Lock-Free Collaboration Support for Cloud Storage Services with Operation Inference and Transformation [*]

Jian Chen[1*], Minghao Zhao[1*], Zhenhua Li[1✉], Ennan Zhai[2]
Feng Qian[3], Hongyi Chen[1], Yunhao Liu[1,4], Tianyin Xu[5]
[1]*Tsinghua University,* [2]*Alibaba Group,* [3]*University of Minnesota,* [4]*Michigan State University,* [5]*UIUC*

Abstract

This paper studies how today's cloud storage services support collaborative file editing. As a tradeoff for transparency/user-friendliness, they do not ask collaborators to use version control systems but instead implement their own heuristics for handling conflicts, which however often lead to unexpected and undesired experiences. With measurements and reverse engineering, we unravel a number of their design and implementation issues as the root causes of poor experiences. Driven by the findings, we propose to reconsider the collaboration support of cloud storage services from a novel perspective of *operations* without using any locks. To enable this idea, we design intelligent approaches to the inference and transformation of users' editing operations, as well as optimizations to the maintenance of files' historic versions. We build an open-source system UFC2 (User-Friendly Collaborative Cloud) to embody our design, which can avoid most (98%) conflicts with little (2%) overhead.

1 Introduction

Computer-supported collaboration allows a group of geo-distributed people (*i.e.,* collaborators) to cooperatively work online. To enable this, the most common technique is Version Control Systems (VCSes) like Git, SVN and Mercurial, which require the mastery of complex operations and thus are not suited to non-technical users [58]. In contrast, dedicated online editors, such as Google Docs and Overleaf, provide web-based easy-to-use collaboration support, but with limited functions and "walled-garden" concerns [8, 10, 13, 68]. As an alternative approach, cloud storage services (*e.g.,* Dropbox, OneDrive, Google Drive, and iCloud) have recently evolved their functionality from simple file backup to online collaboration. For example, over 300,000 teams have adopted Dropbox for business collaboration, submitting ∼4000 file edits per second [62]. For ease of use, collaboration is made transparent by almost every service today through *automatic file synchronization*. When a user modifies a file in a "sync folder" (a local directory created by the service), the changed file will be automatically synchronized with the copy maintained at the cloud side. Then, the cloud will further distribute the new version of the file to the other users sharing the file.

[*]Co-primary authors. Zhenhua Li is the corresponding author.

Pattern 1: Losing updates Alice is editing a file. Suddenly, her file is overwritten by a new version from her collaborator, Bob. Sometimes, Alice can even lose her edits on the older version.	All studied cloud storage services
Pattern 2: Conflicts despite coordination Alice coordinates her edits with Bob through emails to avoid conflicts by enforcing a sequential order. Every edit is saved instantly. Even so, conflicts still occur.	All studied cloud storage services
Pattern 3: Excessively long sync duration Alice edits a shared file and confirms that the edit has been synced to the cloud. However, Bob does not receive the updates for an excessively long duration.	Dropbox, OneDrive, SugarSync, Seafile, Box
Pattern 4: Blocking collaborators by opening files Alice simply opens a shared Microsoft Office file without making any edits. This mysteriously disables Bob's editing the file.	Seafile (only for Microsoft Office files)

Table 1: Common patterns of unexpected and undesired collaborative editing experiences studied in this paper.

Collaboration inevitably introduces *conflicts* – simultaneous edits on two different copies of the same file. However, it is non-trivial to automatically resolve conflicts, especially if the competing edits are on the same line of the file. Instead of requiring users to learn complex diff-and-merge instructions to solve conflicts in VCSes, all of today's cloud storage services opt for transparency and user-friendliness – they devise different approaches to preventing conflicts or automatically resolving conflicts. Unfortunately, these efforts do not work well in practice, often resulting in unexpected results. Table 1 describes four common patterns of unexpected/undesirable collaborative experiences caused by cloud storage services.

To "debug" these patterns from the inside out, we study eight widely-used cloud storage services based on traffic analysis with trace-driven experiments and reverse engineering. The studied services include Dropbox, OneDrive, Google Drive, iCloud Drive, Box [2], SugarSync [20], Seafile [16], and Nutstore [11]. Also, we collect ten real-world collaboration traces, among which seven come from the users of different services and the other three come from the contributors of well-known projects hosted by Github. Our study results reveal a number of design issues of collaboration support in today's cloud storage services. Specifically, we find:

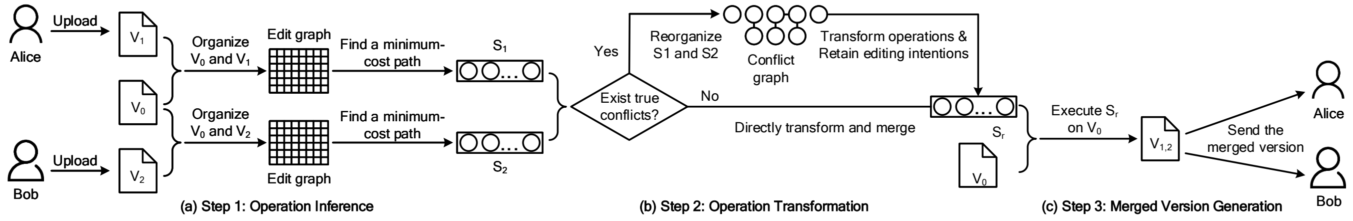

Figure 1: Working principle for merging two versions of the same file at the cloud side: (a) inferring the operation sequences S_1 and S_2 that respectively change V_0 to V_1 and V_2 using edit graphs; (b) transforming and merging S_1 and S_2 into S_r with the minimal conflict, based on a conflict graph and topological sorting when necessary; (c) executing S_r on V_0 to generate the merged version $V_{1,2}$.

- Using file-level locks to prevent conflicts is difficult due to the unpredictability of users' real-time editing behavior (as cloud storage services can neither designate nor monitor the editor) and the latency between clients and the server.
- Existing conflict-resolution solutions are too coarse-grained and do not consider user intention – they either keep the latest version based on the server-side timestamp or distribute all the conflicting versions to the users.

Most surprisingly, we observe that the majority of "conflicts" reported by these cloud storage services are not *true conflicts* but are artificially created. In those false-positive conflicts (or false conflicts), the collaborators were editing different parts of a shared file. This is echoed by the common practice of mitigating false conflicts in cloud storage service-based collaborative editing by intentionally dividing an entire text file into multiple separate files [18, 23]. Such false conflicts can be automatically resolved at the server side without user intervention.

In this paper, we show that it is feasible to provide effective collaboration support in cloud storage services by intelligently merging conflicting file versions using the *three-way merge* method [54, 63], where two conflicting versions are merged based on a common-context version. This is enabled by the inference and transformation of users' editing operations; meanwhile, no lock is used so as to achieve the transparency and user-friendliness. As depicted in Figure 1, our basic idea is to first infer the collaborators' operation sequences [1(a)], and then transform these sequences based on their true conflicts (if any) [1(b)] to generate the final version [1(c)]. Compared to a file-level or line-level conflict resolution (*e.g.,* adopted by Dropbox or Git), our solution is more fine-grained: modifications on different parts of the same file or even the same line can be automatically merged.

Building a system with the above idea, however, requires us to address two technical challenges. First, inferring operation sequences in an efficient way is non-trivial, since it is a computation-intensive task for cloud storage services[1]. As illustrated in Figure 1(a), when two versions V_1 and V_2 emerge, we need to first find the latest *common-context* version V_0 hosted at the cloud, and then infer two operation sequences S_1 and S_2 that convert V_0 to V_1 and V_2, respectively. The common approach using dynamic programming [33, 44, 57] may take excessive computing time in our scenario, *e.g.,* $\sim$30 seconds for a 500-KB file. To address the issue, we leverage an *edit graph* [4, 55] to organize V_0 and V_1, and thus essentially reduce the inference time, *e.g.,* $\sim$200 ms for a 500-KB file.

The second challenge is how to transform and merge S_1 and S_2 into S_r with *minimal conflict*, *i.e.,* 1) simplifying manual conflict resolution of text files by sending only one merged version ($V_{1,2}$) to the collaborators; and 2) retaining the collaborators' editing intentions while minimizing the amount of conflicts to be manually resolved in $V_{1,2}$. As illustrated in Figure 1(b), it is easy to directly transform and merge S_1 and S_2, via *operation transformation* [39], if there is no true conflict. To address the challenging case (of true conflicts), we utilize a *conflict graph* [53] coupled with *topological sorting* to reorganize all operations, so as to prioritize the transformation of real conflicting operations and minimize their impact on the transformation of other operations.

Besides solving the above challenges, we facilitate conflict resolution by maintaining each shared file's historic versions at the cloud with CDC (content-defined chunking [59]) deduplication. For a user-uploaded version, we adopt full-file sync for small files and delta sync for larger files to achieve the shortest upload time. For a server-merged version, we design *operation-based CDC* (OCDC) which exploits the implicit operations inferred during conflict resolution to accelerate CDC – only the boundaries of those chunks affected by the operations need recalculation.

We build UFC2 (User-Friendly Collaborative Cloud) on top of Amazon EFS (Elastic File System) and S3 to implement our design. Our evaluation using real-world traces indicates that conflicts generated during collaboration are significantly reduced by 98% on average (the remainder are true conflicts). Meanwhile, the incurred time overhead by a conflict resolution is usually between 10 and 80 ms, which is merely 0.6%–4% (2% on average) of the delivery time for a file update. In addition, our designed OCDC optimization outpaces the traditional CDC by $\sim$3 times, thus reducing the data chunking time from 30–400 ms to 10–120 ms for a common file. Finally, we have made all the source code and measurement data publicly available at `https://UFC2.github.io`.

[1]In contrast, it is straightforward and lightweight to acquire a user's operation sequences in Google Docs [7], Overleaf [15], and similar services, where a dedicated editor is used and monitored in real time.

Trace	Timespan	# Col-s	# Files	# Versions	Avg. File Size	Major File Types
Dropbox-1	11/2/2018–2/6/2019	5	305	3527	86 KB	tex (52%), pdf (16%), Matlab src (24%) & fig (4%)
Dropbox-2	4/3/2019–5/14/2019	6	216	2193	67 KB	tex (57%), pdf (21%), Matlab fig (9%)
OneDrive	3/15/2019–5/31/2019	5	253	2673	83 KB	tex (61%), pdf (15%), Matlab fig (7%)
iCloud Drive	2/1/2019–4/30/2019	6	301	3211	59 KB	tex (53%), pdf (22%), Matlab fig (12%)
Box	3/21/2019–5/2/2019	8	273	2930	60 KB	tex (66%), pdf (27%)
SugarSync	4/11/2019–5/26/2019	9	325	3472	89 KB	tex (49%), pdf (25%), Matlab src (19%) & fig (3%)
Seafile	2/17/2019–4/30/2019	7	251	2823	71 KB	tex (55%), pdf (19%), Matlab fig (10%)
Spark-Git	1/15/2018–3/27/2019	58	15181	129957	4 KB	Scala (78%), Java (6%), py (5%)
TensorFlow-Git	7/24/2018–3/27/2019	86	16754	246016	9 KB	py (30%), C header (14%) & src (29%), txt (20%)
Linux-Git	9/9/2018–3/30/2019	87	63865	901167	13 KB	C header (31%) & src (42%), txt (16%)

Table 2: Statistics of the ten real-world collaboration traces. "Col-s" means collaborators, "src" means source code, and "py" means python.

2 Design Challenges

In this section, we employ trace-driven experiments, special benchmarks, and reverse engineering to deeply understand the design challenges of collaborative support in today's cloud storage services. In particular, we analyze the root causes of poor experiences listed in Table 1.

2.1 Study Methodology

In order to quantitatively understand how today's cloud storage services behave under typical collaborative editing workloads, we first collected ten real-world collaboration traces as listed in Table 2. Among them, seven are provided by users (with informed consent) that collaborate on code/document writing using different cloud storage services. The other three are extracted from well-known open-source GitHub projects. Each trace contains all the file versions uploaded by every involved user during the collection period.

For the first seven traces, relatively few (*i.e.,* 5–9) collaborators work on a project for a couple of months. Each of their workloads is unevenly distributed over time: during some periods collaborators frequently edit the shared files, whereas during the other periods there are scarcely any edits to the shared files. By contrast, in the last three traces, a large number of collaborators constantly submit their edits for quite a few months, and thus generate many more file versions. In addition, the collaborators involved in all the ten traces are located across multiple continents.

Using these traces, we conducted a comparative measurement study of eight mainstream cloud storage services: Dropbox, OneDrive, Google Drive, iCloud Drive, Box, SugarSync, Seafile, and Nutstore. For each service, we ran its latest PC client (as of Jul. 2019) on Windows-10 VMs rented from Amazon EC2; these VMs have the same hardware configuration (a dual-core CPU@2.5 GHz, 8 GB memory, and 32 GB SSD storage) and network connection (whose downlink/uplink bandwidth is restricted to 100 / 20 Mbps by WonderShaper to resemble a typical residential network connection [1, 19]).

We deployed puppet collaborators on geographically distributed VMs across five major regions to replay a trace, with one client software and one puppet collaborator running on one VM. Specifically, we rented AWS VMs in South America, North America, Europe, the Middle-East, and the Asia-Pacific (including East Asia and Australia). We instructed the puppet collaborators to upload different file versions (as recorded in the trace) to the cloud. To safely reduce the duration of the replay, we skipped the "idle" timespan in the trace during which no file is edited by any collaborator. In addition, we strategically generated some "corner cases" that seldom appear in users' normal editing, so as to make a deeper and more comprehensive analysis. For example, we edited fix-sized small (KB-level) files to measure cloud storage services' sync delay, so as to avoid the impact of file size variation; we edited a random byte on a compressed file to figure out their adoption of delta sync mechanisms; and we performed specially controlled edits to investigate their usage of locks, as well as their delivery time of lock status.

We captured all the IP-level sync traffic in the trace-driven and benchmark experiments via Wireshark [25]. From the traffic, we observe that almost all the communications during the collaboration are carried out with HTTPS sessions (using TLS v1.1 or v1.2). By analyzing the traffic size and occurrence time of respective HTTPS sessions, we can understand the basic design of these eight mainstream cloud storage services, *e.g.,* using full-file sync or delta sync mechanisms to deliver a file update.

To reverse engineer the implementation details, we attempted to reverse HTTPS by leveraging man-in-the-middle attacks with Charles [3], and succeeded with OneDrive, Box, and Seafile. For the three services, we are able to get the detailed information of each synced file (including its ID, creation time, edit time, and to our great surprise the concrete content), as well as the delivered lock status and file update. Furthermore, since Seafile is open source, we also read the source code to understand the system design and implementation, *e.g.,* its adoption of FIFO message queues and the CDC delta sync algorithm.

For the remaining five cloud storage services, we are unable to reverse their HTTPS sessions, as their clients do not accept the root CA certificates forged by Charles. For these services, we search the technical documentation (including design documents and engineering blogs) to learn about their designs, such as locks and message queues [5, 9, 12, 14, 21, 22, 31].

Cloud Storage Service	Lock Mechanism	Conflict Resolution	Message Queue	File Update Method
Dropbox	No lock	Keep all the conflicting versions	LIFO	`rsync`
OneDrive	No lock	Keep all the conflicting versions	Queue	Full-file sync
Google Drive	No lock	Keep only the latest version	-	Full-file sync
iCloud Drive	No lock	Ask users to choose among multiple versions	-	`rsync`
Box	Manual locking	Keep all the conflicting versions	Queue	Full-file sync
SugarSync	No lock	Keep all the conflicting versions	Queue	`rsync`
Seafile	Automatic/manual*	Keep all the conflicting versions	FIFO	CDC
Nutstore	Automatic locking	Keep all the conflicting versions	-	Full-file & `rsync`

Table 3: A brief summary of the collaboration support of the eight mainstream cloud storage services in our study. "$*$": Seafile only supports automatic locking for Microsoft Office files. "-": we do not observe obvious queuing behavior.

2.2 Results and Findings

Our study quantifies the occurrence of conflicts in different cloud storage services, and uncovers their key design principles as summarized in Table 3.

Occurrence probability of conflicts. When the ten traces are replayed with each cloud storage service, we find considerable difference (ranging from 0 to 4.8%) in the *ratio* of conflicting file versions (generated during a replay) over all versions, as shown in Table 4. Most notably, Google Drive appears to have never generated conflicts, because once it detects conflicting versions of a file (at the cloud) it only keeps the latest version based on their server-side timestamps. In contrast, the most conflicting versions are generated with iCloud Drive, because its *sync delay* (*i.e.,* the delivery time of a file update) is generally longer than that of the other cloud storage services (as later indicated in Figure 3 and Table 5). In comparison, for each trace Nutstore generates the fewest conflicting versions (with Google Drive not considered), as its automatic locking during collaboration can avoid a portion (7.6%–19.1%) of conflicts.

Locks. We observe that the majority of the studied cloud storage services (Dropbox, OneDrive, Google Drive, iCloud Drive, and SugarSync) never use any form of locks for files being edited. As a consequence, collaboration using these products can easily lead to conflicts. Box, Seafile, and Nutstore use coarse-grained file-level locks; unfortunately, we find that their use of locks is either too early or too late[2], leading to undesired experiences. This is because cloud storage services are unable to acquire users' real-time editing behaviors and thus cannot accurately determine when to request/release locks. Specifically, locking too early leads to Pattern 4 in Table 1, locking too late (locking after editing) leads to Pattern 1, and unlocking too early leads to Pattern 2.

Box only supports manual locks on shared files. When Alice attempts to lock a shared file f and Bob has not opened it, f is successfully locked by Alice and then Bob cannot edit it (until it is manually unlocked by Alice). However, if Bob has already opened f when Alice attempts to lock it, he can still edit it but cannot save it, because when Bob attempts to save his edit the file editor (*e.g.,* MS Word) will re-check the permission of f. *In essence, Box implements locks by creating a process on Bob's PC, which attempts to "lock" a file by changing the file's permission as read-only.* In this case, if Bob is using an *exclusive* editor (not allowing other applications to write the file it opened), Alice's edits cannot be synced to Bob, thus leading to Pattern 3; otherwise, Bob's edits will be overwritten by Alice's, leading to Pattern 1.

Seafile automatically locks a shared file f when f is opened by an MS Office application, and f will not be unlocked until it is closed. This locking mechanism is coarse-grained and may lead to Pattern 4. For non-MS Office files, Seafile supports manual locks in the same way as Box, and thus they have the same issue in collaboration.

Nutstore attempts to lock a shared file f automatically, when Alice saves her edit. At this time, if Bob has not opened f, f is successfully locked by Alice and Bob cannot edit it; after Alice's saved edit is propagated to Bob, f is automatically unlocked. However, if Bob opened the shared file before Alice saves the file, Nutstore has the same problems as Box and Seafile (Patterns 1 and 3 in Table 1).

Finally, we are concerned with the delivery time of a *lock status* (*i.e.,* whether a file is locked). According to our measurements, the lock status is delivered in real time with $\sim$100% success rates. As in Figure 2, the delivery time ranges from 0.7 to 1.6 seconds, averaging at 1.0 second. This indicates that today's cloud storage services implement dedicated infrastructure (*e.g.,* queues) for managing locks.

In summary, implementing desirable locks in cloud storage services is not only complex and difficult but also somewhat expensive. Therefore, we feel it wiser to give up using locks.

Conflict resolution. We find three different strategies for resolving the conflicts. *First*, Google Drive only keeps the latest version (defined by the timestamp each version arrives at the cloud). All the older versions are discarded and can hardly be recovered by the users (Google Drive does not reserve a version history for any file). Note that this notion of "latest" may not reflect the absolute latest (which depends on the client-side time), *e.g.,* when the real latest version

[2]Ideally, a file should be locked right before the user starts editing, and unlocked right after the user finishes the editing.

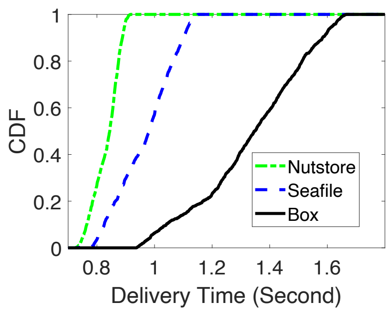

Figure 2: CDF of the delivery time of a lock status. Note that among all the studied services, only three of them use locks.

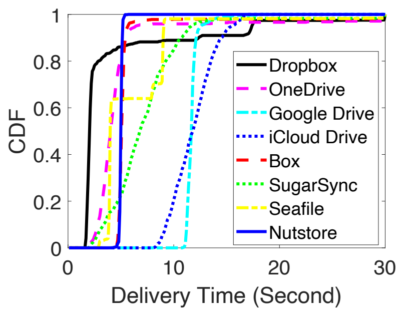

Figure 3: CDF of the delivery time of a file update, where the file is several KBs in size.

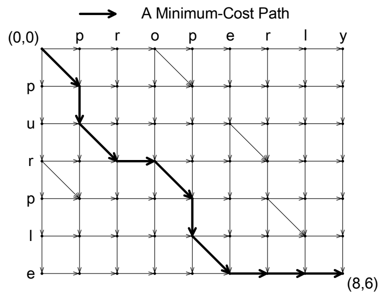

Figure 4: A simple edit graph for reconciling V_0 (the horizontal word "properly") and V_1 (the vertical word "purple").

Trace	DB	OD	GD	ID	Box	SS	SF	NS
DB1	4.4%	4.4%	0	4.5%	4.3%	4.3%	4.3%	3.6%
DB2	4.7%	4.7%	0	4.8%	4.6%	4.7%	4.6%	3.8%
OD	4.1%	4.1%	0	4.2%	4.0%	4.0%	4.1%	3.5%
ID	4.1%	4.0%	0	4.1%	4.1%	4.1%	4.1%	3.4%
Box	4.3%	4.3%	0	4.4%	4.2%	4.3%	4.3%	3.7%
SS	4.2%	4.1%	0	4.2%	4.2%	4.1%	4.2%	3.7%
SF	4.5%	4.5%	0	4.6%	4.5%	4.5%	4.5%	3.8%
SG	1.3%	1.3%	0	1.3%	1.3%	1.3%	1.3%	1.2%
TG	3.5%	3.5%	0	3.5%	3.5%	3.5%	3.5%	3.2%
LG	4.0%	4.0%	0	4.0%	4.0%	4.0%	4.0%	4.0%

Table 4: Ratio of conflicting file versions (over all versions) when the ten traces are replayed with each of the studied cloud storage services. DB=Dropbox, OD=OneDrive, GD=Google Drive, ID=iCloud Drive, SS=SugarSync, SF=Seafile, NS=Nutstore, SG=Spark-Git, TG=TensorFlow-Git, and LG=Linux-Git.

Cloud Service	Min	Median	Mean	P99	Max
Dropbox	1.6	2.0	141.2	**312**	**17751**
OneDrive	1.6	4.0	33.4	**106**	**4415**
Google Drive	10.9	11.7	11.7	12.9	18.1
iCloud Drive	8.1	11.8	11.9	11.9	16.9
Box	4.4	5.1	41.8	**115**	**6975**
SugarSync	2.0	6.8	51.3	**124**	**7094**
Seafile	2.7	4.0	53.8	**99**	**9646**
Nutstore	4.2	5.0	5.0	5.0	5.6

Table 5: Statistics (in unit of second) of the delivery time of a file update, where the file is several KBs in size.

arrives earlier due to network latency. *Second*, iCloud Drive asks the user to choose one version from all the conflicting versions. The user has to compare them by hand, and then make a decision (which is often not ideal). *Third*, a more common solution is to keep all the conflicting versions in the shared folder, and disseminate them to all the collaborators. This solution is more conservative (which does not cause data loss), but leaves all burdens to users. Moreover, given the distributed nature, merging efforts from the collaborators could cause further conflicts if not coordinated well.

Given the difficulties in resolving conflicts, we advocate that cloud storage services should make more effort to proactively avoid, or at least significantly reduce, the conflicts.

Delivery latency and message queue. Delivery latency of a file (update) prevalently exists in cloud storage at both infrastructure (*e.g.,* S3 and Azure Blob) and service (*e.g.,* Dropbox) levels [34, 35, 43, 64, 67, 74]. It stems from multiple factors such as network jitter, system I/O, and load balancing in the datacenter [43, 50]. We measure the delivery time of a file update regarding the eight cloud storage services. As in Figure 3 and Table 5, some services always have reasonable delivery time. On the other hand, in a few services, the maximum delivery time reaches several hours for a KB-level file, and the 99-percentile (P99) delivery time can reach hundreds of seconds. The unpredictability and long tail latency can sometimes break the time order among file updates, which is the main root cause of Patterns 2 and 3.

Additionally, we find that the implementation of message queues in some cloud storage services aggravates the delivery latency. Specifically, different services have very different message queue implementations, leading to different queueing behaviors. For a FIFO queue (used by Seafile), when the server is overloaded, many requests for file/fetch updates are processed by the server but not accepted by the client due to client-side timeout, thus wasting the server's processing resources. This problem can be mitigated by using LIFO queues (used by Dropbox). However, for a LIFO queue, the requests from "unlucky" users (who encounter the server's being overloaded after issuing fetch update requests) wait for a long duration. We suspect that the services with excessively long delivery time are using big shared queues with no QoS consideration, and may benefit from using a dedicated queue like QJUMP [41].

File update methods. Collaboration results in frequent, short edits to files. *Delta sync* is known to be efficient in updating short edits, compared with *full-file sync* where the whole file has to be transferred [49]. To understand the file update method, we let Alice modify a Z-byte highly compressed file, where $Z \in \{1, 1K, 10K, 100K, 1M\}$, and observed the traffic usage in delivering the file update. By comparing the traffic usages in uploading and downloading an update, we find that OneDrive, Google Drive, and Box adopt full-file sync, and the others adopt delta sync (`rsync` [72] or CDC [59]). Especially, we confirm Seafile's adoption of CDC from its source code [17]. In terms of Nutstore, it adopts a hybrid file update method: full-file sync for small ($\leq$64 KB) files and delta sync for the other files, so as to achieve the highest update speed, because small and large files are more suitable for full-file and delta sync, respectively (full-file sync requires fewer rounds of client-server message exchanges).

2.3 Implications

Our study results show that today's cloud storage services either do not use any locks or use coarse-grained file-level locks to prevent conflicts. The former would inevitably lead to conflicts. The latter, however, is hard to prevent conflicts in practice for two reasons: 1) it is hard to accurately predict user's editing behaviors in real time and therefore to determine the timing of applying the lock, and 2) the latency between the client and the server can vary significantly, so file-level conflicts are generally inevitable. Furthermore, the study shows that full-file and delta sync methods can be combined to accelerate the delivery of a file update. To address the revealed issues, we explore the possibility of developing lock-free conflict resolution by inferring fine-grained user intentions. We also explore a hybrid design of full-file and delta sync methods for efficient file update and synchronization.

3 Our Solution

This section aims to address the challenges uncovered in §2. Our key idea is to model file editing events as `insert` or `delete` *operations* (§3.2). Based on the operation model, we *infer* the collaborators' operation sequences (§3.3), and then *transform* these sequences (§3.4) based on their conflicts to generate the final version. We explain the above procedure with a simple case of two file versions, and demonstrate its applicability to the complex case of multiple versions (§3.5). We also design optimizations to the maintenance of shared files' historic versions (§3.6),

3.1 True and False Conflicts

We examine the conflicting file versions as listed in Table 4 in great detail. We find that ~1/3 of them come from non-text (*e.g.,* PDF or EXE) files, which, as mentioned in §1, are typically generated based on text files and thus can be simply deleted or regenerated from text files for pretty easy conflict resolution. The remainder relate to text files, the vast majority of which, to our surprise, only contain "false positive" conflicts as the collaborators in fact operated on different parts of a shared file.

Take the Dropbox-1 collaboration trace as an example. When it is replayed with Dropbox or OneDrive, among the 3,527 file versions hosted at the cloud side, 154 text files are considered (by Dropbox and OneDrive) to be conflicting versions and then distributed to all the collaborators. Actually, 152 out of the 154 *apparently* conflicting versions can be correctly merged at the cloud side. The remaining two cannot be correctly merged as two collaborators happen to edit the same part of the shared file in parallel, thus generating 9 *true* conflicts. In other words, the vast majority of the (coarse-grained) file-level conflicts are *false* (positive) conflicts when seen at the (fine-grained) operation level.

3.2 Explicit and Implicit Operations

We model *operation* as the basic unit in collaborative file editing. A shared file can be regarded as a sequence of characters, and an *explicit* operation is a user action that has truly occurred to the shared file, modifying some of its characters. In detail, an explicit operation O consists of seven properties:

- There are two possible *operation types*: `insert` and `delete`; $O.type$ represents the operation type of O.
- The *targeted string* is the string that will be inserted or deleted by O, which is denoted by $O.str$.
- The *length* of O is the (character) length of $O.str$, which is denoted by $O.len$.
- The *position* of O is where $O.str$ will be inserted to or deleted from in the shared file, which is denoted by $O.pos$.
- O must be performed on a context (file version), which is called the *base context* of O, or denoted as $O.bc$.
- O is performed on $O.bc$ to generate a new context, which is called the *result context* of O, or denoted as $O.rc$.
- The range of characters impacted by O in $O.bc$ is the *impact region* of O, denoted as $O.ir$. It is calculated as:

$$O.ir = \begin{cases} [O.pos, O.pos+1) & \text{if } O.type = \texttt{insert}; \\ [O.pos, O.pos+O.len) & \text{if } O.type = \texttt{delete}. \end{cases}$$

This formula tells that when a string is inserted to $O.bc$, the insert operation only affects the position (in $O.bc$) where the string is inserted; but when a string is deleted from $O.bc$, the positions where all the characters of the string formerly appear at $O.bc$ are affected.

Automatically acquiring a user's explicit operations is trivial and lightweight when the editor can be monitored, *e.g.,* in Google Docs [7] and Overleaf [15]. In these systems, users are required to use a designated online file editor, by monitoring which all the collaborators' explicit operations can be directly captured in real time.

In contrast, our studied cloud storage services are supposed to work independently with any editors and support any types of text files, thus bringing great convenience to their users (especially non-technical users). Therefore, we do not attempt to monitor any editors or impose any restrictions on the file types, and thus cloud storage services cannot capture users' explicit operations. Instead, we choose to analyze users' *implicit operations* based on the numerous file versions hosted at the cloud side. For a shared file f, implicit operations represent the cloud-perceived content changes to f (*i.e.,* the eventual result of a user's editing actions), rather than the user's editing actions that have actually happened to f. Obviously, implicit operations, as well as their various properties, have to be indirectly *inferred* from the different versions of f. Since we focus on implicit operations in this work, we simply use "operations" to denote "implicit operations" hereafter.

3.3 Operation Inference (OI)

When no conflict happens, inferring the operations from two consecutive versions of a file is intuitive, so in this part we only consider the OI when two conflicting versions emerge at the cloud. Note that when there are more than two conflicting versions, our described algorithms below still apply.

When two conflicting versions of a file, V_1 and V_2 (of n_1 and n_2 bytes in length) are uploaded to the cloud by two collaborators, the cloud first pinpoints their latest *common-context* version V_0 (of n_0 bytes in length) hosted in the cloud. Generally, the cloud knows which version is consistent with a collaborator's local copy during her last connection to the cloud. When the collaborator uploads a new version, this "consistent" version is regarded as the base context (version) of the new version, so that all versions of a shared file constitute a *version tree*, in which the parent of a version is its base context. Therefore, to pinpoint V_0 is to find the latest common ancestor of V_1 and V_2 in the version tree.

After pinpointing V_0, the cloud starts to infer the operation sequences (S_1 and S_2) that change V_0 to V_1 and V_2, respectively. To infer S_1, the common approach is to first find the longest common subsequence (LCS) between V_0 and V_1 using *dynamic programming* [33, 44, 57]. Then, by comparing the characters in V_0 and the LCS one by one, a sequence of `delete` operations can be acquired, which changes V_0 to the LCS; in a similar manner, a sequence of `insert` operations that changes the LCS to V_1 can be acquired. After that, the acquired `delete` and `insert` operations are combined to constitute S_1 (S_2 is constituted in a similar manner). Unfortunately, this common approach requires $O(n_0 * n_1)$ computation complexity, which may require considerable time for a large file, *e.g.,* ~30 seconds for a 500-KB file.

To address this problem, we leverage an *edit graph* [4, 55] to organize V_0 and V_1. Figure 4 exemplifies how to calculate the LCS between two words "properly" (V_0, on the horizontal axis) and "purple" (V_1, on the vertical axis) using an edit graph, where a diagonal edge has weight 0 and a horizontal or vertical edge has weight 1. Accordingly, finding the LCS between V_0 and V_1 is converted to finding a minimum-cost path that goes from the start point (*i.e.,* $(0,0)$ in Figure 4) to the end point (*i.e.,* $(8,6)$ in Figure 4). With an edit graph, the problem can be solved with $O((n_0 + n_1) * d)$ complexity [55], where $d = n_0 + n_1 - 2l$ is the number of horizontal and vertical edges (*i.e.,* the length of difference between V_0 and V_1) and l is the number of diagonal edges (*i.e.,* the length of the LCS). Note that d is usually much smaller than n_0 and n_1 in practice: in our collected traces, the median and mean values of $\frac{d}{n_0+n_1}$ are merely 0.12% and 2.19%. Thus, the cloud can infer S_1 and S_2 efficiently using the edit graph, *e.g.,* for a 500-KB file the inference time is typically optimized from ~30 seconds to ~200 ms, resulting in a 150× reduction.

3.4 Operational Transformation (OT)

After the operation sequences S_1 and S_2 are inferred, which contain s_1 and s_2 operations respectively (all operations in a sequence are sorted by their position and have the same base context V_0), the cloud first detects whether there exist true conflicts, and then constructs a *conflict graph* [53] (as shown in Figure 5) if there are any. A conflict graph is a directed acyclic graph that has $s_1 + s_2$ vertices representing the aforementioned $s_1 + s_2$ operations. After that, *operation transformation* (OT) [39] is adopted to transform and merge S_1 and S_2 into a result sequence S_r, which can be executed on V_0 to generate the merged file version $V_{1,2}$.

Detecting true conflicts. In order to detect true conflicts between S_1 and S_2, the cloud first merges S_1 and S_2 into a temporary sequence S_{temp} sorted by the operations' position, and initializes the conflict graph G with $s_1 + s_2$ vertices and 0 edges. Then, for each operation in S_{temp}, the cloud checks whether the operations behind it conflict with it – this is achieved by checking whether the impact regions of two operations overlap each other. If two operations $S_{temp}[i]$ and $S_{temp}[j]$ are real conflicting operations, an edge $e_{i,j}$ connecting v_i to v_j (denoted by solid arrows in Figures 5a and 5b) is added to G to represent a true conflict. If there are no true conflicts between any two operations, G is useless and simply discarded. The detection, in the worst case (where each operation in S_1 conflicts with each operation in S_2), bears $O((s_1 + s_2)^2)$ complexity. However, in common cases there exist only a few conflicts, and thus the detection can be quickly carried out with $O(s_1 + s_2)$ complexity.

Basics of OT. As the *de facto* technique for conflict resolution in distributed collaboration, OT [39] has been well studied [40, 61] and used (*e.g.,* Google Docs [7], Overleaf [15], Wave [24], and Etherpad [6]). It resolves conflicts by transforming parallel operations on a shared file to equivalent sequential operations (if possible). A very simple example of OT is shown in Figure 6. More details and examples of OT can be found at https://UFC2.github.io

OT when there are no true conflicts. According to our detection results on the ten collaboration traces (cf. Table 2),

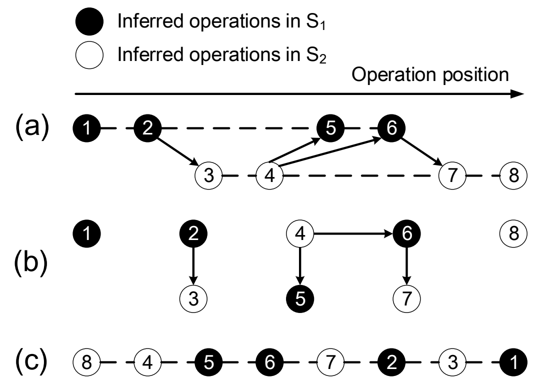

Figure 5: Reordering conflicting operations with a conflict graph. (a) In the two operation sequences S_1 and S_2, a dashed line denotes a sequence, while a solid arrow represents a true conflict. (b) S_1 and S_2 are reorganized into a conflict graph, where conflicting operations are linked with directed edges. (c) In the result sequence S_r, operations are sorted by their topological order in the conflict graph.

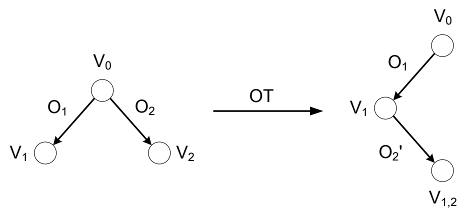

Figure 6: An example of OT that merges V_1 and V_2, in which O_2 is transformed to O_2' to resolve the conflict between O_1 and O_2.

when a file-level conflict occurs there are no true conflicts with a very high (>95%) probability, which is consistent with the results of our manual examination in §3.1. When there are no true conflicts detected, the cloud directly applies OT on S_1 and S_2 to generate S_r and $V_{1,2}$. Traditionally, the computation complexity of OT is deemed as $O((s_1+s_2)^2)$. In our case, since there are no true conflicts and S_{temp} are already sorted by the operations' position, we choose to transform the operations in S_{temp} in their descending order of position, thus achieving a much lower complexity of $O(s_1+s_2)$. After the transformation, we get S_r and execute S_r on V_0 to generate the merged version.

OT in the presence of true conflicts. If there are true conflicts detected, it is impossible to directly and correctly resolve the conflicts as in the above case. Consequently, we choose to prioritize the mitigation of user intervention while preserving potentially useful information, so as to facilitate users' manual conflict resolution. Specifically, two principles should be followed: 1) the cloud should send only one merged version $V_{1,2}$ to the collaborators for easy manual conflict resolution; and 2) users' editing intentions should be retained as much as possible, while the number of conflicts that have to be manually resolved in $V_{1,2}$ had better be minimized.

To realize the two principles, our first step is to utilize *topological sorting* [46] to reorganize and help transform S_1 and S_2 (via their conflict graph G) following two rules. First, real conflicting operations should be transformed and put into S_r in the ascending order of their position, so that their conflicts can be resolved at one time and thus do not negatively impact the transformation of other operations. Second, non-conflicting operations should be put into S_r in the descending order of their position, so that they can be quickly transformed like in the case of no true conflicts.

After S_1 and S_2 are topologically sorted and put into S_r (see Figure 5c), we apply our customized OT scheme to embody the aforementioned two principles for resolving true conflicts. First of all, we classify true conflicts into different categories that are suited to different processing strategies. Given two conflicting operations O_1 and O_2 working on the same base context (V_0), there seem to be four different categories of conflicts in the form of "$O_1.type/O_2.type$": 1) `delete/delete`, 2) `delete/insert`, 3) `insert/delete`, and 4) `insert/insert`. Here "/" means $O_1.pos \leq O_2.pos$. However, by carefully examining the impact regions of O_1 and O_2 ($O_1.ir$ and $O_2.ir$) in each category, we find that `insert/delete` conflicts are never true conflicts, because an insert operation only affects the targeted string at the position it appears, and never affects a to-be-deleted string that starts behind this position. Thus, we only need to deal with the other three categories as follows.

- For a `delete/delete` conflict, all the characters deleted by the users (say, Alice and Bob) are $O_1.str \cup O_2.str$. To retain both users' editing intentions as much as possible, we choose to delete only the characters both users want to delete (*i.e.,* $O_1.str \cap O_2.str$), while preserving the other characters with related information. For example, let $V_0 =$ "We need foods, water, clothes, and books."; O_1 made by Alice is to `delete` "foods, water, " at position 8, whereas O_2 made by Bob is to `delete` "water, clothes, " at position 15. In this case, O_1 is transformed to `insert` "[Alice delete:foods,]" at position 8, and O_2 is transformed to `insert` "[Bob delete:clothes,]" at position 30 (= 8+ the length of "[Alice delete:foods,]"). After the two transformed operations are executed on V_0, the merged version $V_{1,2}$ is "We need [Alice delete:foods,][Bob delete:clothes,]and books." This is not a perfect result, but is pretty easy to be manually resolved by Alice and Bob.
- For a `delete/insert` conflict, we notice that the characters deleted by Alice might be the literal context of the characters inserted by Bob. Thus, the deleted characters should be preserved to facilitate (mostly Bob's) manual conflict resolution. For example, let $V_0 =$ "There is a cat in the courtyard."; O_1 is to `delete` " in the courtyard" at position 14, changing V_0 to V_1 ("There is a cat."), whereas O_2 is to `insert` "spacious " at position 22, changing V_0 to V_2 ("There is a cat in the spacious courtyard."). Without appropriate transformation, the merged version is "There is a catspacious .", which is obviously confusing. In this

Trace	# File Versions	# Conflicting Versions	# MV Conflicts	# Conflicts	# True Conflicts	Reduction of Conflicts
Dropbox-1	3527	154	8	501	9	98.2%
Dropbox-2	2193	104	12	257	5	98.1%
OneDrive	2673	109	10	284	7	97.5%
iCloud Drive	3211	133	9	402	8	98.0%
Box	2930	125	5	374	8	97.9%
SugarSync	3472	147	13	523	11	97.9%
Seafile	2823	126	11	411	9	97.8%
Spark-Github	129957	1728	133	6724	167	97.5%
TensorFlow-Github	246016	8621	845	66231	1097	98.3%
Linux-Github	901167	36048	3210	216584	2882	98.7%

Table 6: Measurement statistics when the ten collaboration traces are replayed with UFC2. "MV Conflicts" denote the conflicts of multiple versions, *i.e.,* ≥ 3 conflicting versions are generated from the same base version.

case, O_1 is split into two operations: one is to `insert` "[Alice delete: in the]" at position 14, and the other is to `insert` "[Alice delete: courtyard]" at position 37 (= 14+ the length of "[Alice delete: in the]"); and O_2 is transformed to `insert` "[Bob insert: spacious]" at position 37. Afterwards, $V_{1,2}$ is "There is a cat[Alice delete: in the][Bob insert: spacious][Alice delete: courtyard].", which is also imperfect but easy to be manually resolved.

- For an `insert/insert` conflict, except when $O_1.str = O_2.str$ (which rarely happens), we choose to preserve both $O_1.str$ and $O_2.str$ by inserting $O_2.str$ after $O_1.str$, meanwhile adding the related information. For example, let $V_0 =$ "We need foods and books." O_1 is to `insert` ", water," at position 13, whereas O_2 is to `insert` ", clothes," at the same position. In this case, $V_{1,2}$ is "We need foods, [Alice insert:water][Bob insert:clothes], and books."

3.5 Merging Conflicts of Multiple Versions

Our above-designed scheme, despite being described with a simple case of two versions, is also applicable to solving conflicts between multiple versions. Multi-version conflicts do not often happen in practice, *e.g.,* we can calculate from Table 6 that they only account for 9% of the total conflicts.

In this complex case, suppose multiple collaborators (say $n \geq 3$) simultaneously edit the same base version V_0 and then generate n conflicting versions V_1, V_2, V_3, ... , V_n. To resolve such conflicts, we first figure out the operation sequences (*i.e.,* S_1, S_2, S_3, ..., S_n) for each version using edit graphs, which represent the changes in each version relative to their common base version V_0. Afterwards, with our devised operation transformation method, all the operation sequences are merged one by one, so as to generate the result operation sequence $S_{r_{1,2,3,...,n}}$. Specifically, S_1 and S_2 are first merged to generate $S_{r_{1,2}}$, and then S_3 are merged with $S_{r_{1,2}}$ to generate $S_{r_{1,2,3}}$. This procedure is repeated until all the operation sequences are merged, resulting in $S_{r_{1,2,3,...,n}}$. Finally, similar to the simple case of two versions , $S_{r_{1,2,3,...,n}}$ is executed on V_0 to generate the final version $V_{1,2,3,...,n}$.

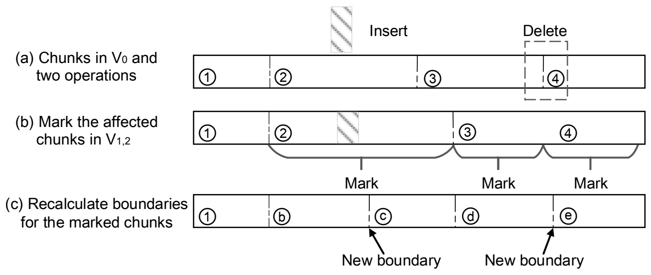
Insert
Delete
(a) Chunks in V0 and two operations
(b) Mark the affected chunks in V1,2
(c) Recalculate boundaries for the marked chunks
①
②
③
④
Mark
Mark
Mark
ⓑ
ⓒ
ⓓ
ⓔ
New boundary
New boundary

Figure 7: Boundary recalculation in OCDC. Chunk ② is split into ⓑ and ⓒ as its size exceeds the size limitation of a single chunk after the characters are added. Chunks ③ and ④ are re-partitioned as ⓓ and ⓔ as the total sizes of their remaining parts exceed the size limitation of a single chunk (otherwise, they will be combined).

3.6 Maintenance of Historic Versions

The merged version $V_{1,2}$ of a shared file, as well as the previous versions, should be kept in the cloud so that 1) users can retrieve any previous versions as they wish, and 2) the cloud can pinpoint V_0 from historic versions in future conflict resolutions. To save the storage space for hosting historic versions, we break each version into variable-sized data chunks using CDC [59] for effective chunk-level deduplication.

For a user-uploaded file version, guided by the findings in §2.2, we adopt full-file sync for small ($\leq$64 KB) files and CDC delta sync for larger files to achieve the (expected) shortest upload time. Here we adopt CDC delta sync rather than the more fine-grained `rsync` to make our delta sync strategy compatible with the aforementioned CDC-based version data organization. In other words, we allow a little extra network traffic to save expensive computation cost.

For a server-merged version $V_{1,2}$, we exploit the implicit operations inferred during the aforementioned conflict resolution to accelerate CDC, which is referred to as *operation-based CDC* (OCDC). Specifically, for each operation in the result sequence S_r, we examine whether its impact region overlaps the boundaries of any chunks of V_0 (see Figure 7 (a)); if yes, we mark the boundary (or boundaries) as "changed" (see Figure 7 (b)). After examining all operations in S_r, we use the unchanged boundaries to split $V_{1,2}$ into multiple parts, and recalculate the block boundaries of those parts that contain

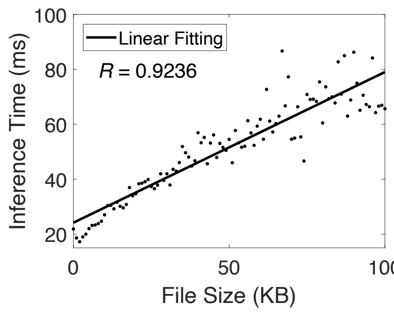

Figure 8: Time overhead incurred by our devised operation inference. Here R is the correlation coefficient between the measurements and linear fitting.

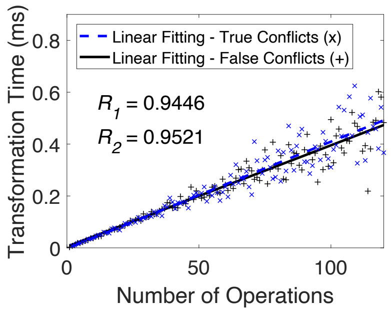

Figure 9: Time overhead incurred by our devised operation transformation. Here R_1 and R_2 are the correlation coefficients with and without true conflicts.

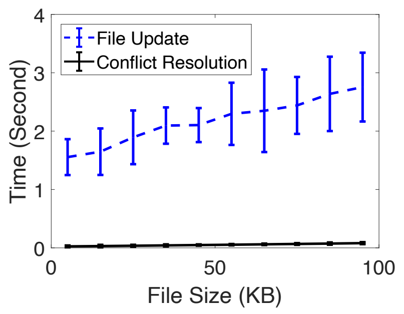

Figure 10: Total time overhead of a conflict resolution *vs.* the delivery time of a file update (using the hybrid full-file/delta sync method).

"changed" boundaries (see Figure 7 (c)). OCDC is especially effective when there is only small difference between V_1/V_2 and V_0 (which is the usual case in practice).

4 Implementation and Evaluation

To implement our design, we build a prototype system UFC2 (User-Friendly Collaborative Cloud) on top of Amazon Web Services (AWS) with 5,000 lines of Python code, and evaluate UFC2 using real-world workloads in multiple aspects.

4.1 Implementation

At the infrastructure level of UFC2, we host the (hierarchical) metadata of historic versions in Amazon EFS for efficient file system access, and the (flat) data chunks in Amazon S3 for economic content storage – note that the unit storage price of EFS (~\$0.3/GB/month) is around 10× higher than that of S3 [38]. Besides, the web service of UFC2 runs on a standard VM (with a dual-core CPU @2.5 GHz, 8-GB memory, and 32-GB SSD storage) rented from Amazon EC2. Moreover, the employed EFS storage, S3 storage, and EC2 VM are located at the same data center in Northern Virginia, so there is no bottleneck among them. At the client side, we deploy puppet collaborators on geo-distributed VMs rented from Amazon EC2 to replay our collected ten real-world collaboration traces (cf. Table 2). Details of these VMs and the replay processes are the same as those described in §2.1.

4.2 Experiment Results

Ratio of conflicts resolved. Our first metric to evaluate the collaboration support of cloud storage services is the number of conflicts. We replay the ten traces with UFC2, and observe that the file versions generated by UFC2 (at the cloud side) are slightly different from those generated by Dropbox/OneDrive/iCloud Drive/Box/SugarSync/Seafile (cf. §2.2) due to the variation (*esp.,* in latency) of network environments; also, the resulting conflicts are slightly different. Notably, all the false conflicts are automatically resolved by UFC2. The remaining conflicts are all true conflicts that should be manually resolved by the collaborators, assisted with the helpful information automatically added by UFC2. As listed in Table 6, the ratio of conflicts is reduced by 97.5%–98.7% for different traces, *i.e.,* an average reduction by 98%.

Time overhead of conflict resolution. Conflict resolution in UFC2 consists of two steps: operation inference (OI, §3.3) and operation transformation (OT, §3.4). Thus, we first examine the time overhead incurred by the two steps separately, and then analyze the total time of conflict resolution (compared to the delivery time of a file update).

First, we record the time of OI in every conflict resolution when replaying the ten traces with UFC2. The results are plotted as a scatter diagram shown in Figure 8, together with a linear fitting. The correlation coefficient (R) between the measurements and the linear fitting results is as large as 0.9236, indicating that the time of OI is generally proportional to the file size. This is because by leveraging an edit graph, we reduce the computation complexity of OI from $O(n_0 * n_1)$ to $O((n_0 + n_1) * d)$ (refer to §3.3 for the details).

Second, we record the time of OT in every conflict resolution, and find it is very small (<1 ms) compared to the time of OI. As shown in Figure 9, the time of OT is highly proportional to the number of operations; in addition, the performance is quite similar with or without true conflicts. According to §3.4, the complexity of our devised OT is $O(s_1 + s_2)$, which explains the experiment results.

Further, we calculate the total time of a conflict resolution, and record the delivery time of the corresponding file update (using the hybrid full-file/delta sync method). As shown in Figure 10, the total time of a conflict resolution is 10–80 ms, while the delivery time of a file update is 1.5–3 seconds. The former is merely 0.6%–4% (on average 2%) of the latter, showing that our conflict resolution brings negligible performance overhead to the collaboration in cloud storage.

Time overhead of OCDC *vs.* traditional CDC. We record the time spent in breaking a merged file version into data

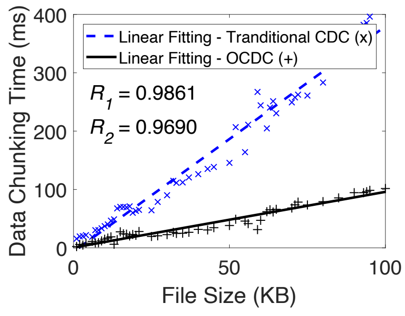

Figure 11: Data chunking time for a common file, using OCDC *vs.* traditional CDC.

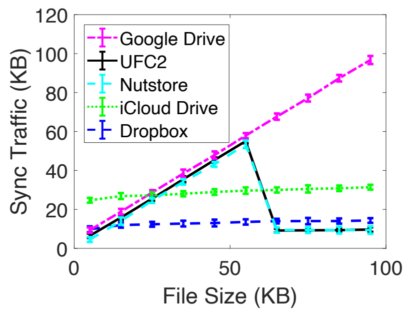

Figure 12: Sync traffic of UFC2 and representative cloud storage services for a file update when there are no file-level conflicts.

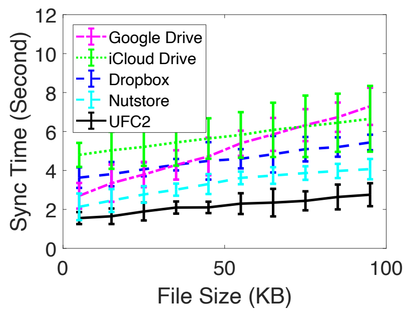

Figure 13: Sync time of UFC2 and representative cloud storage services for a file update when there are no file-level conflicts.

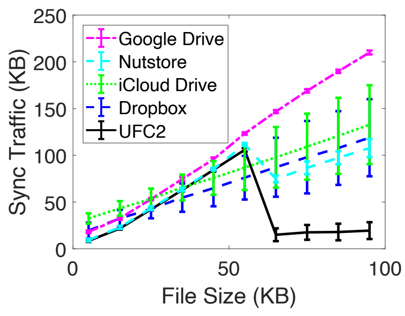

Figure 14: Sync traffic of UFC2 and representative cloud storage services for a file update when there exist file-level conflicts.

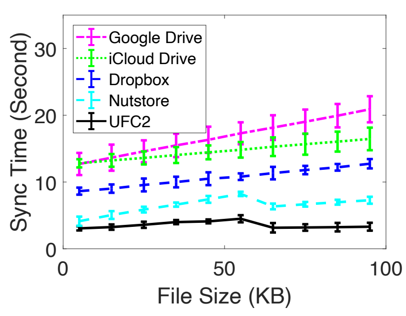

Figure 15: Sync time of UFC2 and representative cloud storage services for a file update when there exist file-level conflicts.

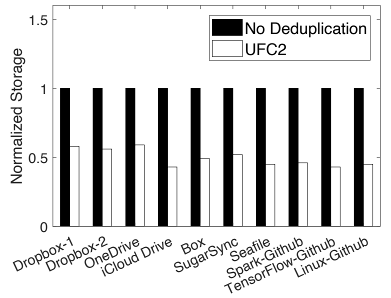

Figure 16: Normalized storage overhead of historic file versions for the ten real-world collaboration traces.

chunks with OCDC when replaying the ten traces with UFC2. For comparison, we also break the same merged file version into data chunks with traditional CDC.

As shown in Figure 11, for both OCDC and traditional CDC, the data chunking time is highly proportional to the file size. This is quite intuitive because a larger file is usually broken into more chunks. Additionally, we notice that OCDC outperforms traditional CDC by ∼3 times, reducing the data chunking time from 30–400 ms to 10–120 ms.

Network overhead. We compare the sync traffic of UFC2 with those of Dropbox, Google Drive, iCloud Drive, and Nutstore, for a file update. We only select the four cloud storage services since Dropbox, Google Drive, and iCloud Drive each represent a typical strategy for conflict resolution adopted by existing cloud storage services (*i.e.,* keep all conflicting versions, only keep the latest version, and force users to choose one version, cf. §2.2) while Nutstore is the only service that combines full-file sync and delta sync to enhance the file update speed.

As shown in Figure 12, when there are no file-level conflicts, the sync traffic of Google Drive is close to the file size, as Google Drive adopts full-file sync. In contrast, Dropbox and iCloud Drive always consume nearly 10 KB and 30 KB of sync traffic respectively due to their adoption of delta sync; we infer that the sync granularity of Dropbox is finer than that of iCloud Drive. In contrast, Nutstore and UFC2 resemble Google Drive for small (≤64 KB) files and Dropbox for larger files, as they both adopt full-file sync for small files and delta sync for larger files to achieve the shortest sync time (see Figure 13). This hybrid sync method results in substantial savings of sync traffic for Nutstore and UFC2 after the turning point (64 KB) in Figures 12 and 14.

As shown in Figure 14, when there exist file-level conflicts, the sync traffic of Google Drive is nearly twice of the file size. This is because (the client of) Google Drive first uploads the local version, and then downloads the cloud-hosted newer version to overwrite the local version. In contrast, the sync traffic consumed by Dropbox or iCloud Drive is close to the file size; this is because the client of Dropbox (or iCloud Drive) renames one of the conflicting versions, and the renamed one is uploaded as a newly-created file using full-file sync (which usually consumes more traffic than necessary since delta sync can still be applied).

The case of Nutstore in Figure 14 is a bit complex: for small files, its sync traffic is nearly twice of the file size (similar to Google Drive); for larger files, the traffic is slightly larger than the file size (similar to Dropbox/iCloud Drive). This is because Nutstore renames one of the conflicting versions when a file-level conflict occurs – if the file is small ($\leq$64 KB), the two files are both uploaded to the cloud using full-file sync; otherwise, the renamed file is uploaded using full-file sync (which usually consumes unnecessary traffic) whereas the original file is uploaded using delta sync.

Finally, we examine the case of UFC2 in Figure 14. Its client first uploads a conflicting version and then downloads the merged version from the cloud. For a small file, the two versions are both delivered using full-file sync, so the sync traffic is nearly twice of the file size; for a larger file, the two versions are both delivered using delta sync (which is more traffic-saving than what Nutstore does for a larger file), so the sync traffic is always as small as $\sim$20 KB. This is why UFC2 achieves the shortest sync time, as shown in Figure 15.

Storage Overhead. For the maintenance of a file's historic versions, the straightforward approach is to store all versions separately without data deduplication; its storage overhead is taken as the baseline and normalized as 1.0, as shown in Figure 16. Utilizing CDC-based deduplication, the storage overhead of UFC2 is normalized between 0.43 and 0.59 (0.49 on average) with respect to the ten traces. In comparison, the storage overhead of Google Drive is normalized as small as 0.05–0.1, because Google Drive only stores the latest version and discards all previous versions. We do not quantify the storage overhead of the other mainstream cloud storage services since we do not know their cloud-side storage organization.

5 Related Work

Various schemes have been proposed to address the collaboration conflicts in distributed file systems (DFS) and version control systems (VCSes). In this section, we survey the typical schemes and compare them to our design choices.

Conflict resolution in DFSes. LOCUS [73], Coda [47] and InterMezzo [32] mark files with unresolved conflicts as inconsistent, so that these files are inaccessible until users manually rename and merge them. These schemes prevent users from accessing the conflicting files before conflicts are resolved, and the idea of restrictive access is inherited by some recent cloud-backed file systems such as SCFS [30].

In contrast, Ficus [60] and Rumor [42] attempt to design specific conflict resolvers (using semantic knowledge of certain file types or user-made rules), so as to automatically merge conflicts of specific kinds. Bayou [69] preserves all conflicting files and allows users to access them. Similar approaches are adopted by recent large-scale systems like Dynamo [36], TierStore [37], Depot [52], and COPS [51], where all conflicting file versions are preserved, and users are forced to manually resolve all file-level conflicts. In fact, the above described strategies are also adopted (in part) by our studied popular cloud storage services.

Our work essentially differs from the aforementioned schemes by providing not only effective but also transparent and user-friendly collaboration support for replicated files in distributed environments. The desired features are enabled by our novel perspective and intelligent technical approaches in addressing the concurrent conflicts.

Conflict resolution in VCSes. Popular VCSes, such as SVN, CVS, Git, RCS [71], and SunPro [26], generally operates at a (text) line level. To resolve the conflicts between two versions of a shared file, they use delta algorithms like `bdiff` [70] and UNIX `diff` [45] to find the modified lines, which are then simply combined to form a merged version. However, if two users' modifications are made on the same line, they have to manually pick which line to retain. Recently, a more advanced approach called *structured merge* [27,28,48,75] has emerged in the software engineering community, which takes the syntactic structure of a program into account and thus can resolve very detailed conflicts happening to non-essential elements (*e.g.,* comments, tabs, and blanks) of a program. Different from VCSes' line-level or syntactic approaches that is mostly designed for developers, our work studies conflict resolution for general-purpose cloud storage services designed for regular end users.

6 Conclusion

Despite a rich body of techniques for resolving conflicts in collaborative systems [29,40,56,65,66], today's mainstream cloud storage services still use the simplest form, *i.e.,* coarse-grained file-level conflict detection and resolution. Given that collaboration has become a major use case of cloud storage services, existing mechanisms, as revealed in this paper, are deficient, inconvenient, and sometimes frustrating.

To address the issue, we make a series of efforts towards understanding and improving collaboration in cloud storage services from a novel perspective of operations without using any locks. We find that the vast majority of conflicts reported by today's cloud storage services are false conflicts, and design intelligent approaches to efficient operation inference, user-friendly operation transformation, and judicious maintenance of historic versions. We implement all the approaches in an open-source prototype system that can significantly reduce collaboration conflicts and meanwhile preserve the transparency and user-friendliness of cloud storage services.

Acknowledgements

We thank our shepherd, Geoff Kuenning, and the anonymous reviewers for their valuable feedback and suggestions. Also, we thank Liangyi Gong for his help in typesetting, and Fengmin Zhu for his generous discussion. This work is supported in part by the National Key R&D Program of China under grant 2018YFB1004700, the National Natural Science Foundation of China (NSFC) under grants 61822205, 61632020 and 61632013, and the Beijing National Research Center for Information Science and Technology (BNRist).

References

[1] Average U.S. Internet Speeds More Than Double Global Average. https://www.ncta.com/whats-new/average-us-internet-speeds-more-double-global-average.

[2] Box – Secure File Sharing, Storage, and Collaboration. https://www.box.com/.

[3] Charles Web Debugging Proxy. https://www.charlesproxy.com/.

[4] Diff Match Patch is a High-performance Library in Multiple Languages that Manipulates Plain Text. https://github.com/google/diff-match-patch.

[5] Dropbox Tech Blog. https://blogs.dropbox.com/tech/.

[6] Etherpad: Really Real-time Collaborative Document Editing. https://github.com/ether/etherpad-lite.

[7] Google Docs: Free Online Documents for Personal Use. https://www.google.com/docs/about/.

[8] Google Drive privacy warning – could yours have leaked data? https://www.welivesecurity.com/2014/07/11/google-drive-privacy-warning/.

[9] Meet Bandaid, the Dropbox Service Proxy. https://blogs.dropbox.com/tech/2018/03/meet-bandaid-the-dropbox-service-proxy/.

[10] Never-Googlers: Web Users Take the Ultimate Step to Guard Their Datae. https://www.washingtonpost.com/technology/2019/07/23/never-googlers-web-users-take-ultimate-step-guard-their-data/.

[11] Nutstore – Share Your Files Anytime, Anywhere, with Any Device. https://www.jianguoyun.com/.

[12] Nutstore Help Center. http://help.jianguoyun.com/.

[13] Online Discussion – Those who refuse to use Google for privacy reasons. https://www.reddit.com/r/apple/comments/9ed07l/those_who_refuse_to_use_google_for_privacy/.

[14] Optimizing Web Servers for High Throughput and Low Latency. https://blogs.dropbox.com/tech/2017/09/optimizing-web-servers-for-high-throughput-and-low-latency/.

[15] Overleaf, Online LaTeX Editor. https://www.overleaf.com/.

[16] Seafile - Open Source File Sync and Share Software. https://www.seafile.com/en/home/.

[17] Seafile Source Code. https://github.com/haiwen/seafile.

[18] Simultaneous Collaborative Editing of a LaTeX File (Online Forum Discussion). https://tex.stackexchange.com/questions/27549/simultaneous-collaborative-editing-of-a-latex-file.

[19] Speedtest Global Index – Global Speeds August 2019. https://www.speedtest.net/global-index.

[20] SugarSync – Cloud File Sharing, File Sync & Online Backup From Any Device. https://www2.sugarsync.com/.

[21] SugarSync Help Center. https://support.sugarsync.com/hc/en-us/.

[22] The SugarSync Blog. https://www.sugarsync.com/blog/.

[23] Tool for the (collaborative) job. https://blogs.ams.org/phdplus/2016/09/11/tool-for-the-collaborative-job/.

[24] Wave | Real-time Collaboration and Coediting Service. https://www.codox.io/.

[25] Wireshark Network Protocol Analyzer. http://www.wireshark.org/.

[26] E. Adams, W. Gramlich, S. S. Muchnick, and S. Tirfing. SunPro: Engineering a Pratical Program Development Environment. In *Proceedings of International Workshop on Advanced Programming Environments*, pages 86–96. Springer-Verlag, 1986.

[27] S. Apel, O. Leßenich, and C. Lengauer. Structured Merge with Auto-tuning: Balancing Precision and Performance. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 120–129. ACM, 2012.

[28] T. Apiwattanapong, A. Orso, and M. J. Harrold. JDiff: A Differencing Technique and Tool for Object-oriented Programs. *Automated Software Engineering*, 14(1):3–36, 2007.

[29] H. Attiya, S. Burckhardt, A. Gotsman, A. Morrison, H. Yang, and M. Zawirski. Specification and Complexity of Collaborative Text Editing. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 259–268. ACM, 2016.

[30] A. Bessani, R. Mendes, T. Oliveira, et al. SCFS: A Shared Cloud-backed File System. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 169–180, 2014.

[31] R. Bhargava. Evolution of Dropbox's Edge Network, 2017. https://blogs.dropbox.com/tech/2017/06/evolution-of-dropboxs-edge-network/.

[32] P. Braam, M. Callahan, P. Schwan, et al. The InterMezzo File System. In *Proceedings of the 3rd of the Perl Conference, O'Reilly Open Source Convention*, 1999.

[33] G. Canfora, L. Cerulo, and M. Di Penta. Ldiff: An Enhanced Line Differencing Tool. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 595–598. IEEE Computer Society, 2009.

[34] Z. Chen, Q. He, Z. Mao, H.-M. Chung, and S. Maharjan. A Study on the Characteristics of Douyin Short Videos and Implications for Edge Caching. In *Proceedings of the ACM Turing Celebration Conference - China (TURC)*, page 13:1–13:6. ACM, 2019.

[35] Y. Cui, N. Dai, Z. Lai, M. Li, Z. Li, Y. Hu, K. Ren, and Y. Chen. Tailcutter: Wisely Cutting Tail Latency in Cloud CDNs under Cost Constraints. *IEEE/ACM Transactions on Networking*, 27(4):1612–1628, 2019.

[36] 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 *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, pages 205–220. ACM, 2007.

[37] M. Demmer, B. Du, and E. Brewer. TierStore: A Distributed Filesystem for Challenged Networks in Developing Regions. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 35–48. USENIX, 2008.

[38] J. E, Y. Cui, M. Ruan, Z. Li, and E. Zhai. HyCloud: Tweaking Hybrid Cloud Storage Services for Cost-efficient Filesystem Hosting. In *Proceedings of the IEEE Conference on Computer Communications (INFOCOM)*, pages 1–9. IEEE, 2019.

[39] C. Ellis and S. Gibbs. Concurrency Control in Groupware Systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 399–407. ACM, 1989.

[40] A. J. Feldman, W. P. Zeller, M. J. Freedman, and E. W. Felten. SPORC: Group Collaboration Using Untrusted Cloud Resources. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pages 337–350. USENIX, 2010.

[41] M. P. Grosvenor, M. Schwarzkopf, I. Gog, R. N. Watson, A. W. Moore, S. Hand, and J. Crowcroft. Queues Don't Matter When You Can JUMP Them! In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 1–14, 2015.

[42] R. G. Guy, P. L. Reiher, D. Ratner, M. Gunter, W. Ma, and G. J. Popek. Rumor: Mobile Data Access Through Optimistic Peer-to-peer Replication. In *Advances in Database Technologies*, pages 254–265. Springer, 1999.

[43] B. Hou and F. Chen. GDS-LC: A Latency-and Cost-aware Client Caching Scheme for Cloud Storage. *ACM Transactions on Storage*, 13(4):40, 2017.

[44] J. J. Hunt, K.-P. Vo, and W. F. Tichy. Delta Algorithms: An Empirical Analysis. *ACM Transactions on Software Engineering and Methodology*, 7(2):192–214, 1998.

[45] J. W. Hunt and M. D. MacIlroy. *An Algorithm for Differential File Comparison*. Bell Laboratories Murray Hill, 1976.

[46] A. B. Kahn. Topological Sorting of Large Networks. *Communications of the ACM*, 5(11):558–562, Nov. 1962.

[47] J. J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 213–225. ACM, 1991.

[48] O. Leßenich, S. Apel, C. Kästner, G. Seibt, and J. Siegmund. Renaming and Shifted Code in Structured Merging: Looking Ahead for Precision and Performance. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 543–553. IEEE, 2017.

[49] Z. Li, C. Jin, T. Xu, C. Wilson, Y. Liu, L. Cheng, Y. Liu, Y. Dai, and Z.-L. Zhang. Towards Network-level Efficiency for Cloud Storage Services. In *Proceedings of the Conference on Internet Measurement Conference (IMC)*, pages 115–128. ACM, 2014.

[50] G. Liang and U. C. Kozat. Fast Cloud: Pushing the Envelope on Delay Performance of Cloud Storage with Coding. *IEEE/ACM Transactions on Networking*, 22(6):2012–2025, 2014.

[51] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't Settle for Eventual: Scalable Causal Consistency for Wide-area Storage with COPS. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 401–416. ACM, 2011.

[52] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. Depot: Cloud Storage with

Minimal Trust. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pages 307–322. USENIX, 2010.

[53] D. Marx. Graph Colouring Problems and Their Applications in Scheduling. *Periodica Polytechnica Electrical Engineering (Archives)*, 48(1-2):11–16, 2004.

[54] T. Mens. A State-of-the-art Survey on Software Merging. *IEEE Transactions on Software Engineering*, 28(5):449–462, 2002.

[55] E. W. Myers. An *O(ND)* Difference Algorithm and Its Variations. *Algorithmica*, 1(1):251–266, Nov. 1986.

[56] D. A. Nichols, P. Curtis, M. Dixon, and J. Lamping. High-latency, Low-bandwidth Windowing in the Jupiter Collaboration System. In *Proceedings of the Annual ACM Symposium on User Interface and Software Technology (UIST)*, pages 111–120. ACM, 1995.

[57] Y. S. Nugroho, H. Hata, and K. Matsumoto. How Different Are Different Diff Algorithms in Git? *Empirical Software Engineering*, pages 1–34, 2019.

[58] S. Perez De Rosso and D. Jackson. What's Wrong with Git?: A Conceptual Design Analysis. In *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward!)*, pages 37–52. ACM, 2013.

[59] C. Policroniades and I. Pratt. Alternatives for Detecting Redundancy in Storage Systems Data. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 73–86. USENIX, 2004.

[60] P. Reiher, J. Heidemann, D. Ratner, G. Skinner, and G. Popek. Resolving File Conflicts in the Ficus File System. In *Proceedings of the USENIX Summer Technical Conference*, pages 183–195. USENIX, 1994.

[61] B. Shao, D. Li, T. Lu, and N. Gu. An Operational Transformation Based Synchronization Protocol for Web 2.0 Applications. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work (CSCW)*, pages 563–572. ACM, 2011.

[62] C. Smith. 33 Staggering Dropbox Statistics and Facts (2019) | By the Numbers, 2019. `https://expandedramblings.com/index.php/dropbox-statistics/`.

[63] M. Sousa, I. Dillig, and S. K. Lahiri. Verified Three-way Program Merge. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–29, 2018.

[64] Y. Su, D. Feng, Y. Hua, and Z. Shi. Understanding the Latency Distribution of Cloud Object Storage Systems. *Journal of Parallel and Distributed Computing*, 128:71–83, 2019.

[65] C. Sun, X. Jia, Y. Zhang, Y. Yang, and D. Chen. Achieving Convergence, Causality Preservation, and Intention Preservation in Real-time Cooperative Editing Systems. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 5(1):63–108, Mar. 1998.

[66] D. Sun and C. Sun. Context-Based Operational Transformation in Distributed Collaborative Editing Systems. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 20(10):1454–1470, Oct. 2009.

[67] L. Suresh, M. Canini, S. Schmid, and A. Feldmann. C3: Cutting tail latency in cloud data stores via adaptive replica selection. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 513–527, 2015.

[68] D. Svantesson and R. Clarke. Privacy and Consumer Risks in Cloud Computing. *Computer law & security review*, 26(4):391–397, 2010.

[69] D. B. Terry, M. Theimer, K. Petersen, A. J. Demers, M. Spreitzer, and C. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 172–182. ACM, 1995.

[70] W. F. TICHY. The String-to-String Correction Problem with Block Moves. *ACM Transactions on Computer Systems*, 2(4):309–321, 1984.

[71] W. F. Tichy. RCS – A System for Version Control. *Software: Practice and Experience*, 15(7):637–654, 1985.

[72] A. Tridgell and P. Mackerras. The Rsync Algorithm. Technical report, 1996. `https://openresearch-repository.anu.edu.au/bitstream/1885/40765/3/TR-CS-96-05.pdf`.

[73] B. Walker, G. Popek, R. English, C. Kline, and G. Thiel. The LOCUS Distributed Operating System. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 49–70. ACM, 1983.

[74] Z. Wu, C. Yu, and H. V. Madhyastha. Costlo: Cost-effective Redundancy for Lower Latency Variance on Cloud Storage Services. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 543–557, 2015.

[75] F. Zhu and F. He. Conflict Resolution for Structured Merge via Version Space Algebra. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):166, 2018.