after affected data blocks are COWed.*

small overhead (mostly less than 5% except utime, which is around 10%) across all operations, as compared to their performance in the original ext4., This overhead is mainly caused by operation logging and other bookkeeping performed by the view manager.

Next, we show the normalized response time of operations that do trigger copy-on-write in Table 8. These operations are performed on a 10MB file after the file is created and marked COW in the frozen view. All operations cause all 10MB of file data to be copied from the active view to the frozen view. The copying overhead is listed under the "Before COW" column, which indicates that these operations occur before the affected data blocks are COWed. When the cache is warm, which is the common case, the data copying does not involve any disk I/O but still incurs up to 7x overhead. To evaluate the worst case performance (when the cache is cold), we deliberately force the system to drop all caches before we perform these operations. As one can see from the table, all data blocks are read from disk, thus causing much higher overhead. Note that cold cache cases are rare and may only occur during memory pressure. We further measure the performance of the same set of operations on a file that has already been fully COWed. As shown under the "After COW" column, the overhead is negligible, because no data copying is performed.

5.4 ViewBox with Dropbox and Seafile

We assess the overall performance of ViewBox using three workloads: openssh (building openssh from its source code), iphoto_edit (editing photos in iPhoto), and iphoto_view (browsing photos in iPhoto). The latter two workloads are from the iBench trace suite [15] and are replayed using Magritte [27]. We believe that these workloads are representative of ones people run with synchronization services.

The results of running all three workloads on ViewBox with Dropbox and Seafile are shown in Table 9. In all cases, the runtime of the workload in ViewBox is at most 5% slower and sometimes faster than that of the unmodified ext4 setup, which shows that view-based synchronization does not have a negative impact on the foreground workload. We also find that the memory overhead of ViewBox (the amount of memory consumed by the view manager to store frozen views) is minimal, at most 20MB across all three workloads.

We expect the synchronization time of ViewBox to be longer because ViewBox does not start synchronizing the current state of the file system until it is frozen, which may cause delays. The results of openssh confirm our expectations. However, for iphoto_view and iphoto_edit, the synchronization time on ViewBox with Dropbox is much greater than that on ext4. This is due to Dropbox's lack of proper interface support for views, as described in Section 4.2.3. Because both workloads use a file system image with around 1200 directories, to create the view metadata for each view, ViewBox has to query the Dropbox server numerous times, creating substantial overhead. In contrast, ViewBox can avoid this overhead with Seafile because it has direct access to Seafile's internal metadata. Thus, the synchronization time of iphoto_view in ViewBox with Seafile is near that in ext4.

Note that the iphoto_edit workload actually has a much shorter synchronization time on ViewBox with Seafile than on ext4. Because the photo editing workload involves many writes, Seafile delays uploading when it detects files being constantly modified. After the workload finishes, many files have yet to be uploaded. Since frozen views prevent interference, ViewBox can finish synchronizing about 30% faster.

6 Related Work

ViewBox is built upon various techniques, which are related to many existing systems and research work.

Using checksums to preserve data integrity and consistency is not new; as mentioned in Section 2.3, a number of existing file systems, including ZFS, btrfs, WAFL, and ext4, use them in various capacities. In addition, a variety of research work, such as IRON ext3 [22] and Z^2FS [31], explores the use of checksums for purposes beyond simply detecting corruption. IRON ext3 introduces transactional checksums, which allow the journal to issue all writes, including the commit block, concurrently; the checksum detects any failures that may occur. Z^2FS uses page cache checksums to protect the system from corruption in memory, as well as on-disk. All of these systems rely on locally stored redundant copies for automatic recovery, which may or may not be available. In contrast, ext4-cksum is the first work of which we are aware that employs the cloud for recovery. To our knowledge, it is also the first work to add data checksumming to ext4.

Similarly, a number of works have explored means

	ext4 + Dropbox		ViewBox with Dropbox		ext4 + Seafile		ViewBox with Seafile	
Workload	Runtime	Sync Time	Runtime	Sync Time	Runtime	Sync Time	Runtime	Sync Time
openssh	36.4	49.0	36.0	64.0	36.0	44.8	36.0	56.8
iphoto_edit	577.4	2115.4	563.0	2667.3	566.6	857.6	554.0	598.8
iphoto_view	149.2	170.8	153.4	591.0	150.0	166.6	156.4	175.4

Table 9: **ViewBox Performance.** *This table compares the runtime and sync time (in seconds) of various workloads running on top of unmodified ext4 and ViewBox using both Dropbox and Seafile. Runtime is the time it takes to finish the workload and sync time is the time it takes to finish synchronizing.*

of providing greater crash consistency than ordered and metadata journaling provide. Data journaling mode in ext3 and ext4 provides full crash consistency, but its high overhead makes it unappealing. OptFS [7] is able to achieve data consistency and deliver high performance through an optimistic protocol, but it does so at the cost of durability while still relying on data journaling to handle overwrite cases. In contrast, ViewBox avoids overhead by allowing the local file system to work in ordered mode, while providing consistency through the views it synchronizes to the cloud; it then can restore the latest view after a crash to provide full consistency. Like OptFS, this sacrifices durability, since the most recent view on the cloud will always lag behind the active file system. However, this approach is optional, and, in the normal case, ordered mode recovery can still be used.

Due to the popularity of Dropbox and other synchronization services, there are many recent works studying their problems. Our previous work [30] examines the problem of data corruption and crash inconsistency in Dropbox and proposes techniques to solve both problems. We build ViewBox on these findings and go beyond the original proposal by introducing view-based synchronization, implementing a prototype system, and evaluating our system with various workloads. Li et al. [19] notice that frequent and short updates to files in the Dropbox folder generate excessive amounts of maintenance traffic. They propose a mechanism called update-batched delayed synchronization (UDS), which acts as middleware between the synchronized Dropbox folder and an actual folder on the file system. UDS batches updates from the actual folder and applies them to the Dropbox folder at once, thus reducing the overhead of maintenance traffic. The way ViewBox uploads views is similar to UDS in that views also batch updates, but it differs in that ViewBox is able to batch all updates that reflect a consistent disk image while UDS provides no such guarantee.

7 Conclusion

Despite their near-ubiquity, file synchronization services ultimately fail at one of their primary goals: protecting user data. Not only do they fail to prevent corruption and inconsistency, they actively spread it in certain cases. The fault lies equally with local file systems, however, as they often fail to provide the necessary capabilities that would allow synchronization services to catch these errors. To remedy this, we propose ViewBox, an integrated system that allows the local file system and the synchronization client to work together to prevent and repair errors.

Rather than synchronize individual files, as current file synchronization services do, ViewBox centers around views, in-memory file-system snapshots which have their integrity guaranteed through on-disk checksums. Since views provide consistent images of the file system, they provide a stable platform for recovery that minimizes the risk of restoring a causally inconsistent state. As they remain in-memory, they incur minimal overhead.

We implement ViewBox to support both Dropbox and Seafile clients, and find that it prevents the failures that we observe with unmodified local file systems and synchronization services. Equally importantly, it performs competitively with unmodified systems. This suggests that the cost of correctness need not be high; it merely requires adequate interfaces and cooperation.

Acknowledgments

We thank the anonymous reviewers and Jason Flinn (our shepherd) for their comments. We also thank the members of the ADSL research group for their feedback. This material is based upon work supported by the NSF under CNS-1319405, CNS-1218405, and CCF-1017073 as well as donations from EMC, Facebook, Fusion-io, Google, Huawei, Microsoft, NetApp, Sony, and VMware. Any opinions, findings, and conclusions, or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the NSF or other institutions.

References

[1] lvcreate(8) - linux man page.

[2] ZFS on Linux. http://zfsonlinux.org.

[3] Amazon. Amazon Simple Storage Service (Amazon S3). http://aws.amazon.com/s3/.

[4] Apple. Technical Note TN1150. http://developer.apple.com/technotes/tn/tn1150.html, March 2004.

[5] Lakshmi N. Bairavasundaram, Garth R. Goodson, Bianca Schroeder, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. An Analysis of Data Corruption in the Storage Stack. In *Proceedings of the 6th USENIX Symposium on File and Storage Technologies (FAST '08)*, San Jose, California, February 2008.

[6] Jeff Bonwick and Bill Moore. ZFS: The Last Word in File Systems. http://opensolaris.org/os/community/zfs/docs/zfs˙last.pdf, 2007.

[7] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Optimistic Crash Consistency. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, Farmington, PA, November 2013.

[8] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An Empirical Study of Operating System Errors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 73–88, Banff, Canada, October 2001.

[9] Jonathan Corbet. Improving ext4: bigalloc, inline data, and metadata checksums. http://lwn.net/Articles/469805/, November 2011.

[10] Idilio Drago, Marco Mellia, Maurizio M. Munafò, Anna Sperotto, Ramin Sadre, and Aiko Pras. Inside Dropbox: Understanding Personal Cloud Storage Services. In *Proceedings of the 2012 ACM conference on Internet measurement conference (IMC '12)*, Boston, MA, November 2012.

[11] Dropbox. The dropbox tour. https://www.dropbox.com/tour.

[12] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 57–72, Banff, Canada, October 2001.

[13] Google. Google drive. http://www.google.com/drive/about.html.

[14] David Greaves, Junio Hamano, et al. git-read-tree(1): -linux man page. http://linux.die.net/man/1/git-read-tree.

[15] Tyler Harter, Chris Dragga, Michael Vaughn, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. A File is Not a File: Understanding the I/O Behavior of Apple Desktop Applications. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '11)*, pages 71–83, Cascais, Portugal.

[16] Dave Hitz, James Lau, and Michael Malcolm. File System Design for an NFS File Server Appliance. In *Proceedings of the USENIX Winter Technical Conference (USENIX Winter '94)*, San Francisco, California, January 1994.

[17] Minwen Ji, Alistair C Veitch, and John Wilkes. Seneca: remote mirroring done write. In *Proceedings of the USENIX Annual Technical Conference (USENIX '03)*, San Antonio, Texas, June 2003.

[18] Andrew Krioukov, Lakshmi N. Bairavasundaram, Garth R. Goodson, Kiran Srinivasan, Randy Thelen, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Parity Lost and Parity Regained. In *Proceedings of the 6th USENIX Symposium on File and Storage Technologies (FAST '08)*, pages 127–141, San Jose, California, February 2008.

[19] Zhenhua Li, Christo Wilson, Zhefu Jiang, Yao Liu, Ben Y. Zhao, Cheng Jin, Zhi-Li Zhang, and Yafei Dai. Efficient Batched Synchronization in Dropbox-like Cloud Storage Services. In *Proceedings of the 14th International Middleware Conference (Middleware 13')*, Beijing, China, December 2013.

[20] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, Laurent Vivier, and Bull S.A.S. The New Ext4 Filesystem: Current Status and Future Plans. In *Ottawa Linux Symposium (OLS '07)*, Ottawa, Canada, July 2007.

[21] Microsoft. How ntfs works. http://technet.microsoft.com/en-us/library/cc781134(v=ws.10).aspx, March 2003.

[22] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. IRON File Systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, pages 206–220, Brighton, United Kingdom, October 2005.

[23] Ohad Rodeh, Josef Bacik, and Chris Mason. BTRFS: The Linux B-Tree Filesystem. *ACM Transactions on Storage (TOS)*, 9(3):9:1–9:32, August 2013.

[24] Seafile. Seafile. http://seafile.com/en/home/.

[25] Adan Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the XFS File System. In *Proceedings of the USENIX Annual Technical Conference (USENIX '96)*, San Diego, California, January 1996.

[26] Stephen C. Tweedie. Journaling the Linux ext2fs File System. In *The Fourth Annual Linux Expo*, Durham, North Carolina, May 1998.

[27] Zev Weiss, Tyler Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. ROOT: Replaying Multithreaded Traces with Resource-Oriented Ordering. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, Farmington, PA, November 2013.

[28] Microsoft Windows. Skydrive. http://windows.microsoft.com/en-us/skydrive/download.

[29] Erez Zadok, Ion Badulescu, and Alex Shender. Extending File Systems Using Stackable Templates. In *Proceedings of the USENIX Annual Technical Conference (USENIX '99)*, Monterey, California, June 1999.

[30] Yupu Zhang, Chris Dragga, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. *-Box: Towards Reliability and Consistency in Dropbox-like File Synchronization Services. In *Proceedings of the 5th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage '13)*, San Jose, California, June 2013.

[31] Yupu Zhang, Daniel S. Myers, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Zettabyte Reliability with Flexible End-to-end Data Integrity. In *Proceedings of the 29th IEEE Conference on Massive Data Storage (MSST '13)*, Long Beach, CA, May 2013.