Forensic Analysis of G Suite Collaborative Protocols

Shane McCulley
University of New Orleans, smcculle@uno.edu

Forensic Analysis of G Suite Collaborative Protocols

A Thesis

submitted to the Graduate Faculty of the
University of New Orleans
in partial fulfillment of the
requirements for the degree of

Master of Science
in
Computer Science

by

Shane McCulley

B.S., Louisiana State University, 2008

August, 2017

Contents

List of Figures

List of Tables

Listings

Abstract

Widespread adoption of cloud services is fundamentally changing the way IT services are delivered and how data is stored. Current forensic tools and techniques have been slow to adapt to new challenges and demands of collecting and analyzing cloud artifacts. Traditional methods focusing only on client data collection are incomplete, as the client may have only a (partial) snapshot and misses cloud-native artifacts that may contain valuable historical information.

In this work, we demonstrate the importance of recovering and analyzing cloud-native artifacts using *G Suite* as a case study. We develop a tool that extracts and processes the history of Google Documents and Google Slides by reverse engineering the web applications private protocol. Combined with previous work that has focused on API-based acquisition of cloud drives, this presents a more complete solution to cloud forensics, and is generalizable to any cloud service that maintains a detailed log of revisions.

Chapter 1

Introduction

In the past few years, cloud computing has rapidly changed from an emerging technology with great promise to the preferred method of delivering IT services, with nearly 95% of businesses adopting cloud solutions [12]. This has been led by a surge in cloud services, which surpassed cloud infrastructure in sales for the first time in 2016, growing at 3 times the rate of cloud infrastructure hardware and software.

The leader in cloud services is software as a service (SaaS), which accounts for the largest segment of cloud services [6]. In SaaS, the provider manages all aspects of deployment, and applications provided are accessible from a web browser or intermediate interface. This permits customers to delegate control over infrastructure and applications to the cloud service provider in exchange for ease of use, reduced maintenance costs, and scalability.

This has major implications for digital forensics, as the norm shifts from using software *products* to using software *services*. The distributed nature of these cloud services introduces many barriers to identification, preservation, acquisition, and analysis of evidence. Physical media no longer contains only a single user's data, and artifacts of interest are spread across multiple servers and locations. As cloud storage becomes cheaper, more data is migrating from hard drives to cloud drives like OneDrive, included with Windows, and *Google Drive*, a complete office and storage solution available anywhere. Features unique to cloud environments like multi-tenancy and data replication exacerbate existing technological, legal, and organizational challenges in data acquisition. Cloud forensics emerges as an important subfield to address these problems, with early efforts focusing on adapting traditional application forensics tools and techniques.

The traditional analytical model of digital forensics has primarily focused on working with physical media to recover all traces of activity and remnant processing data from both applications and OS. The SaaS model disrupts this view, as the investigator no longer has full control over sources of forensic evidence. Since code and data are delivered on demand, these artifacts become moving forensic targets. For example, a *Google Docs* document is represented on the local machine by a text document containing its url, document ID, and email address; the content may only be viewed or edited by using the browser. This shift in the way IT services are developed, maintained, and delivered necessitates a complementary shift in forensics, where the validity and reliability of forensic science is crucial in this new context and requires new methodologies for identifying, collecting, preserving, and analyzing evidence in multi-tenant cloud environments that offer rapid provisioning, global elasticity and broad-network accessibility. [9]

This paradigm shift requires substantial changes to the entire forensic process. Methods focused on low-level physical acquisition are becoming irrelevant — logical acquisition is becoming the norm. In particular, data acquisition in cloud drives introduces a number of challenges for client-centric analysis. Primary computation records and user history are no longer stored on the local machine, but reside in the cloud. Residual artifacts on the client are temporary with unknown provenance, as the cloud is the authoritative data source. As shown in figure 1.1, the client is used mainly as a cache for SaaS applications, and is not guaranteed to have a complete or updated copy of the data. Most cloud drive services provide some sort of revision history, but clients only maintain a single version on the local machine. Thus a client-centric approach to cloud drive acquisition is inherently unsound and incomplete.

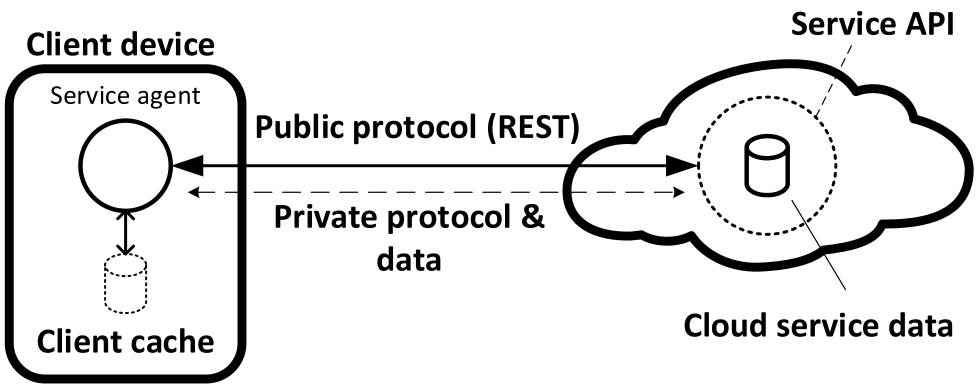


Figure 1.1: Cloud drive architecture [14]

The move from a standalone device to the cloud is accompanied by an exponential increase in the amount of data as it becomes the primary delivery method of IT services. This is exacerbated by the presence of multiple revisions for each file. The sheer volume of data requires new methods to identify, acquire, and analyze relevant evidence.

One promising area of analysis includes *cloud-native artifacts*, data objects that maintain the persistent state of web/Saas applications [14]. These artifacts are downloaded as needed by the application and have no serialized representation on the client. For example, a document or slide from *G Suite* does not maintain a permanent presence on the local machine. Each time it is opened, the state of the document is interpreted by the browser after receiving a *cloud-native artifact*. This is simultaneously a new problem and a new opportunity, as a large part of forensics involves reasoning about prior states and the effects of user actions. Direct access to the source of state change for an application would give us a clear picture of user actions and document evolution. This requires solving the problem of acquiring, storing, and standardizing a local representation of cloud-natives for analysis.

Though these are big challenges to overcome, *cloud-native artifacts* show great promise for forensic analysis. We find that in *Google Docs* and *Google Slides*, these internal data structures are requested in terms of document revisions, comprising a *chunked snapshot* object and an append-only changelog of user edits. The snapshot contains text and necessary styles needed to produce the document as of the starting revision, followed by a list of actions in the order they were performed. These logs contain the entire editing history of a document or presentation, which is perfect for forensics — the document can be examined at any point in time. Instead of piecing together history from a series of (possibly incomplete) snapshots, we have the full revision history available. This is a wealth of information which is unusual for traditional forensics, which has few analogs; most forensic tools are developed to handle a single version of an artifact. As the changelog is append-only, there is no chance of spoiling the history of the artifact. Clearly, there is a need to develop forensic tools that can acquire, process, and store *cloud-native artifacts* as well as development of new models that incorporate these artifacts.

Previous work on API-based acquisition from cloud drives approaches these problems by working directly with the authoritative data source. Relying on the provider's public interface has many advantages, such as ease of use, access to file metadata, and avoiding the necessity of reverse engineering each cloud drive servicee. These API calls have well-defined semantics and detailed specifications which lends itself to forensic tool development. The proof-of-concept tool `kumodd` [13] showcases these features, enabling total or partial acquisition of cloud drive data, including revisions. However, *cloud-native artifacts* such as *Google Docs* and *Google Slides* are only available as snapshots converted to standard document formats. This provides a forensically sound solution to basic acquisition challenges, but it also omits a significant amount of information of forensic interest.

To address the shortcomings of the public API-based approach, we examine the private communication protocols used by these services. With persistent state being maintained in the cloud and the browser performing mostly user interaction functions, we target these private calls through which the browser sends and receives updates. We provide a baseline forensic analysis of *cloud-native artifacts*, using *G Suite* as a case study to discuss the issues and approaches on working with these artifacts. Specifically, we:

- Analyze the shortcomings of traditional client-based forensic approaches and expand upon previous API-based acquisition and analysis by targeting *cloud-native* *artifacts*. We analyze and document *G Suite* collaboration protocols with a focus on the internal changelog of *Google Docs* and *Google Slides*.

- Develop a prototype, `kumodocs`, that acquires *cloud-native artifacts* for *Google Docs* and *Google Slides* for any revision range, allowing us to extract plain text, images, comments, and suggestions from the document at any point in time. `Kumodocs` also recovers deleted images as well as resolved comments and suggestions no longer visible on the document, and provides a means to store and interpret these artifacts independently of *G Suite*.

- Present an intermediate format for changelog storage in an effort to standardize tools and methods for *cloud-native artifacts* analysis. This modular approach facilitates tool reuse, as artifacts from new services only need to be translated into this intermediate format. Any changes in the internal data structure can also be fixed quickly by addressing the parser for that particular service.

The remainder of this work is organized as follows: Chapter 2 provides an overview of traditional client-based cloud forensics and improvements made through API-based methods; Chapter 3 presents our analysis of *G Suite* collaboration protocols; Chapter 4 details our proof-of-concept tool, `kumodocs`, and our challenges in creating it. Chapter 5 and 6 include our discussion and conclusions.

Chapter 2

Related Work

Previous research in cloud storage forensics has consisted mainly of adapting traditional application forensics to finding artifacts remaining on the client after using cloud services. Disk and memory images of the client are taken before and after some particular use case, and differential analysis is applied to locate every trace of computation and activity left by popular services.

Section 2.1 provides a review of several representative studies of client-side acquisition and analysis. Section 2.2 summarizes inherent flaws in the client-side approach. Section 2.3 presents newer work that suggests solutions to these shortcomings by utilizing APIs in a service-centric approach.

2.1 Client acquisition and analysis

Chung et al. [2] examined popular cloud service providers among a range of services, including *Amazon S3*, *Google Docs*, *Box*, and *Evernote* to determine artifacts left on systems running *Windows*, *MacOS*, *Android*, and *iOS*. They proposed a process model for forensic investigation of cloud storage services based on artifact analysis on the client to determine what kind of data exists, where this data is located, and how it can be utilized. Key areas targeted include physical memory (if available), internet history, log files, and directories. Android phones were rooted to provide backup information and iPhone data was obtained from iTunes and iPhone backup files.

Accessing *Google Docs* through a web browser created temporary files with the content of current documents and presentations open. Once the browser was closed, these files were deleted. This lack of permanent data on the client demonstrates one of the shortcomings of client-based analysis.

Hale [7] analyzed digital artifacts left behind after *Amazon Cloud Services* been accessed or manipulated from a *Windows* machine. Like *Google Drive*, *Amazon Cloud Services* can be accessed through a standalone application, a web browser, or a mobile device; however, mobile devices were not explored in this study. Using the browser, artifacts were found in browser history and cache files. He analyzed traces left from the installation and usage of the client application, finding artifacts in the *Windows* registry, application installation files, SQLite database, and an append-only transaction log. The database contained records of pending upload/download operations, which were moved to the log after completion or interruption.

Quick and Choo [10] analyzed data remnants on a Windows 7 computer and an Apple iPhone after accessing a *Dropbox* account to store, upload, and access data. Hash analysis, common file locations, and keyword searches were used to determine if a *Dropbox* account has been active. They found evidence of usage including usernames in browser history after web access from Mozilla Firefox, Google Chrome, and Microsoft Internet Explorer. Further artifacts were found in directory listings, prefetch files, link files, thumbnails, registry, and memory captures. The same methodology was applied by Quick and Choo to *Google Drive* [11] in a follow-up work that documented client-side operations and artifacts useful as a starting point for investigators.

Martini and Choo [8] researched *ownCloud*, a self-hosted open source file sync and share server. *OwnCloud* offers a unique solution and opportunity of study, as both the server and client are under control of the user. They were able to extract sync and file management metadata, cached files, authentication data, browser, and mobile client artifacts. Client analysis revealed artifacts able to link a user to a particular *ownCloud* instance, even after deletion of evidence on the client. They argued the open-source nature of *ownCloud* is likely to influence future developments in open source cloud storage products, which may include similar artifacts.

2.2 Limitations of client-side acquisition and analysis

The approaches so far involve extensive analysis of the client in search of artifacts left by cloud services. The key assumption underlying these studies is that all artifacts of interest can be obtained from the client. However, the client is not the authoritative source of data; there is no guarantee the client has the latest data, or that the client contains a complete record of activies. In addition, application artifacts are not persistently stored on the client, and web applications tend to use local storage as a cache with no guarantees to its completeness or accuracy. There are three major inherent shortcomings in client acquisition and analysis of artifacts [14]:

2.2.1 Partial replication

In the studies listed above, researchers were able to find evidence of cloud-specific usage, and occasionally which files were accessed or modified. However, they were unable to find these files on the client. As cloud storage is a distributed storage solution, it makes sense that none of the clients would have a complete copy of the data. Selective replication of data is necessary to allow access to mobile devices with small local storage. Current trends show exponential growth in digital content, a conclusion that is unlikely to change in the future as cloud storage becomes cheaper and users generate even more data. In addition, metadata artifacts recovered from the client are only relevant to transient local data, and tell us little about what artifacts are available in the cloud. It is necessary to have direct access to the cloud drive's metadata in order to identify missing artifacts and verify integrity; without this information, the completeness and accuracy of client-side data acquisition cannot be assured.

2.2.2 Artifact revisions

Most cloud service providers, especially offering collaborative services, have some provision for accessing previous versions of files for editing or recovery. While the user rarely has control over when and how revisions are stored, it does offer a small means of version control and the ability to revert major mistakes, a feature users have come to expect. These file revisions are a particularly interesting source of forensic data, which has few analogs in traditional forensic targets. The ability to view a document's evolution over time directly instead of composing a best guess from scraps of user actions gleaned from log and timestamp data is invaluable, and reduces the chance of evidence deletion if these revision artifacts can be recovered. However, the client has only one revision at best, usually the latest version. Client-side acquisition will miss these revisions, and as mentioned previously the lack of metadata means it is impossible to know what revisions are available or missing.

2.2.3 Cloud-native artifacts

Web applications rarely store persistent state on the client. As code and data are sent as needed, the cloud must maintain this state information in *cloud-native artifacts*, data objects that have no serialized representation in the local filesystem. [14] This is a new problem for forensics, as there is no way to obtain these artifacts from the client; they must be requested from the cloud service itself or captured in private communication protocols as the browser sends and receives updates. In *G Suite* collaboration services, we find that these internal data structures are append-only revision logs that contain the entire history of a document. API support for these artifacts is limited — for example, *Google Docs* and *Google Slides* presents a method for downloading a snapshot of these artifacts at some point in time in PDF or standard file formats, but this misses the rich revision history that is of forensic interest.

2.3 API-based acquisition

The key to a forensically sound acquisition of data relies on targeting the source of data — namely, the cloud service provider. The most direct way involves utilizing the provider's infrastructure through the publically offered API. This is the lowest available level of abstraction for forensic acquisition. Methods are often provided for metadata access, giving a wealth of information of how and when data was accessed or modified as well as integrity guarantees for acquisition. Cloud service providers offer well-defined mechanisms for access through the API, which is the same mechanism that web applications use (with the exception of private/undocumented calls). These can be modeled and tested, allowing a formal and precise approach tool development.

2.3.1 Kumodd and motivation for G-Suite Study

In [13], Roussev et al. demonstrate the feasibility of an API-based approach through the development of `kumodd`, an API-based acquisition tool that works with *Google Drive*, *Dropbox*, *OneDrive*, and *Box*. Public API methods are utilized to perform content discovery, target selection, and target acquisition. `Kumodd` acquires files and associated metadata, with filtering options based on targets of interest. These filtering options can be used to obtain a subset of the cloud drive, or the entire drive may be requested for a complete acquisition. Revisions are also enumerated and downloaded for all files requested, with creation timestamps appended to the filenames for distinction. It can also obtain snapshots of *cloud-native artifacts* in standard formats such as PDF, ODT, DOCX, etc.

`Kumodd` fully addresses 2 of the major problems with client-side acquisition — that of partial replication and revisions. However, `kumodd` only partially addresses the third challenge, as is only able to obtain snapshots of documents and not *cloud-native artifacts* themselves. Since these artifacts are internal data structures utilized by the the browser to communicate state information, there is no reason to expose these artifacts via the public API. These artifacts are communicated through private internal protocols to the web applications as noted in figure 1.1. Thus, any acquisition and analysis efforts would involve reverse engineering these private protocols, which motivates our case study of *G Suite* collaborative protocols in order to fully address the issue of *cloud-native artifacts*.

Chapter 3

G Suite Analysis

Our study of *G Suite* protocols is partially motivated by *Draftback*, a browser extension that allows character-by-character playback of *Google Docs* history. This extension, created by writer and programmer James Somers [15], was developed to study how writing documents evolve over time. Specifically, by knowing where and when edits occur in a document, the entire document history can be replayed from any point. The history of each character is recorded and assigned a unique persistent ID, allowing a user to follow the history of particular sections of a document over time. Mapping this data to a chart creates a visual representation of edited history.

3.1 Draftback

Being able to rewind a document to any point in time – even to the beginning – is exactly the kind of information forensic examiners are interested in. *Google* records and preserves elementary actions used in building up a document in a *revision log*, which is used to power *Draftback's* functionality. While intending to be a tool for analyzing and improving writing, this is likely the first cloud-native tool with forensic applications. All plain-text from a document can be replayed in the browser with real-time delays, or at super speed, with detailed statistical visualizations mapping those changes to document location. The extension is started from within a *Google Document*, which retrieves and processes the revision log; this process took 6 minutes for a x-page document with 7600 revisions.

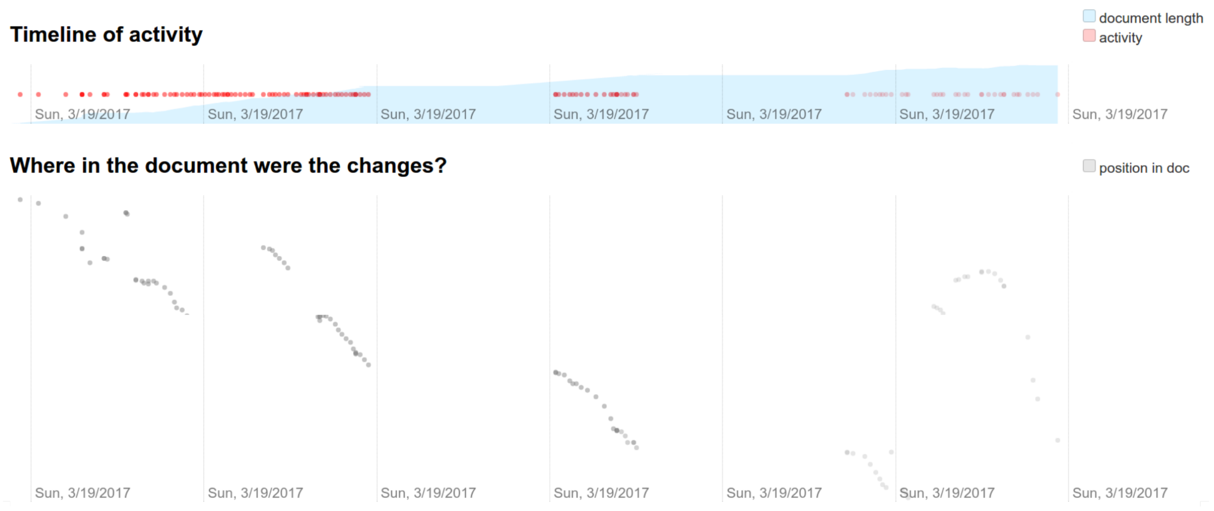


FIGURE 3.1: *DraftBack* edit visualization

For large documents such a history is invaluable, enabling focused investigation of relevant sections. The revision log is obtained by (partially) reverse engineering web protocols – no client-side data collection is performed. The only requirement consists of valid user credentials; with this in hand, examination can be performed anywhere with Internet access. *Draftback*'s major drawback is its incomplete handling of the *Docs* protocol – reverting to a prior version breaks all functionality past that point.

While not motivated by forensics, this is an example of SaaS analysis that showcases the kind of interesting and useful *cloud-native artifacts* that can be obtained by going directly to the data source. Recall that we defined *cloud-native artifacts* as data objects that maintain the persistent state of web/Saas applications and have no serialized representation on the client. The forensic application of these results served as a starting point for our work in reverse engineering *G Suite* collaborative protocols to acquire and analyze these cloud natives.

3.2 Background

Most online text editors use *contentEditable*, the native browser widget for editing rich text. An application notifies the browser to make a string of text editable and lets the browser handle the specific details. When a user types, requests a style change, or changes the layout of text, the browser interprets these commands and changes the page's HTML accordingly. This delegation means creating behaviors consistent across all browsers is exceedingly difficult.

Advantages to using native browser rendering include ease of implementation and optimization. Layout performance is heavily optimized in modern web browsers, freeing apps from handling the most resource-intensive task. However, this comes at a cost of limited control over document behavior and feature implementation. Bugs arising from browser interactions are unable to be fixed, and a consistent user experience cannot be guaranteed. This motivated *Google* to create an editor that keeps full layout control and bypasses the browser itself.

3.3 Documents

In 2010, *Google* revealed a new version of *Google Docs* [3] showcasing greater real-time online collaboration. The new *Google Docs* editor, codenamed *kix*, was custom-built to look and feel like a traditional word processor. This was achieved by avoiding the *contentEditable* field, bypassing the browser completely by developing a new editing surface and layout engine in JavaScript. All text entered is captured and displayed on a dynamically changing surface according to *kix*'s layout manager. Even the cursor is a simulated element, a 2-pixel-wide div manually placed on the screen at the appropriate coordinates. This allows multiple concurrent users to each have their own cursor displayed on the screen, giving visual cues that aid in collaboration by indicating areas currently being edited. These changes also allow simple features like, tab stops, floating images, and a ruler to be added, which were impossible to support with a browser's native HTML layout engine at the time. [5]

Determining the location of elements on the editing surface is the most difficult thing the *kix* editor does. Typing a single letter goes through several steps before appearing on screen. First, the letter is drawn off-screen, measured for size, and the screen location is determined for placement. If this letter appears before other words, everything is pushed forward with boundary checking creating new lines as appropriate. Once the new character positions are determined, the page is updated with dynamic HTML elements and filled with content. Due to the efficiency and optimizations of an engine built from scratch this happens at an incredible speed, giving the look and feel of a desktop word processor.

3.3.1 Google's collaboration technologies

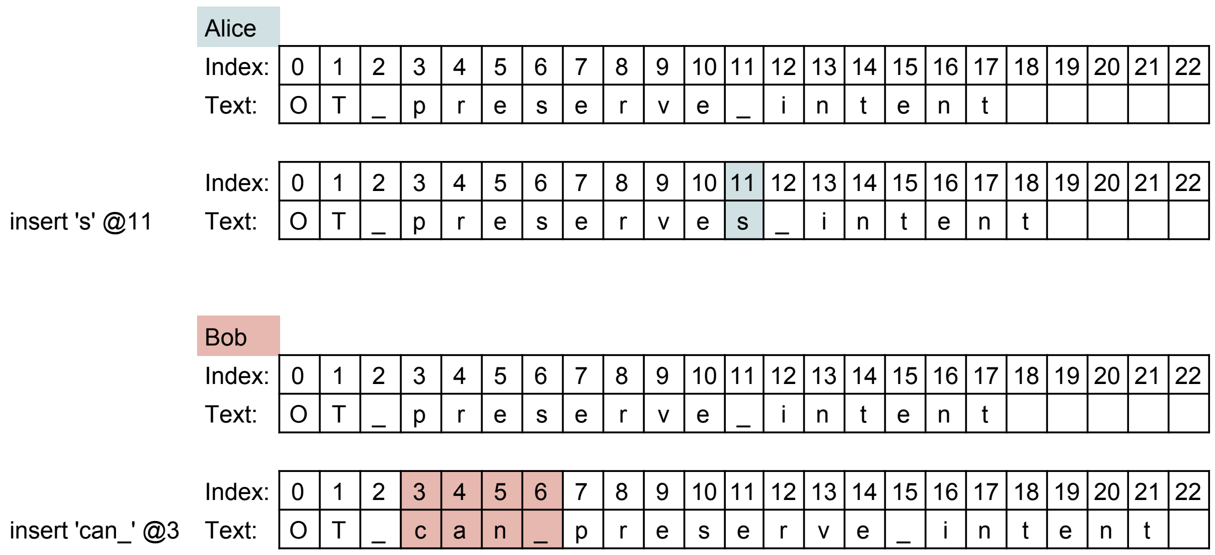
The *kix* editor is not just an online clone of existing standalone word processors. It was "designed specifically for character-by-character real time collaboration" [5]. By writing their editor from the ground up, *Google* solved two major problems of real-time collaboration:

1. Maintaining consistency when multiple users edit the same area of a document, and
2. Properly merging multiple simultaneous edits.

The key technology enabling the first point is *operational transformation*(OT)[4], a consistency and concurrency management solution that avoids locking by transforming edits relative to each other. Each change in a document involves one of three elementary actions: inserting text, deleting text, or applying styles to text. These actions occur at specific indices in a document, which can be transformed against prior edits to achieve a consistent version for every user. The second issue is solved by Google's *collaboration protocol*, which allows rapid communication between clients and determines when OT is used.

To see how these elementary actions can be transformed against each other, consider a document with the text "OT preserve intent" being edited simultaneously by two users, Alice and Bob. The document contains a grammar error that can be fixed two ways — "OT *preserves* intent" or "OT *can* preserve intent". Alice adds `"s"` to preserve, which ends at index 11. The corresponding primitive sent to the editor is {`insert "s"`

@11}. At the same time, Bob writes "can_" before preserve, which the editor interprets as {insert "can"_ @3}.



Alice

Index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
Text:	O	T	_	p	r	e	s	e	r	v	e	_	i	n	t	e	n	t					

insert 's' @11

Index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
Text:	O	T	_	p	r	e	s	e	r	v	e	s	_	i	n	t	e	n	t				

Bob

Index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
Text:	O	T	_	p	r	e	s	e	r	v	e	_	i	n	t	e	n	t					

insert 'can_' @3

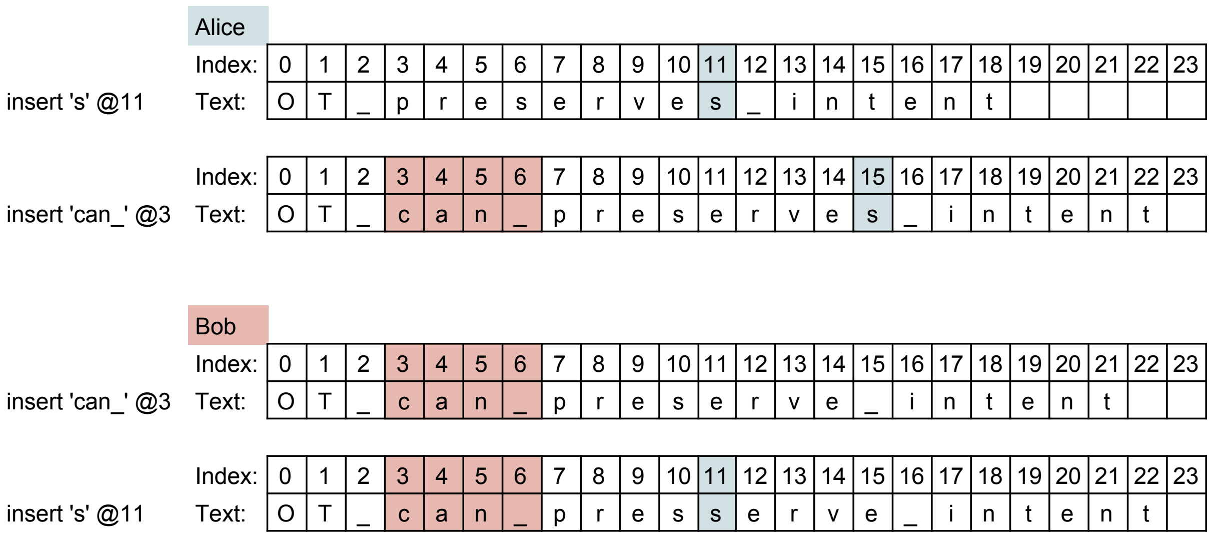
Index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
Text:	O	T	_	c	a	n	_	p	r	e	s	e	r	v	e	_	i	n	t	e	n	t	

FIGURE 3.2: Each view with only local changes applied.

Without operational transformation, each user would apply their own changes locally, and then receive the other's change from the server. Alice's client would apply 's' at index 11 and then receive an action from the server instructing his client to insert 'can_' at index 3, ending in "OT can preserves intent".

Simultaneously, Bob's client inserts "can_" at index 3, giving "OT can preserve intent". The server sends Alice's edit as {insert "s" @11}. Since the index sent to Bob from Alice's update is not relative to local changes made, the final version on Bob's client is "OT can presserve intent". The order in which the operations are applied result in two different outcomes and an inconsistent document.

OT fixes this problem by translating the indices based on local changes when an edit is received from the server. In the first case, inserting text at index 11 does not effect the position of characters before it – so no change is necessary. In the second case when Bob's client inserts "can_" at index 3, each character's new position is increased by **4**, the length of the string inserted. Any changes Bob's client receives referencing old index positions must be shifted by 4 to account for the local changes that Alice's client was unaware of. Once this transformation is applied, the new instruction becomes {insert



Alice
Index: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
insert 's' @11 Text: O T _ p r e s e r v e s _ i n t e n t
Index: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
insert 'can_' @3 Text: O T _ c a n _ p r e s e r v e s _ i n t e n t
Bob
Index: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
insert 'can_' @3 Text: O T _ c a n _ p r e s e r v e _ i n t e n t
Index: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
insert 's' @11 Text: O T _ c a n _ p r e s s e r v e _ i n t e n t

FIGURE 3.3: Inconsistent documents after all edits applied.

`"s" @15}`. The letter inserts at the correct position, and both users see the *same* final product — "OT can preserves intent"

Correct implementation of operational transformation assures every collaborator will have a consistent document once all changes have been applied. The basic process involves edits being applied locally and then sent to a server, which is then distributed to all collaborators. Actions are transformed relative to the local document when incoming edits are received. For instance, a style change of `{italicize @10-20}` transformed against `{insert "key" @15}` would result in `{italicize @10-23}`. If there are no conflicts, the actions are applied without transformation - such as in the first case of our example. This technology allows real-time collaboration in the same area of a document without conflicts.

While operational transformation is the mechanism that unifies primitive actions sent and received from users, the *collaboration protocol* is what enables and determines when it is used. Collaboration consists of a series of edits being sent from a client to the server, and those changes being transmitted from the server to other collaborators. This protocol outlines what data clients and servers are responsible for maintaining, how this data is updated, and when these updates are communicated. The client tracks four key elements of the document[4]:

1. The most recent revision number received from the server.
2. Local changes queued to be sent to the server.
3. Local changes sent to the server but not acknowledged.
4. The current state of the document with local changes applied.

The server is responsible for maintaining the internal representation of the document, which is a complete history of all processed changes called a *revision log*. When edits are made in the document, the client sends these changes to the server. The client waits for the server to acknowledge processing the changes; until then, all local changes are saved in a pending log. Only one pending change is sent to the server at a time.

The server also has a queue of pending changes yet to be processed and applied to the document. Once an update is accepted into the revision log, the server informs the client. The client now repeats this process from the beginning, sending the contents of its pending log, erasing the pending log, and storing new changes until the next acknowledgement. The differences in pending logs between client and server result in two separate views of the document. The client views the document's state with all local changes applied, and from the server's perspective the document state consists of the last processed change.

When changes get processed, the server updates all other clients with the accepted edit and the corresponding edit number. These incoming changes are compared to the list of pending changes the client maintains. In the case of simultaneous or conflicting edits, operational transformation is used to change the pending actions. The client applies these transformed changes along with the server's edits, and the revision number is updated to match the server's version. The server processes changes in the other in which they are received, informing clients of the current "official" state and the revision number after each change.

To see how this process works, consider the following example with 2 users editing a shared document, Luiz and John. The server is represented in the middle, while Luiz and John's editors are represented on the left and right sides, respectively. User edits are represented in ovals, which are sent between the server and other clients.

The document starts empty, with Luiz typing `"Hello"`. These edits are added to the pending list to be sent to the server, and the local document is updated immediately. Luiz's client has no sent changes awaiting confirmation, so the pending log is sent to the server. The edits are moved to the *sent* list, and any new changes will remain in the *pending* list until confirmation of the last update.

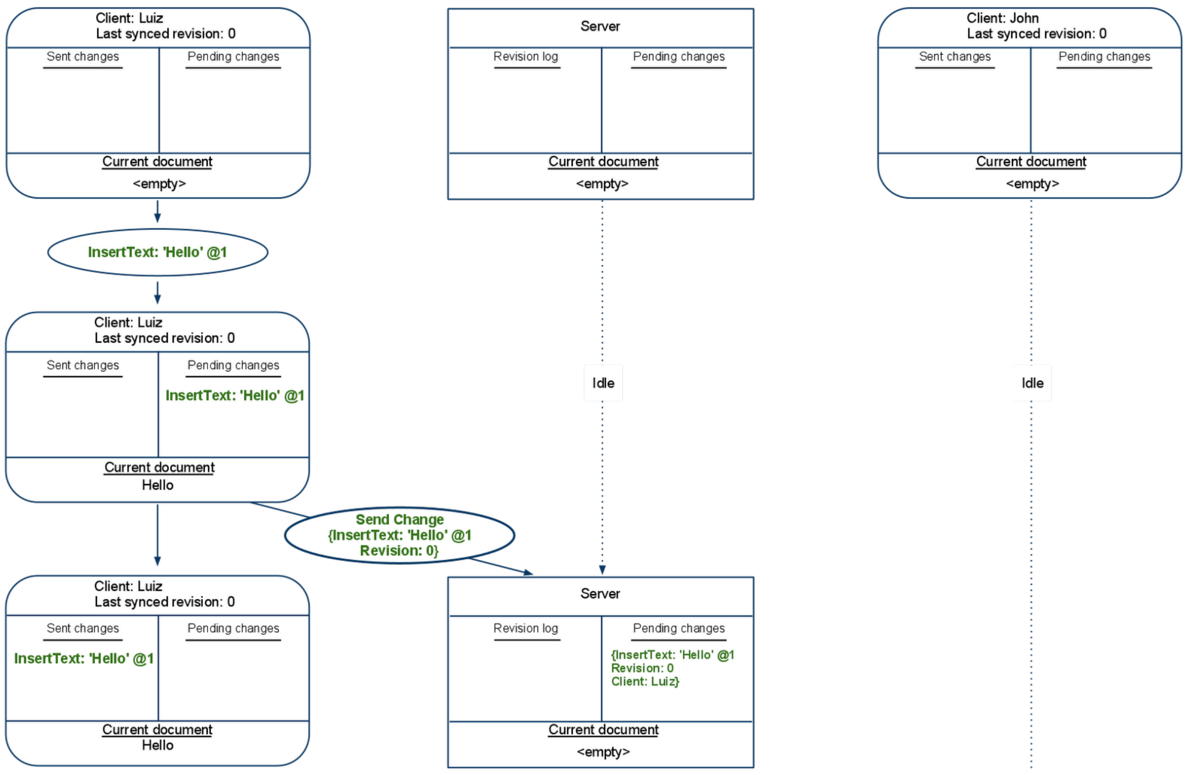


FIGURE 3.4: Luiz types `"Hello"` into his browser[4]

Next, John enters `"!"` as Luiz simultaneously types `"world"`. At this point, the server has Luiz's first edit in its list of pending changes. Luiz's view of the document includes all local changes made, and reads *Hello world*. Luiz's client has sent `Hello` to the server, and all further changes will go into the pending list until receiving acknowledgement from the server. John types `"!"`; his client has not received any changes from the server, so his document view is a single exclamation point.

The server has a single edit in its pending changes, so this is applied to the document. The document is now at *revision 1*, and every client is notified to sync with the new version. For Luiz, this confirms his change has been accepted; the edit is removed from

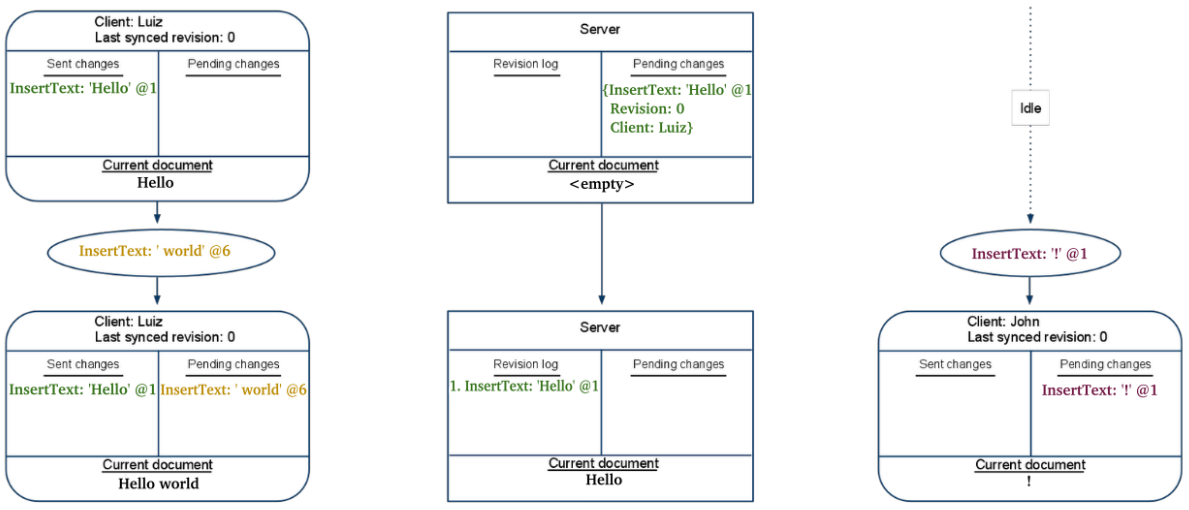


FIGURE 3.5: Luiz writes "world", and John writes "!" simultaneously.[4]

the list of sent changes, and the document updates to revision 1. John's client receives the revision as an instruction: {insert "Hello" @1, revision 1}. Pending changes are transformed against the incoming changes - since "Hello" is 5 characters long, John's edit of "!" should be transformed to index **6**. John's document now reads *Hello!* at revision 1, and there is a pending change to send to the server.

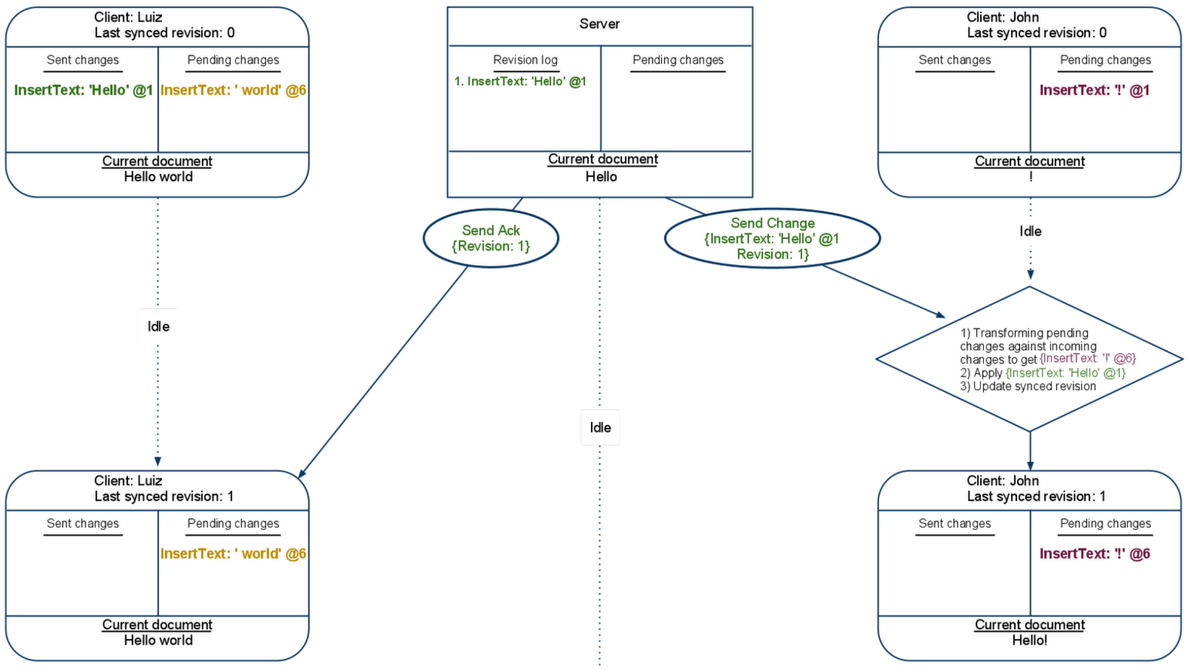


FIGURE 3.6: Luiz receives server ack, and John receives Luiz's first change.[4]

Both clients now send their pending changes to the server, locally moving the change from the pending list to the sent list. The server receives each change and places it into the pending queue. Since Luiz's message {`insert "world" @6`} was received first, the server processes this request and sends an update to all clients. John's client receives the new edit; since John's edit has not been processed, {`insert "!" @6`} is still pending. Pending changes are again transformed with operational transformation, and John's insertion is moved from index 6 to index 12, as `world` occupies indices 7-11. John's client now reads `Hello world!`, and Luiz sees `Hello world`, with both clients updating to revision 2.

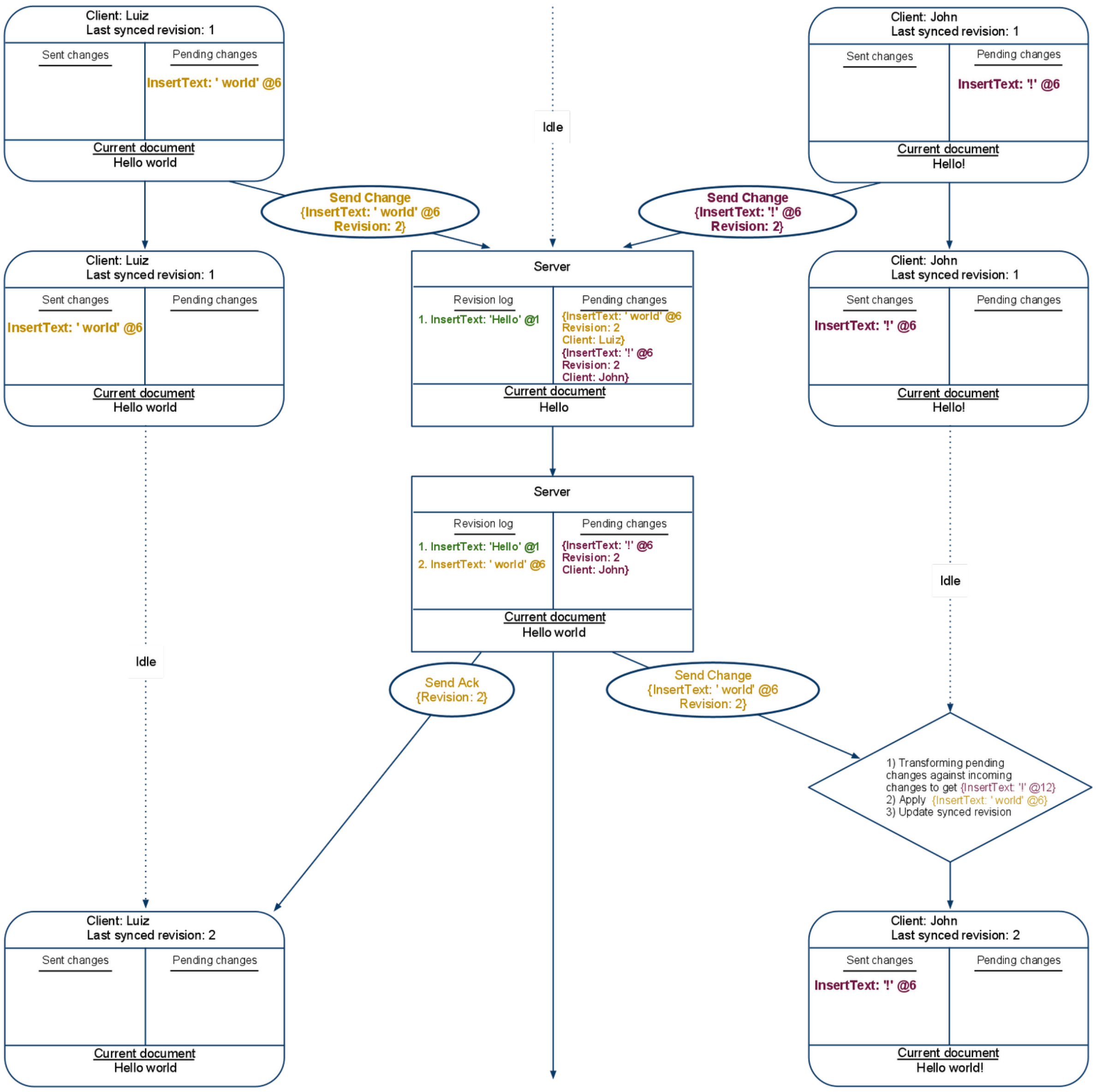


FIGURE 3.7: Both clients send a new edit and receive the latest revision {`insert "world" @6`}[4]

The server now has one final edit to process, {`insert "!" @6`} from John as *revision 2*. A previous change was already entered as revision 2, so this edit must be transformed against previous changes. This results in the same edit John's client applied locally, {`insert "!" @12`} and is sent to each client as *revision 3*. Luiz's client updates the view to `Hello world!` and John's client removes the change from the sent list. Both clients now update their last synced version to *revision 3* and have a consistent view now that all changes have been applied.

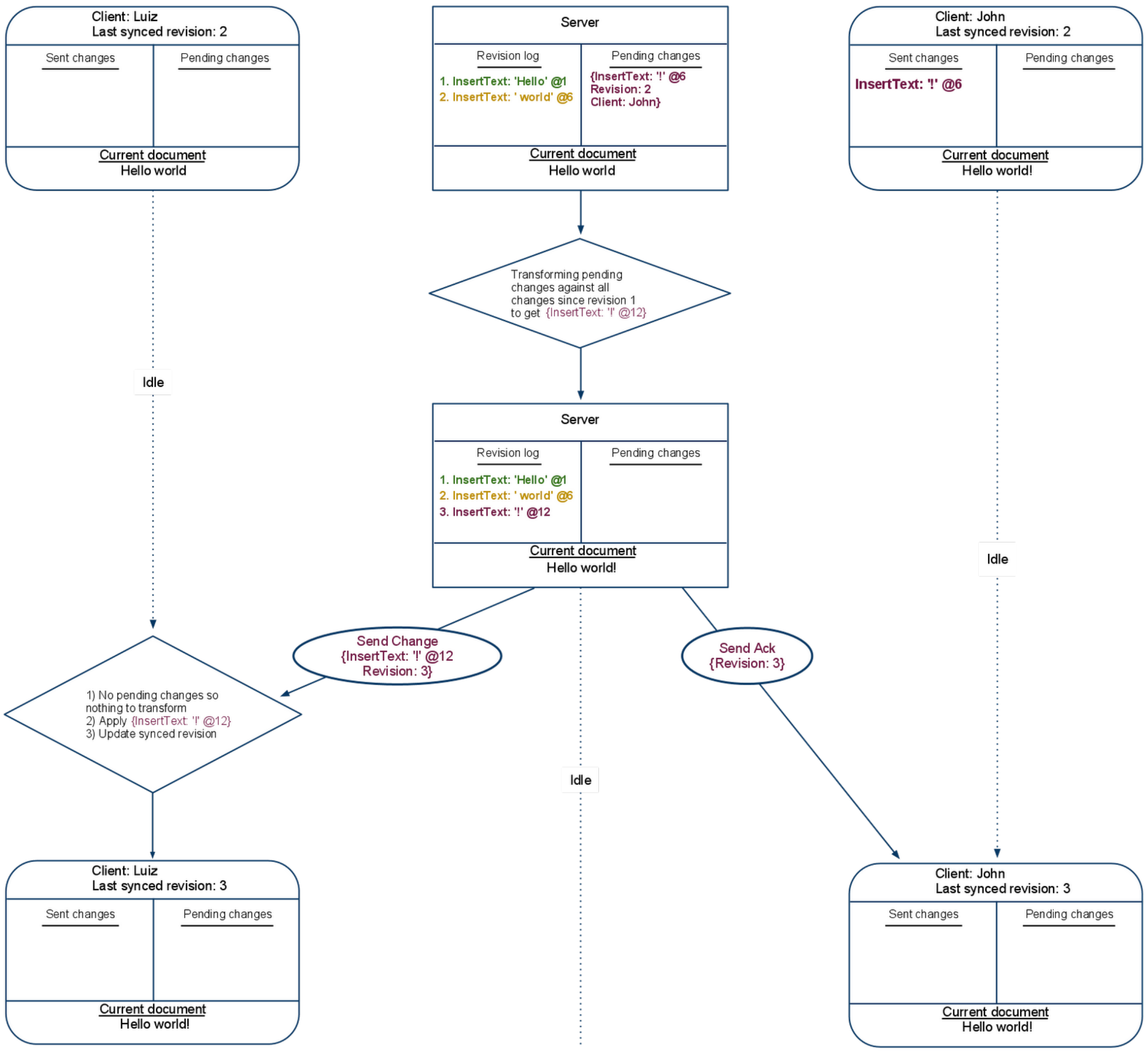


FIGURE 3.8: After all edits have been received, both clients have a consistent view[4]

3.3.2 Revisions

Google's approach to storing revisions is motivated by the desire for fast and efficient communication between client and server. Rather than revisions being stored as snapshots and compared with previous versions, the complete editing history of a document is kept. The revision log is populated by a series of updates from each editing client as mentioned above. The client sends an update using the `save` method, where the new revision number and edit are sent as form data in JSON encoding:

```
{rev:254
bundles:[{"commands":[{ "ty":"is","ibi":11645,"s":"ten" }],
  "sid":"7c0694922e589fca","reqId":164}]
```

This example shows a user typing the word **ten**, which is inserted before index `11645`. The current session is marked by the **sid** (session ID), which has submitted 164 revisions. As a whole, this is the document's **254**th revision as noted by **rev**. A new session could be started by closing the browser, and reopening the same document; the first edit would then be issued a new `sid` and `reqId` would be reset at **0**. While waiting for acknowledgement, the client will collect all further edits into the pending log as described above. The server commits the change to the revision log after processing, and attaches a server-side timestamp.

This revision log is the internal representation of a document's state, requested by the client whenever a document is displayed. There is no permanent record of the log on the local machine after processing, an example of a *cloud-native artifact*. Any range in the document may be requested by specifying a starting and ending revision number to view prior versions. The general structure of the revision log consists of a *chunked snapshot* followed by a *changelog*, which are deeply nested JSON structures consisting of arrays and objects with key/value pairs. The keys are abbreviated, with most of them being 2-4 characters long.

The chunked snapshot contains all information required to create the document as of the starting revision. Length varies greatly, depending upon the number of embedded *kix* objects and paragraphs, as well as the starting revision. The first revision of a page, before any user edits, consists of applying default text and paragraph styles. A chunked snapshot from revision 1 contains only 2 objects representing the application of text and paragraph styles.

For documents containing text, the first element of the snapshot consists of a plaintext string of all text in the document. This is followed by several style definitions that hold over the entire document, such as margins, language, and default heading styles for *title*, *subtitle*, and headings *H1* through *H6*. Inserted elements are listed by referring to their *kix* anchor ID and associated styles. Following this is a list of contiguous format areas and associated styles, as well as any comments or suggestions that appear in between. A new format area is created whenever a character contains some text style differing from the previous character; the associated change and the index are noted as a new span. New paragraphs have a list of style modifiers to denote any paragraph changes, such as indentation or alignment, as well as IDs that can be associated with a table of contents for quick navigation. Extremely long documents have the snapshot broken up into multiple sections, with plaintext strings of approximately 4100 characters. After the first object, they all follow the same form - the plaintext string comes first, followed by the list of spans and paragraphs with associated styles. The browser then constructs the snapshot by pasting the given plaintext and applying styles over indicated ranges.

The *changelog* contains a list of arrays of each elementary action performed on the document, allowing any revision to be viewed. The basic form of each revision contains a dictionary of attribute styles to apply, an action (insert, delete, or apply style), any inserted text and indices for the action, followed by identifying information. The identifying information consists of an epoch timestamp in Unix format, the Google ID of the author, revision number, session ID, session revision number, and the revision itself. Every time the document is opened, a new session is generated and given a unique ID; the number of revisions submitted under this session are counted in the session revision number. Some revisions, such as inserting objects, appear as a single ”entry”, a transaction consisting of several actions that were applied at the same time. A version requested from revision 200-300 would contain a chunked snapshot as of revision 200, followed by 100 arrays in the changelog for each revision. The ability to request exact ranges allows *kix* to present detailed revision history in the browser with minimal effort, avoiding the need to constantly replay the same history.

To display a specific version in the browser, the log is played from the beginning until the requested revision is reached; replaying the entire log produces the most recent version. This greatly reduces the amount of storage for each file and communication overhead, and allows for edits from multiple users to be integrated seamlessly due to the granularity of the changes. When viewing the history in the browser, each revision is accompanied by a revisions/load request from the client, specifying a start and end revision. insert picture This load request can be used to access the changelog directly, which is sent as a JSON file.

Collaborative editing is supported by quick client updates, with a new revision being saved as often as every 150 ms under constant input. This update time depends upon several factors, including the minimum update time, speed of user input, and the amount of time the client spends waiting for the server to acknowledge the last update. Most revisions consist of adding or deleting 1-3 characters at a time. While collaborating, the granularity of these changes together with operational transformations result in a consistent document, able to handle multiple simultaneous edits in the same location from different users.

Revisions can be accessed through the browser and retrieved through the public API, although the options presented are limited. Only *major* revisions are presented to the user. The mechanism behind major revision creation is unclear; however, major style changes and length of time since last update seem to be major factors. For example, a document tracking experiments with over 7600 revision actions has 20 major revisions, while another used for reverse engineering purposes has 30 major revisions with only 750 revision actions. A major revision seems more likely to occur if the time since last edit has been a day or longer. Creating a new session by exiting the browser and accessing *Google Docs* again was not enough to trigger a major revision.

As the revision log contains the internal state of a document, it is sent to the browser upon loading or after requesting a revision. This network traffic can be viewed by using developer tools that come with most modern browsers. Upon loading a specific revision, the client sends a request of the form `https://docs.google.com/document/d/<doc_id>/revisions/load?id=<doc_id>&start=<start_rev>&end=<end_rev>&token=<auth_token>`. The `doc_id` is the unique document identifier, which is part of the standard URL for editing a document; `start_rev` and `end_rev` denote the start and end revision number, and `auth_token` is an OAuth2 authentication token. Requests must be well formed; the starting revision can not be less than 1 or greater than the ending revision, and the ending revision cannot exceed the most recent revision number of the document.

This request can be manually copied and pasted into the browser, editing the starting and ending revision ranges as required. Chrome and Firefox also provide a "save as cURL" option in developer tools, which can be entered into the command line. The response to this request is a standard JSON file as described above, which begins with the closing braces `)]}'`, likely a safety closure or requirement of some internal protocol.

3.3.3 Changelog data

Each revision in the changelog corresponds to a single JSON array, where the first element of the array is a JSON object, which is represented as key/value pairs. The structure becomes deeply nested when values contain further arrays, with even more objects contained within them. The keys are abbreviated, most likely due to performance reasons, and not easily readable by humans. Revisions in the changelog have a similar structure to the chunked snapshot, with the addition of identifying information. Elementary actions consist of insert, delete, adjust style actions, followed by a dictionary of parameters associated with the action type.

Occasionally, revisions will contain multiple elementary actions to be performed simultaneously. These actions are wrapped in an overall type of `mlti` (multiaction), a transactional action allowing several insert or delete edits to occur with the same timestamp. Multiactions most commonly occur with manipulating page objects, but can also be found when a single revision update contains both an insert and a delete action. Any updates that contain multiple differing actions will be resolved into a single transaction.

Due to the append-only nature of the changelog, we were able to map each possible action in the document to a corresponding revision. While this gave a list of keys associated with each action, deciphering the meaning of each key was not as straightforward. Text and paragraph styles were the most common style keys, each being prefixed by `ts_` or `ps_`, respectively. Text keys consisted mostly of straightforward abbreviations – `ts_ff` for *text style font family*, `ts_bf` for *bold font*, and `ts_it` for *italic*. Each paragraph and text style has a corresponding flag key ending in `_i`, such as `ts_bf_i` for bold font. This flag denotes whether the style continues to be inherited from the preceding span; false indicates a change, true continues the implied style.

Paragraph styles include default header settings for title, subtitle, and custom headers H1 through H6. Each header provides settings for alignment, indentation, line spacing, spaces before and after paragraph, and several unknown keys. Text styles include parameters for bold, italic, strike-through, underline, background and foreground color, font family, font size, vertical alignment, and small caps. A full listing of key mappings can be found in the appendix.

Actions that manipulate page elements, such as inserting or deleting tables, equations, pictures, etc., are bundled as a transaction of multiple elementary actions. The first action inserts a placeholder character at the appropriate index, usually a star ($\star$). Element properties are modified through an `adjust element` (ae) action, which sets values such as margins, width and height, title, and ID. The remainder involves adjusting page and text styles as necessary. Inserted objects are associated with a *kix* "anchor" and ID by the `tether element` (te) action. Deleting an object requires a combination of `delete element` (de) with the appropriate ID, followed by a string deletion of the placeholder value.

```
{ "ty": "ae",
  "et": "inline",
  "id": "kix.iz8zrpzcxarv",
  "epm": {
    "ee_eo": {
      "eo_ml": 9,
      "eo_mr": 9,
      "eo_mt": 9,
      "eo_mb": 9,
      "i_cid":  "fee3481971000e_0",
      "eo_at": "Points scored",
      ⋮
      "i_wth": 450,
      "i_ht": 278.25
}}}
{
  "ty": "te",
  "id": "kix.iz8zrpzcxarv",
  "spi": 2
}
```

LISTING 3.1: Example element actions, adjust (`ae`) and tether (`te`)

Picture elements may be inserted from a given URL, uploaded from the local file system, or taken from the user's *Google Drive*. Insertion is similar to other page elements, containing a string placeholder, element properties, and a tether. Picture properties noted in the changelog include margins, width, height, source, ID, and element type. Contents of image source vary with the method of insertion: for local files, it is blank; for URL uploads, the external address is listed; and for drive uploads, source points to a temporary `googleusercontent.com` address. The temporary link hosted by Google's content delivery network(CDN) requires authorization to view, showing a forbidden 403 error otherwise. This URL saved to the changelog is only valid for about an hour until deactivation.

Suggestions are recorded in the changelog as regular revisions. Changing the edit mode to suggesting creates suggestions when altering the page. Each suggestion has a unique ID associated to it, and is recorded as a regular revision. Actions are tagged with **s** for suggestion types, such as **ste** (suggestion tether element) instead of `te` or **iss** (insert string suggestion) for `is`. All text entered or deleted in suggesting mode becomes a part of the permanent record of the document.

The layout engine colors any suggestion text, and text deletion ranges are indicated by a strike-through. Element insertion or removal contains colored borders, with the suggested action listed on the right side of the screen. Each collaborator has an associated color marking their changes. Suggestions are resolved by accepting or rejecting them on the page. Resolving a suggestion with either choice removes the it permanently, adding the necessary actions to the changelog for removal. Text ranges marked for deletion are unmarked, suggested insertion text is deleted, and any manipulated objects are returned to the original state. Upon accepting a suggestion, the actions are replayed in the log "normally" without the *suggestion* tag on each action.

Comments are the last embedded object with specific changelog behavior. Similar to suggestions, comments are considered page elements and are tethered to the page with an anchor and unique ID. However, the changelog contains only this reference to the *kix* object. Any comments and replies made on the page are not sent to the revision log. The editor sends a `POST /document/d/:id/`**`save`** request when adding revisions to the log, where **:id** is the unique document ID and edits are sent as form data. When adding comments or replies, however, a `POST /document/d/:id/sync` request adds the comments directly to the page using the comment API, without any trace left in the changelog. The server's response to a `sync` update is a list of all suggestions and comments associated with the current document.

While viewing revision history in the browser, users are given an option to "restore" a previous revision. This does not change any previous history in the changelog. Rather, the necessary steps to create a snapshot as of the reverted version are combined in a large transaction consisting of the type **rvrt**. This *revert* action contains a snapshot of the same format as *chunked snapshot*, a list of instructions required to create the document as of the listed revision.

3.3.4 Retrieval and processing

To facilitate acquisition of this revision log, we built a tool in *Python*, called `kumodocs`, which acquires the revision logs for a given document. The tool requires a user to sign in to a Google account, which creates the necessary OAuth2 authentication tokens required. The Google Drive API is then used to list all available documents, along with the available number of revisions. Revision logs are then retrieved using a `revisions/load` request as above with the authenticated process.

In order to simplify processing with existing tools, `kumodocs` parses the nested JSON result into a flat CSV format. Nested style dictionaries are merged into a single dictionary object. Each line contains a timestamp, user id, revision number, session id, session revision, action type, followed by the merged dictionary of key/value pairs involved in any modifications. Transactions are decomposed into separate actions, with each sharing the same identification information. This format is human-readable and easier to use text processing tools via the command line. The style modifications are encoded in dictionaries, which are easily parsed by *Python* or *JavaScript* to replay the events in a different editor.

The second benefit to creating an intermediate log format is an attempt at standardization. By writing tools that process this CSV instead of the raw revision JSON, we can generalize the process to any service that records fine-grain revisions. New services would require only a parsing tool to convert their protocol to the intermediate standard. This format we have suggested contains an arbitrary number of identification elements, followed by some action and a dictionary of instructions.

3.4 Images

Examining the HTML elements themselves in the page, we discovered some interesting behavior associated with images. The image element loaded on the page did not use the corresponding source URL in the changelog, but a different CDN link. Local uploads were referenced using HTML5's FileSystem API as (`filesystem:https://docs.google.com/persistent/docs/documents/:id/image/:i_cid`), where **:id** is the document ID and **:i_cid** is the image Cosmo ID. However, when viewed on another client, a similar link from `googleusercontent.com` was generated. The CDN link referenced by the image element did **not** require permission to view and seemed to be permanent, in contrast to the temporary link requiring authorization in the changelog. This link persisted even after the image was deleted from the document, with the image being available to anyone with the appropriate CDN address.

Image insertions were found to involve an API call to `createphoto`, which returns an image ID along with height and width parameters. An image copy is created in Google's CDN by *Cosmo*, the storage backend for *G Suite*. Another private API method was discovered with a document containing images - `renderData`. This request is similar to the `load` method used to retrieve the changelog:

```
https://docs.google.com/document/d/:id/renderdata?id=:id
```

The request body for this `renderData` call contains image Cosmo IDs in the form:

```
renderOps:{"r0": ["image",{"cosmoId":"1aC ... LTJs",
                             "container":"1UP ... uxk"}],
           "r1": ["image",{"cosmoId":"1MS ... HGM",
                             "container":"1UP ... uxk"}],
          ...
}

response:
)]}'
{"r0":"https://lh4.googleusercontent.com/kk15-B6fEX2i29PZs_uEh ... 2Cu",
 "r1":"https://lh4.googleusercontent.com/njSAZEAvDS2AAYaoPhMlY ... Wdl"}
```

Listing 3.2: Renderdata request body and response

`CosmoId` corresponds to the changelog field `i_cid` for every embedded image, and the `container` value was the same as the document ID. The `renderData` response contained a list of the same CDN links referenced by the inserted image HTML elements. We found these links to be accessible by anyone regardless of authorization.

We then tested access to these private API calls with Cosmo IDs obtained from the changelog. Posting a generic HTTP request was insufficient; redirection to a sign-in page was received. Using *Google*'s API through Python, we were able to send a `renderData` request with an authorized service, returning a list of accessible image URLs. Editing access to a document was found to be sufficient to access the CDN links, and once the URLs were obtained they could be viewed by anyone.

Since the image links returned were visible without authentication, we tested various configurations of the request to see if there were any security holes. Pictures were inserted into 2 documents; our test account had editing access to the first but no permissions for the second. Calling `renderdata` without access gave no results as expected. However, we found that replacing the `container` and `:id` to the authorized document but requesting unauthorized `cosmoIds` yielded a CDN link. Accessing this URL resulted in an *ACL* *denied* error, even for the account that uploaded the image; we were unable to find any meaningful use for this link, but its existance is strange. Following this test, used authorized an `container` and `:id` to obtain images from a different document; this was successful, returning the expected CDN-hosted images. This shows the origin is unimportant as long as the account sending the `renderdata` request has editing access to the document containing the `cosmoId`.

We conducted several experiments to determine the nature of the CDN links, as the permanent ones persisted even after image deletion. Two new photos were taken, ensuring no possibility of *Google* having previously cached these images. One photo was uploaded to *Google Drive*, and then added to a document; the other was embedded directly in the page from the local file system. Both images were deleted in the document, and the CDN-hosted links continued to be available without authentication for over a month.

The second experiment involved uploading two images in the same way, from *Google Drive* and the local system. However, instead of deleting the images within the document, the entire document was deleted. The images were no longer available through the CDN link after approximately 1 hour.

Our experiments show that, as long as *at least one* revision has a reference to an image, this link remains. They are accessible through `renderData` and the response is viewable by anyone. This behavior is forensically interesting, as recovering images thought long-deleted is possible while the original document remains. However, users generally expect that objects deleted from a document do not remain indefinitely.

The CDN links have extremely long identifiers of over 100 characters. The size and apparent randomness of the identifier makes it extremely unlikely to be guessed; `renderData` requires authorization to access the CDN links. This feature seems reasonable from a security standpoint, as images need to be kept as long as a user can restore a previous version containing that image. The identifier has no obvious relation to its container ID or user ID.

3.5 Slides/Drawings

Slides and *Drawings* utilize the same internal API as *Docs*, which transmits state through revision logs. The client interprets and renders these logs with a JavaScript layout engine similar to *kix* to produce each presentation or drawing. Using the same private `load` request as before, the logs were accessible. The overall structure was similar, with each revision corresponding to a single "action" array (which may be a transaction of several simultaneous actions).

```
{ "changelog":
[[ 4 ,[                                                  #  4:  transaction
   [ 12 ,"g1e...b_0_5",2,0,[1,"simple-light-2"]],   # 12:  Create slide/theme
   [ 18 ,["g1e...b_0_5"],[],[2,"TITLE_AND_BODY"]],  # 18:  Set layout
   [ 3 ,"g1e...b_0_6",108,                           #  3:  create text box
[2.8402,0.0,0.0,0.1909,12468.0,17801.0],[55,0,54,15],"g1e...b_0_5"]},
   [ 3 ,"g1e...b_0_7",108,
[2.8402,0.0,0.0,1.1388,12468.0,46099.0],[55,0,54,1],"g1e...b_0_5"],
   [ 3 ,"g1e...b_0_8",158,
[2.032,0.0,0.0,1.143,15252.0,27432.0],[55,0,54,16],"g1e...b_0_5:notes"],
   [ 3 ,"g1e...b_0_9",108,
[1.8288,0.0,0.0,1.3716,27432.0,1736.0],[55,1,54,1],"g1e...b_0_5:notes"]]],
                             148...381,"138...163",35,"10b...bb0",15,null],]
```

Listing 3.3: Creating a new slide, revision 35

The most important structural difference from previous revision logs is the lack of key values. Instead of being an array of JSON objects (key:value pairs), these logs are an array of arrays — further increasing the ambiguity of each value. Each array ended with the same identifying information: an epoch timestamp in Unix format, a unique 20-digit Google account ID, revision number, session ID, session number, and the same ending null value. We also found the granularity of changes to be higher, with each typed character having its own revision; pasted data would appear as a single revision, but regular typing was insufficient to produce insertion/deletion of 2+ characters in the same revision.

Using differential analysis, we examined how each value changed with varying page actions. The first value in each array corresponds to a type field, with major types noted as follows:

- **15** : inserting text in the box with the given ID
- **16** : deleting a range of text in the box with the given ID
- **3** : Text box creation with coordinate data for positioning
- **12** : slide creation
- **4** : Transaction operation (multiple elementary actions)

Listing 3.3 and Listing 3.4 show several entries in a *Slides* revision log from version 36 to 44. A new slide is created, causing a transaction ([4]) of simultaneous actions. Revision 35 is a single array with type **4** containing 6 nested actions. Types **12** and **18** initialize the slide canvas with the corresponding theme (12) and layout (18), followed by creation of the default text boxes with action **3**. The last element denotes the parent of each object, which is the slide ID `g1e...b_0_5` in this case. Title and subtitle boxes are the first two text boxes created, followed by text box creation for speaker notes; default boxes created vary depending on layout selected.

Each slide and text box has a unique ID associated with it, as well as a parent container. Text or objects are inserted directly using this ID. Actions creating or moving objects with size contain a 6-element array describing the (x, y) position, orientation (text boxes may be rotated), and scalar size.

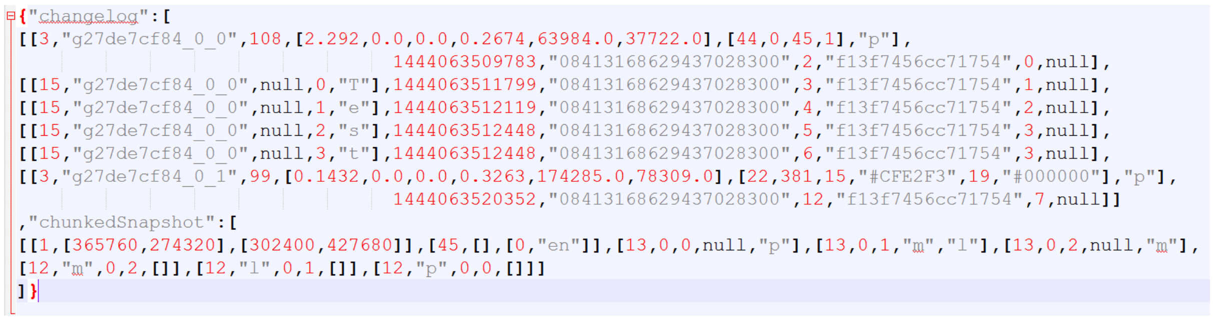
```
[[15,"g1e...b_0_7",null,0,"A"],148...381,"138...163",36,...],
[[15,"g1e...b_0_7",null,1,"d"],148...381,"138...163",37,...]},
[[15,"g1e...b_0_7",null,2,"d"],148...381,"138...163",38,...]},
[[15,"g1e...b_0_7",null,1,"i"],148...381,"138...163",39,...]},
[[15,"g1e...b_0_7",null,2,"n"],148...381,"138...163",40,...]},
[[16,"g1e...b_0_7",null,2,3],148...381,"138...163",41,...]},
[[16,"g1e...b_0_7",null,1,2],148...381,"138...163",42,...]},
[[15,"g1e...b_0_7",null,1,"e"],148...381,"138...163",43,...]},
[[15,"g1e...b_0_7",null,2,"d"],148...381,"138...163",44,...]}
```

Listing 3.4: Typing `Added` with 2 backspaces into `g1e...b_0_7`; ID data trimmed

In Listing 3.4, we have a user typing `"Addin"`, followed by `2 backspaces` and then `"ed"`. Inserted text is highlighted in yellow, and deletion ranges are highlighted in red. The second value of the array, **g1e...b_0_7**, is the target container; this corresponds to the subtitle box, as it was the second text box created in slide creation transaction. Insertion does not require the slide ID which holds the container, as membership is established at time of creation or when an object is moved.

Duplicating slides is a large transaction; there is no single "copy" command. Rather, the entire page is recreated in the same theme and layout, followed by text box creations for each text box on the old slide. Any text currently displayed is pasted into the boxes in the appropriate format. These new objects are given unique IDs and the new slide is added to the list of top-level slides in the correct order.

Theme modification is a similar transaction to duplication; however, the creation of each element involves style changes consistent with the requested theme. After each element is duplicated in the new theme, the old page has every existing text box deleted, and then the slide itself is removed. Deleting a slide follows this same procedure, where the slide itself is cleared of all contained objects before being removed itself. The slide itself is recreated in the requested layout and theme, followed by construction of each existing text box and insertion of existing text. These new objects are assigned unique IDs; in



```
{"changelog":[
[[3,"g27de7cf84_0_0",108,[2.292,0.0,0.0,0.2674,63984.0,37722.0],[44,0,45,1],"p"],
                                    1444063509783,"08413168629437028300",2,"f13f7456cc71754",0,null],
[[15,"g27de7cf84_0_0",null,0,"T"],1444063511799,"08413168629437028300",3,"f13f7456cc71754",1,null],
[[15,"g27de7cf84_0_0",null,1,"e"],1444063512119,"08413168629437028300",4,"f13f7456cc71754",2,null],
[[15,"g27de7cf84_0_0",null,2,"s"],1444063512448,"08413168629437028300",5,"f13f7456cc71754",3,null],
[[15,"g27de7cf84_0_0",null,3,"t"],1444063512448,"08413168629437028300",6,"f13f7456cc71754",3,null],
[[3,"g27de7cf84_0_1",99,[0.1432,0.0,0.0,0.3263,174285.0,78309.0],[22,381,15,"#CFE2F3",19,"#000000"],"p"],
                                    1444063520352,"08413168629437028300",12,"f13f7456cc71754",7,null]]
,"chunkedSnapshot":[
[[1,[365760,274320],[302400,427680]],[45,[],[0,"en"]],[13,0,0,null,"p"],[13,0,1,"m","l"],[13,0,2,null,"m"],
[12,"m",0,2,[]],[12,"l",0,1,[]],[12,"p",0,0,[]]]
]}
```

FIGURE 3.9: *Drawings* revision log excerpt, single text box after `"Test"` is typed

the case of layout or theme change, this sequence involves deleting all previous elements of the old style.

We found *Drawings* revision logs to be a simplified version of *Slides* as shown on Figure 3.9 These logs were available through a similar `load` request. Each drawing is built up from a series of elementary actions like both documents and presentations. Drawing a line or curve is represented by vector instructions in the log, and previous versions are available for viewing or reverting. *Drawings* provides the same drawing functionality available within *Slides*.

3.6 Sheets

Repeating our methodology with *Sheets*, we monitored client/server network interactions to model the behavior and dissect the underlying protocol. When working with a spreadsheet, the client can be seen to communicate with the server after every entry into a cell. Revisions are updated and the saving process is visible; indeed, *Sheets* supports a similar versioning system to *Google Docs* and *Slides*. `Save` and `bind` requests following each transaction were observed similar to earlier protocols. We also observed noticeably different methods for obtaining specific revisions from the server.

Instead of obtaining the revision log and processing it locally, the client obtains a browser-ready HTML document through a **showrevision** request. This document contains the value of every cell to be displayed, without any underlying equation or structural elements; in other words, a snapshot of values as of that revision. As spreadsheets might contain complex equations, this architecture makes sense to ensure a speedy and responsive product. Replicating this revision request from start to finish would result in a series of snapshots that give adequate history, but ultimately cell relations would have to be inferred with no equation data.

The *Sheets* protocol still retained many of the functions of the previous services we studied, including incremental saving and sending what appeared to be revision updates. Using the same **load** request on a spreadsheet *did* result in a revision log. Like all G-suite revision logs we discovered, each revision contains a single object with the standard identifying information. This object, like the *Docs* log, contains key-value pairs; however, the keys are simply enumerated, resulting in a form similar to *Slides* filled with unknown values.

```
[[21299578,[null,[null,"0",6,7,3,4],[null,6445244,3,[null,2,"Test D7"],
    null,null,0],[null,null,[[null,null,513,[null,0],null,null,null,
    null,null,null,null,null,0]]]]],
    1490565260116,"13862709383388366163",6,"4175a87f0cbe8cdc",3,null],
[[21299578,[null,[null,"0",6,7,4,5],[null,6445244,3,[null,2,"Test E7"],
    null,null,0],[null,null,[[null,null,513,[null,0],null,null,null,
    null,null,null,null,null,0]]]]],
    1490565265253,"13862709383388366163",7,"4175a87f0cbe8cdc",4,null]]
```

LISTING 3.5: Sheets revision log, typing `Test D7` and `Test E7`

Like the other logs, the first value (`21299578`) in each revision corresponds to a type. The highlighted area denotes coordinates as cell ranges. Each pair denotes the starting cell (inclusive), and the ending cell (not inclusive) for rows and columns; numbering starts at 0. Both revisions occur on row (6,7), which is a 1-cell-wide entry at row 7. The column range is in blue; for the first revision, (3,4) indicates the D column and (4,5) the E column. Together, this indicates the first revision occurs at cell **D7** and the second at **E7**. Many of the null values are placeholders for equation data, arguments, and target ranges.

```
[[21299578,[null,[null,"0",4,5,5,6],[null,6445233,12,null,0,0],[null,
[[null,[null,[[null,4,null,null,null,null,3],[null,2,null,"AVERAGE",1],
[null,1,"="]]],[null,[[null,2,"=AVERAGE("],[null,3,null,3],
[null,2,")"]]],"R3]FAVERAGE:1]S"]],null,[[null,[[null,
[null,-1,5,-3,-1,1118464],3]]]]]]],
    1490567192549,"13862709383388366163",8,"20f0c0d28d7bd6b8",0,null]]
```

Listing 3.6: F5=AVERAGE(C4:D9) entered in a spreadsheet

In Listing 3.6, the equation `AVERAGE` has been entered into cell (4,5,5,6), which is `F5`. The equation targets noted in yellow and blue are relative distances from the source. Yellow denotes the row range, and blue denotes the column range as before; (-1,5) is a 6-row range starting 1 cell above F5 (4) and ending 5 cells below F5 (10), corresponding to rows 4-10. Highlighted in blue, (-3, -1) is a 2-column span starting 3 cells to the left (column C) and ending 1 cell to the left (column E). As the ending points are not inclusive, the column ends at **D** and the row ends at **9**. This translates to `AVERAGE(C4:D9)`.

The granularity is extremely low; to keep communication to a minimum, revisions are only updated after each text box changes. Typing or deleting characters within a cell do not trigger any actions. This simplifies the client/server API, as the main communication is simply which ranges need to be updated and the new values instead of granular character deltas.

3.7 Forms

Forms is *Google*'s collaborative platform for creating online surveys, quizzes, and web input forms. We were able to recover revision logs in the same manner as the other services. The structure remained similar with a `chunkedSnapshot` and `changelog`; however, the protocol was quite different. Each question is sent as a bundle - like sheets, commit revisions are stored rather than incremental revisions. Questions are bundled together with all components; any edit in the question has the entire bundle sent. When any part of a question is edited, the entire self-contained bundle is sent. All revisions become a part of the permanent history.

```
[[24479122,24479122,"[0,[216878110,"Test question",null,2,
  [[1499080989,[["Option 1",null,null,null,0],["",null,null,null,1]],
                0,null,null,null,null,null,0]]]]"],
[[24479122,24479122,"[0,[216878110,"Test question",null,2,
  [[1499080989,[["Option 1",null,null,null,0]],
                0,null,null,null,null,null,0]]]]"],
[[24479122,24479122,"[0,[216878110,"Test question",null,2,
  [[1499080989,[["Option 1",null,null,null,0],
                ["Option 2",null,null,null,0]],
                0,null,null,null,null,null,0]]]]"],
[[24479122,24479122,"[0,[216878110,"Test question",null,2,
  [[1499080989,[["Option 1",null,null,null,0],
                ["Option 2",null,null,null,0],
                ["Option 3",null,null,null,0]]
                ,0,null,null,null,null,null,0]]]]"],
[[24479122,24479122,"[0,[216878110,"Testing question",null,2,
  [[1499080989,[["Option 1",null,null,null,0],
                ["Option 2",null,null,null,0],
                ["Option 3",null,null,null,0]]
                ,0,null,null,null,null,null,0]]]]"],
```

Listing 3.7: Forms revision log; identification data omitted for brevity

In Listing 3.7, we have a question titled `Test question`, to which answers are being added. Like other services, the beginning of each revision consists of a coded action type, followed by the same identification data at the end. Each answer is titled `Option x`; every time a new answer is added, the entire question bundle is sent again. While other services had various methods to view prior revisions or revert to earlier versions, forms has no revision information in the editing section.

The log is recoverable in the same fashion as with other services, using a private API:

```
https://docs.google.com/forms/d/<form_ID>/revisions/load?start=<start>
&end=<end>&token=<auth_token>
```

For *Forms*, this information on how the questions evolve over time is unavailable anywhere else. Google's *Forms* API is limited to basic actions like creating new forms and adding questions of various types. There are no means to revert to previous revisions.

Responses are viewable as a *Sheets* spreadsheet, which tracks responses and questions over time. This gives a weak version history, as every sheet revision contains a full list of questions as of the time of the answers. However, the way in which the questions change over time is only obtainable from the revision log. This gives roundabout access to "major" revisions; however, intermediate changes that occur between two responses are lost.

3.8 Sites

Sites is the first collaboration solution Google offers that shows a major break from the API underlying all other services. We were not able to recover a revision log in the same fashion; however, this was available through the public API as an *Atom* XML feed. *Sites* offers easy creation and management of websites hosted by Google. This complexity necessitates being developed from the ground up, with functionality unique to websites.

Sites offers commit versioning, with each edit in a webpage being saved as a separate version. Granularity is low, only "snapshots" of each edit are saved. However, it offers availability to all "minor" revisions, unlike most other services. Each page contains its own revision history, which can be easily controlled from the admin panel. Any revision may be reverted to easily. Functionality is similar; in effect, the revision log is available from your admin panel. Any reverts are appended to the list, ensuring permanency of any completely saved page edits.

Sites does borrow the comment backend from the other *G Suite* products. Each page has an optional comment section, which is updated in the same way as others - a **bind** request. There is an extensive public API available, allowing access control changes, content modification, site modification, and listing revisions. Comments are not available through the public API; however, we were able to reverse engineer the protocols used by *Sites* to access the comments. When a *Sites* page is retrieved, a `comment pane` is created at the bottom of the page:

```
new sites.CommentPane('//docs.google.com/comments/u/0/d/:id/api/js...)
```

Listing 3.8: Comment pane creation in *Sites* response

This **:id** is a *container* which holds the comments for that page. When used in place of the `fileID` in the *Google Drive* API, comments and replies were available. The comment resource returned is the same as the other services. This was not a real file; trying to access the metadata of this **:id** returned nothing.

3.9 Suggestions and comments

Docs presents 2 options for collaboration - editing mode, in which changes are instantly applied and recorded in the revision log, and suggestion mode. Each collaborator has an associated color; their suggestions are colored edits made directly in the page. Deletion is represented by a strikethrough style in the personal color. These edits are recorded in the changelog similar to regular revisions with special suggestion insert and delete actions. Collaborators with commenting or editing permission may make suggestions; viewing access is insufficient to make suggestions or comments.

In Listing 3.9, `Tesr` is typed in suggestion mode at index 13 in the document. A single character is deleted (**dss** @15-15), and **t** is added to form `Test`. This is followed by a non-suggestion delete (**ds**) from index 627 to 723. Mark section for delete (**msfd**) is a suggestion to delete existing text in the document, which appears as a strikethrough markup on the page. *Docs* is currently the only service that supports suggestions.

```
"changelog":[
[{"ty":"iss","sugid":"suggest.b5e3j06tc287","s":"Te","ibi":13},
  148...779,"030...951",291,"4a9...sc98",2,null],
[{"ty":"iss","sugid":"suggest.b5e3j06tc287","s":"sr","ibi":15},
  148...522,"030...951",292,"4a9...sc98",3,null],
[{"ty":"dss","si":15,"ei":15},
  148...831,"030...951",293,"4a9...sc98",4,null],
[{"ty":"iss","sugid":"suggest.b5e3j06tc287","s":"t","ibi":15},
  148...462,"030...951",294,"4a9...sc98",5,null],
[{"ty":"ds","ei":733,"si":627},
  148...146,"181...856",295,"6c5...s413",0,null],
[{"ty":"msfd","ei":806,"si":627,"sugid":"suggest.no82u2ec09kq"},
  148...864,"181...856",296,"6c5...413",1,null]]
```

Listing 3.9: Suggestion example, revisions 291-296. Insert in yellow, delete in red

Comments are supported by all collaborative services offered by *G Suite* except *Forms*. Each service interacts with the internal comment API backend in the same manner. There are slight differences in how the initial comment is registered in the revision log:

- *Docs*– Anchor is noted along with the corresponding *kix* ID.
- *Sheets*– The comment is noted by cell location, no ID information is sent.
- *Slides*– No revision log change.
- *Forms*– No revision log change.

```
{"ty": "as",
 "st": "doco_anchor",
 "si": 275,
 "fm": false,
 "sm": { "das_a": { "cv": {"op": "insert", "opIndex": 0,
                           "opValue": "kix.b2nji6cpuond"}
      }
     },
 "ei": 289
}
```

Listing 3.10: Creating a new comment in *Docs*

For all services, comments are not tied to any revisions; they are simply attached to some resource. Direct comment access is provided through the public API for *Docs*, *Sheets*, and *Slides*; *Forms* is the only service that has no comment support. As noted in section 3.8, we were able to retrieve comments from *Sites* by discovering the `comment pane` is virtualized as a *Google Drive* file, which contains all of the page's comments. Attempting to create new replies or comments was met with authorization issues.

3.10 Recoverable Information

Since the revision log contains a complete and detailed history, a wealth of information is recoverable. Even when reverting to previous versions, the old history is left intact due to the append-only nature of the log. Sharing a *G Suite* application for collaboration is, in effect, revealing the entire history. Images thought long-deleted are recoverable as long as the document exists; anyone with editing permission may recover these images.

In documents, suggestions are merely direct document edits that have been marked-up; there is no API support for retrieving or viewing past suggestions. Once they are accepted or rejected, that piece of history is lost from any document snapshot. The change becomes a part of the document, and details about which user made the suggestion and how that suggestion evolved over time are lost. However, this becomes a permanent part of the revision log, which our tool can reconstruct any suggestion history.

Comments are supported among all of the *G Suite* services except *Forms*; they all use the same backend. Updates are sent through the API, which returns a list of all comments existing for that application. These updates largely bypass the revision log; initial comment creation behavior is described in section 3.9. The drive API, however, provides a way to recover all comments on files with at least commenting permissions. Deleted comments are stripped of content but all metadata remains, including time of deletion, creation time, and author.

Listing 3.11 shows an example of deleted comment metadata. Creation and modification (deletion) times are noted in bold. Words highlighted on the page when creating the comment remain even after deletion. The resolved field shows this comment was initially resolved, and then deleted at a later time; this would be false if the comment was deleted before being resolved. Active comments have the same form, with the content of the comment and replies appearing in the highlighted areas.

```
"comments": [
{"resolved": true,
 "kind": "drive\#comment",
 "modifiedTime":  "2017-03-19T21:57:54.732Z"},
 "author": {"me": false, kind": "drive#user", "displayName": ***,
            "photoLink": "//lh3.googleusercontent.com/***/photo.jpg"},
 "deleted":  true,
 "quotedFileContent":  {"value":  "on to google dr"},
                         "mimeType": "text/html"},
 "htmlContent":  "",
 "anchor": "kix.b2nji6cpuond",
 "content":  "",
 "replies": [{
     "kind": "drive\#reply",
     "modifiedTime":  "2017-03-19T21:41:53.196Z",
     "author":{"me": false, "kind": "drive\#user", "displayName": "***",
               "photoLink": "//lh3.googleusercontent.com/***/photo.jpg"},
     "deleted":  true,
     "htmlContent":  "",
     "content":  "",
     "action": "resolve",
     "createdTime":  "2017-03-19T21:41:53.196Z",
     "id": "AAAABCDWzIE"}],
 "createdTime":  "2017-03-19T21:41:23.115Z",
 "id": "AAAABCDWzIA"}
```

Listing 3.11: Deleted comment metadata

Sharing any document for collaboration effectively shares the *entire* history. Even partial permissions, like comment-only, reveal all comments made or deleted since creation. Images and *Drawings* are able to be recovered for as long as the original file exists. The only way to permanently erase previous history is by **cloning**, which creates a new file and inserts the most recent snapshot. This allows collaboration on a file containing sensitive history; however, erasing the revision log also removes the ability to revert to previous revisions. Any new edits made in the cloned document become a part of the revision log as normal.

Chapter 4

Kumodocs

Our proof-of-concept tool, `kumodocs`, was designed to acquire, process, and store *G Suite* cloud-native artifacts. These artifacts are retrieved using private API calls and transformed into a standardized log format. Plain text, images, *Drawings*, comments, and suggestions are extracted from the intermediate log. New services tare easily added, requiring only a module to retrieve and parse logs.

4.1 Installation

Installation and use has been tested on Windows 7, Windows 10, Ubuntu 14.04 LTS, and Ubuntu 16.10 LTS. The source code is platform independent; it should work on any operating system that supports `Python 2.7`. Python comes installed on many Linux distributions and Mac OS X Sierra. Installers are provided for other Mac OS X and Windows operating systems. [13]

4.1.1 Requirements

- Windows or Linux machine
- Python 2.7.x
- Python packages:
  - Pip

  - google-api-python-client 1.6.2
  - httplib 0.10.3
  - oauth2client 4.0.0
- App registered for Google API use at https://console.developers.google.com
- Google account credentials

4.1.2 Python 2.7

Many Linux distributions come with `Python27`. The exact version of Python can be checked with the command line for any OS:

```
shane@ubuntu:~$ python --version
Python 2.7.12
```

Listing 4.1: Checking python version

If the proper version of Python is not available, download the latest 2.7.x installer for Mac OS X or Windows from https://www.python.org/downloads/. Any minor version of Python will suffice; the latest version of 2.7.x will contain the most up-to-date bug fixes. For older Ubuntu systems with Python ≤ 2.6:

```
# Install dependencies
sudo apt-get install build-essential checkinstall
sudo apt-get install libreadline-gplv2-dev libncursesw5-dev libssl-dev
   libsqlite3-dev tk-dev libgdbm-dev libc6-dev libbz2-dev

# Download Python2.7
cd ~/Downloads/
wget https://www.python.org/ftp/python/2.7.12/Python-2.7.12.tgz

# Extract
tar -xvf Python-2.7.12.tgz
```

```
cd Python-2.7.12

# Install
./configure
make
sudo checkinstall
```

Listing 4.2: Installing Python 2.7 on Ubuntu

4.1.3 Kumodocs

Kumodocs is available at the following Github repository:

```
git clone https://github.com/kumofx/kumodocs.git
```

Listing 4.3: Cloning kumodocs repository

The code can also be downloaded as a zip file if `Git` is unavailable:

```
https://github.com/kumofx/kumodocs/archive/master.zip
```

Listing 4.4: Downloading zipped contents without Git

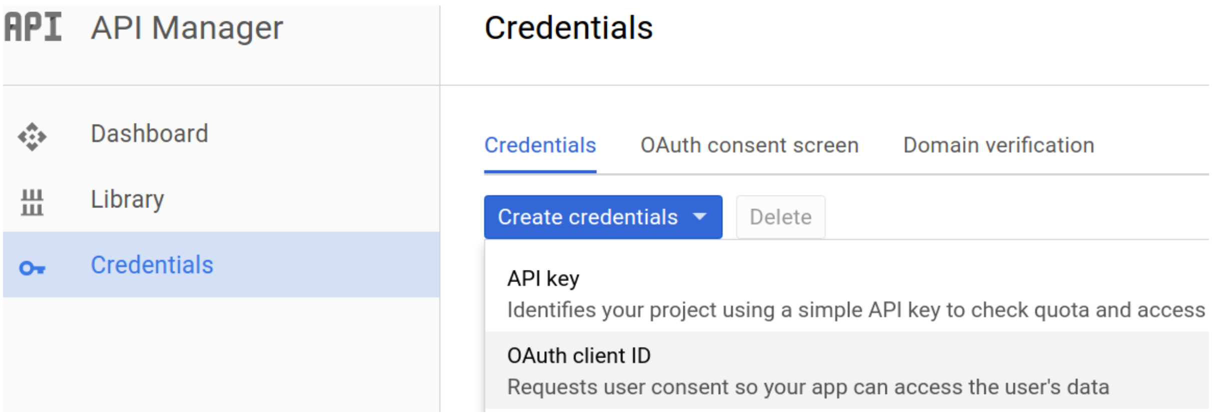
4.1.4 Python packages

The text file `requirements.txt` contains all necessary packages. This can be installed with the python package manager `pip`:

```
pip install -r requirements.txt
```

4.1.5 Configuration

`Kumodocs` requires a client ID and client secret to be saved in `config/gdrive_config.json`. These can be obtained by creating a project at https://console.developers.google.com. Once a project is created, OAuth client ID can be generated on the credentials tab as shown in figure 4.1.

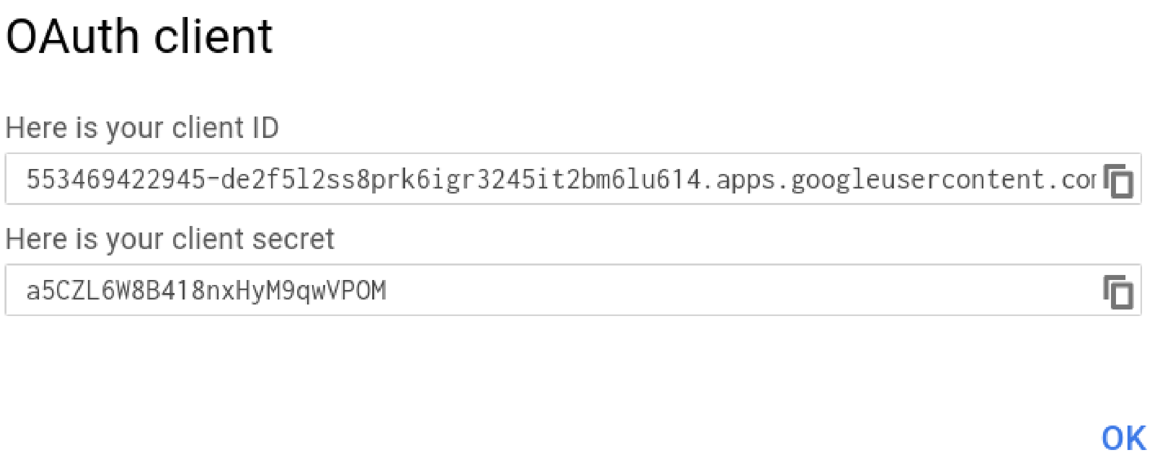


API key
Identifies your project using a simple API key to check quota and access

OAuth client ID
Requests user consent so your app can access the user's data

FIGURE 4.1: Creating OAuth client ID and secret

Choose "other" for app type, name the client, and click create to generate the credentials. Figure 4.2 shows a successful creation. The secret and ID need to be added to `kumodocs/config/gdrive_config.json` to allow the client API access. Running `kumodocs` without the appropriate configuration will prompt the user to enter this client secret and ID.



Here is your client ID
553469422945-de2f512ss8prk6igr3245it2bm6lu614.apps.googleusercontent.coı

Here is your client secret
a5CZL6W8B418nxHyM9qwVPOM

FIGURE 4.2: Successful creation screen

4.2 Use

Once `kumodocs` is configured, it may be used from the command line.

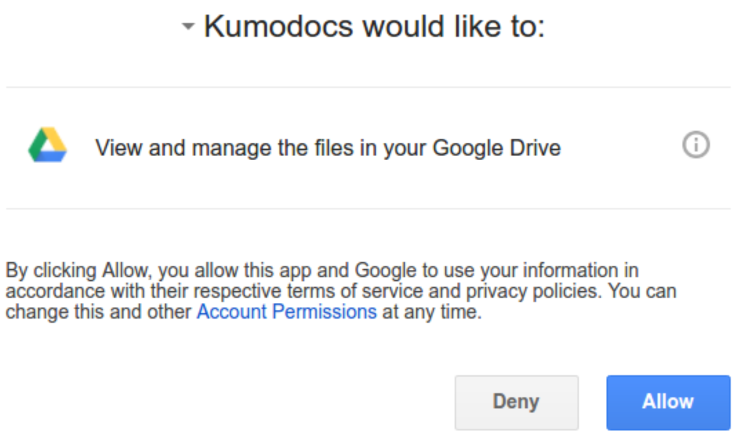
```
python kumodocs.py
```

When `kumodocs` launches, it checks for any saved *Google* credentials. If none are found, the authentication process will open a browser and ask the user to sign in to their account.

```
python kumodocs.py
Your browser has been opened to visit:

    https://accounts.google.com/o/oauth2/auth?scope=https%3A%2F%2Fwww...
```

Figure 4.3 shows the browser request. Once verified, the appropriate tokens are saved in the `config/` folder for future use. Used without any arguments, `kumodocs` will guide the user to choosing a *Document* or *Slide*. A file dialog is opened, virtualizing the contents of the user's *Google Drive*. Each service is a folder, which contains the file names of all files within that service.


Kumodocs would like to:

View and manage the files in your Google Drive

By clicking Allow, you allow this app and Google to use your information in accordance with their respective terms of service and privacy policies. You can change this and other Account Permissions at any time.

FIGURE 4.3: Requesting authorization

Choosing a file will start the appropriate service and retrieve the revision log according to user input. The user choses a revision range, with the maximum number of revisions being displayed. Once a range is chosen, the revision log is fetched, parsed, and the output is saved in `./downloaded/gdocs/<file>_<start>-<end>/`.

```
Chose file: test-doc
Start from revision(max 108): 1
End at revision(max 108): 50
Finished with output in directory ./downloaded/gdocs/test-doc_1-50/
```

LISTING 4.5: Example use case, selecting file `test-doc` from revision 1 to 50

The output directory contains several files depending on the service –

- *Docs*: Any deleted images or *Drawings*, comments, suggestions, and a plain-text snapshot as of the ending revision.
- *Slides*: Any deleted images or *Drawings*, comments, and a plain-text snapshot as of the ending revision. This snapshot is rendered as a tree structure, with a folder for each slide and a text file for each text box.
- *Sites*, *Forms*, and *Sheets*: Only the revision log is obtained.

While `kumodocs` can acquire the revision log for any collaborative service saved in *Google Drive*, we currently only fully process artifacts obtained from *Docs* and *Slides*. Once a parser is finished for the other services, we can obtain any deleted images, *Drawings*, and comments in the same manner.

4.3 Quill playback

In order to support independent playback and archival of *Docs* artifacts, we developed a playback feature using the Quill[1] open-source editor. This web app, written in JavaScript, takes a revision log in intermediate format as input. Each line of the log is replayed in the editor, showing how the document evolved over time. Varying speeds are supported for playback, and a slider allows jumping to any revision. Since this only needs an intermediate log as input, any service could be used as long as the format is properly followed.

4.4 Google Drive API

Our tool uses a number of *Google Drive*'s public and private calls to retrieve data. Undocumented methods are used to retrieve images and revision logs.

4.4.1 Public Drive API

Each requires an authenticated service object, and returns a resource that must be sent by appending `.execute()`.

- **Build service**: service = build("drive", "v2", http=http)

  Builds an authenticated drive service for API calls.

- **list files**: files().list(q=search_param[drive_type], fields='items(title, id)')

  The `fields` parameter filters any response to the specified fields, reducing unnecessary data being sent. The `q` parameter specifies which drive type should be listed.

- **list revisions**: revisions().list(fileId=file_id, fields='items(id)')

  This lists the major revisions of a file; the last major revision denotes the upper limit for log requests.

- **list comments**: comments().list(**params)

  Params is a dictionary containing fileId, includeDeleted, and fields='comment_fields'.

- **File title**: files().get(fileId=file_id, fields='title')

  Get returns all metadata associated with file_id filtered to items listed in fields.

4.4.2 Private Drive API

`Kumodocs` uses private calls to obtain logs and images not available through public means.

- **Revision log**: /{service}/d/{file_id}/revisions/load?start={start}&end={end}

  Returns the log as described in subsection 3.3.2, consisting of a `chunkedSnapshot` and `changelog`

- **Image URLs**: /{service}/d/{file_id}/renderdata?id={file_id}

  Requires authorization to execute; returns image URLs accessible to anyone.

Renderdata requires the following form data:

```
renderOps:{"r0": ["image",{"cosmoId":"1aC ... LTJs",
                           "container":"1UP ... uxk"}],
           "r1": ["image",{"cosmoId":"1MS ... HGM",
                           "container":"1UP ... uxk"}],
          ⋮
}
```

Listing 4.6: Renderdata form data

These requests are sent by accessing the httplib2 object attached to the authenticated service.

```
service._http.request(render_url, method='POST', body=body, headers=headers
   )
```

Listing 4.7: Constructing a renderdata request

4.5 Intermediate format

Each revision log contains transactions, groups of elementary actions simultaneously executed, which create deeply nested structures. The intermediate format flattens this log, separating each transaction component into its own line and given the same identification data. Keys are substituted to a somewhat human-readable form; this mapping can be changed as desired. A vertical bar (**|**) is used to separate values due to the appearance of other common delimiters in the source data; this is an optional parameter to the parser that can be changed. The first field contains a timestamp, followed by a Google ID, revision number, session ID, session revision, and action type. The last line of the revision log contains a dictionary of style modifications and indices to be used to carry out the requested action.

```
144...029|181...856|403|3e1|...c20|46|ins|{"ins_index": 131,
   "string": " two", "type": "is"}
144...029|181...856|403|3e1|...c20|47|ins|{"ins_index": 135,
   "string": "n", "type": "is"}
144...029|181...856|403|3e1|...c20|47|adj|{"end_index": 135, "type": "as",
   "style_type": "paragraph", "start_index": 135, "style_mod": {...}
144...029|181...856|403|3e1|...c20|47|adj|{"end_index": 135, "type": "as",
   "style_type": "list", "start_index": 135, "style_mod": {...}
144...029|181...856|403|3e1|...c20|47|del|{"start_index": 135,
   "end_index": 135, "type": "ds"}
144...029|181...856|403|3e1|...c20|47|del|{"start_index": 133,
   "end_index": 134, "type": "ds"}
```

Listing 4.8: Transaction of 5 actions at revision **403**

Each timestamp is the same, ending in ...029. Insert actions are highlighted in yellow, while delete is in red. All actions occur in the transaction containing revision **403**, and this is the 47th edit in session `3e1...c20`. The style modification dictionaries were shortened due to size constraints, with each containing 11-20 key-value pairs.

4.6 Summary

Kumodocs was designed with abstraction in mind in order to accomodate any collaborative service that stores fine-grain revisions. Other services such as `Zoho Writer` or `Dropbox Paper` would only need a driver to retrieve the revision log and a parser converting this log to the intermediate format used by `kumodocs`. Any revision log converted to the intermediate format will also have playback support.

Chapter 5

Discussion

Our study of *G Suite* protocols was motivated by the shortcomings of adapting traditional forensics to the cloud. Previous work focusing on recovery of trace client data had found mostly evidence of cloud use and occasionally cached data, but few artifacts of actual use. We were able to obtain *cloud-native artifacts* in the form of revision logs from every service except for *Sites*; however, similar information is available in the admin control panel. These cloud natives contain the entire editing history in an append-only form, which prevents spoiling of evidence. In addition to the revision logs, we were able to recover deleted images, *Drawings*, and comment information. These artifacts were available by any user given editing permission to a file.

Even when reverting to previous versions, the old history is left intact due to the append-only nature of the log. Sharing a *G Suite* application for collaboration is, in effect, revealing the entire history. Even partial permissions, like comment-only, reveal all comments made or deleted since creation. Images and *Drawings* are able to be recovered for as long as the original file exists. The only way to permanently erase previous history is by **cloning**, which creates a new file and inserts a snapshot of the most recent revision. This allows collaboration on a file containing sensitive history; however, erasing the revision log also removes the ability to revert to previous revisions. Any new edits made in the cloned document become a part of the revision log as normal.

We found many commonalities in the underlying internal API used by each of the collaborative services. The overall structure of the append-only revision logs was similar in each case: a single array for each revision (which could be nested with multiple actions in a single transaction), ending with a timestamp, the editor's *Google* account ID, revision number, session ID, session revision number. Each array begins with an action type, which indicates how to process the rest of the revision. Differing services handled the rest of the protocol implementation in varying methods; *Docs*, contains abbreviated key-value pairs, while other services are cryptic lists of numbers.

The granularity of changes varies between committed revisions and incremental revisions. In *Slides*, revisions are stored as often as each character; *Docs* often contains clusters of 1-3 characters in revisions. The remaining services have some method of editing fields (cells for *Sheets*, questions for *Forms*, website for *Sites*) that are only saved upon committing changes.

We noted unexpected behavior in the way *Google* stores images. Any images or *Drawings* that are deleted from a file are still recoverable as long as the file still exists. From an application design perspective, some reference would need to be kept in case of reverting to a version before deletion. However, most users would not expect these images to stay forever; even less would expect the entire editing history being shared upon collaboration, including the ability to recover these lost images. These images are also available without authentication; retrieving the CDN links requires proper permission, but may be shared with anyone once obtained. Cloning or deleting a file effectively clears the history and all references to previous images. Even in this case, we found images to remain available in Google's CDN for about an hour before removal. Hastily deleted documents would also have a narrow window of recovery – these images might be recoverable by combining memory and browser forensics with our tool.

Our results show that reverse engineering is likely to be a necessary part of future SaaS analysis, focusing on network protocols. The most important artifacts of cloud services are unavailable on the client, which further validates our motivating concerns. However, being able to monitor all network traffic, create JavaScript and XHR breakpoints, and instrument crucial junctures gives an advantage over traditional reverse engineering. Public API acquisition is incomplete without the acquisition of *cloud-native artifacts* that contain valuable state data unavailable otherwise.

Complexity increases when private calls obfuscate inner workings and the client is presented with the result of server computation. Offloading much of the computation to a centralized servier is great for offering a consistent experience, increasing browser responsiveness and speed. As computing power continue to improve, however, we would expect to see a proportionate amount of work being relegated to browser and client with the same experience maintained. It is likely that reverse engineering efforts will become easier as clients shoulder more of the computation.

Recovering these cloud natives also allows archival of evidence. Among the services we analyzed, there are no reliable means to store files on the local system. Snapshots are available at certain points in time, but this misses revision history, deleted images, comments, and suggestions. Protocols may also change over time, and there are currently no ways to "upload" a revision log to create a new document. Clearly, there is a need for tools to recover, process, and store, and playback these artifacts independently of the native SaaS application.

G Suite collaborative services all use the same API backend, as we can see in the shared behavior in retrieving revision logs and comments. These logs, and the storing of fine-grain revisions, come about as an optimal collaborative editing solution. Similar patterns can be seen in each of the collaborative solutions offered in regards to revisions. This can be seen in other major collaborative services, showing that *G Suite* is fairly representative. We find evidence of incremental revisions of similar structure being sent in *Zoho Writer*, *Dropbox Paper*, and *Microsoft Word Online*. Through a private method, we were able to obtain 25 revisions at a time from *Zoho Writer*'s log, which can be used to build the total contents; however, we did not easily find a means to retrieve editing logs from the other services.

Chapter 6

Conclusion

We showed that traditional client-based forensic approaches were insufficient when applied to cloud environments. Previous work recovered few artifacts of use, as the most important data structures are internal to the web/SaaS applications. This client-centric approach is inherently deficient since it cannot account for *cloud-native artifacts*, internal data structures that maintain the persistent state of web applications. These artifacts have no presence on the client, and are maintained solely in the cloud. By reverse engineering *G Suite* collaborative protocols, we were able to recover these cloud natives in the form of revision logs detailing the complete editing history of a file.

We analyzed protocols in the five online collaboration services offered by *Google*: *Docs*, *Slides*, *Forms*, *Sheets*, and *Sites*. Strong commonalities were found in all services, storing a log of complete editing history. Recovering these revision logs, combined with API-based acquisition, produces a more complete solution for cloud drive forensics. Our proof-of-concept tool, `kumodocs`, is able to acquire, process, store these logs with playback support for *Docs*.

We greatly expanded Somer's initial work on *Google Docs* protocols, analyzing the nature of the *chunked snapshot* and *changelog* in great detail. This was extended to all collaborative services offered by Google, in which we found great similarities in the structure of these revision logs and the internal protocols used by these services. By processing these logs, we were able to recover vast amounts of deleted data from *Docs* and *Slides*, including evidence of previous suggestions and any comments made. Deleted comments have their content stripped, but still contain a wealth of metadata – any conversations occurring through comments and replies can be associated to each author with timestamps given for creation and deletion. Other SaaS/web applications are likely to contain much more detailed artifacts in this manner than standalone client applications.

`Kumodocs` was developed to be easily generalizable to other collaborative services. Revision logs are transformed into an intermediate format, which are processed for artifact recovery. Any new services can be added by the addition of a simple log acquisition and parsing module. We also created a playback mechanism for *Docs* which uses the intermediate format, making this compatible with any logs obtained from other word processing apps, such as *Zoho Writer*. In addition, `kumodocs` can be used as a *privacy* auditing tool, allowing one to view all recoverable images, comments, and suggestions associated with their *Google Drive*.

By combining private and public API, `kumodocs` is able to obtain artifacts unavailable by other means. *Forms*, which has no native revision functionality, still maintains a revision log that we can recover, showing in far greater detail a form's evolution over time. Even though major revisions are available through the public API and browser interface, we are able to obtain a snapshot at **any** point in time. We are also able to access comment data from *Sites*, which is not supported in other means; deleted comments or replies contain the same metadata as the other services, as it uses the same underlying back end.

We found that *G Suite* collaboration exposes the complete editing history – a feature most users would not expect. Even partial commenting permissions allows every comment to be retrieved. Deleted comments and replies contain creation and deletion times as well as authorship information, with the ability to link others in conversation at specific points in time. Any images inserted become a part of the permanent history as well, being accessible to anyone with editing permission. The only reliable way to collaborate on a work in progress without exposing history is to clone the document; however, this erases the ability to revert to prior versions.

Appendix: Changelog keys

This appendix lists known keys found to appear in the changelog; Unknown keys will be listed in the following table.

Table 1: *Docs* changelog keys

Key	Interpretation
	operations
mlti	multi-operation (transaction)
is, ds	insert/delete string
ae, de, ue, te	embedded elements: adjust, delete, update, tether (to anchor)
rvrt	revert to earlier revision
op	(kix) operation: add/remove anchors, embed objects, etc.
sdef_ps, sdef_ts	set default paragraph/text style
as, sm	adjust style/style modifications
msfd	suggestion delete range (mark string for delete)
usfd	undo suggestion delete range (undo string for delete)
sas	suggestion adjust style
sugid	suggestion id
	operation attributes
mts	multi-operation description
s	string argument
si, ei	starting/ending index
ibi	insert before index
tbs_al, tbs_of	table alignment/offset
das_a	doc adj style anchor
	document attributes
ds_pw, ds_ph	page width/height
ds_mt, ds_mb	top/bottom margin
ds_ml, ds_mr	left/right margin
lgs_l	language

Table 2: *Docs* style keys

Key	Interpretation
	header styles
hs_t, hs_st	title, subtitle
hs_nt	normal text
hs_h1	h1
⋮	⋮
hs_h6	h6
	paragraph style
ps_hdid, ps_hd	heading id/style
ps_al, ps_ls	horizontal alignment line space
ps_il, ps_ifl	indent line/first line (amount)
ps_sb, ps_sa	space before/after paragraph (amount)
	text style
ts_ff, ts_fs	font family/size
ts_fgc, ts_bgc	foreground/background color
ts_bd, ts_it	bold/italic
ts_un, ts_st	underline/strikethrough
ts_sc, ts_va	small caps, vertical alignment

Table 3: *Slides* changelog codes

Code	Interpretation
	operations
0	delete box
3	add box
4	transaction
5	modify box
6	adjust page element
9	adjust page style
12	add slide
13	delete slide
14	move slide
15	add text
16	delete text
17	adjust text style
18	set slide attributes
22	insert table
43	transition
44	insert image
	style modifications
$[0, 1]$	bold flag
$[1, 1]$	italics flag
$[2, 1]$	underline
$[4, hexvalue]$	color
$[5, ffamily]$	font family
$[6, fontsize]$	font size (6..400)
$[7, fontmod]$	super/subscript font (1=super, 2=sub)
$[11, spacing]$	line spacing (100/115/150/200)
$[12, halign]$	horizontal alignment (1=left (default), 2=center, 3=right, 4=justified)
$[20, 1]$	strikethrough flag
$[44, valign]$	vertical alignment (0=top, 1=middle, 2=bottom)

Bibliography

[1] J. Chen and B. Mulligan. Quill rich text editor. [Online]. Available: https://github.com/quilljs/quill/

[2] H. Chung, J. Park, S. Lee, and C. Kang, "Digital forensic investigation of cloud storage services," *Digital Investigation*, vol. 9, pp. 81–95, 2012. [Online]. Available: http://dx.doi.org/10.1016/j.diin.2012.05.015

[3] Google. The next generation of Google Docs. [Online]. Available: http://googleblog.blogspot.com/2010/04/next-generation-of-google-docs.html

[4] ——. What's different about the new google docs: Making collaboration fast. [Online]. Available: https://drive.googleblog.com/2010/09/whats-different-about-new-google-docs.html

[5] ——. Whats different about the new google docs? [Online]. Available: https://drive.googleblog.com/2010/05/whats-different-about-new-google-docs.html

[6] S. R. Group, "2016 review shows $148 billion cloud market growing at 25% annually." [Online]. Available: https://www.srgresearch.com/articles/2016-review-shows-148-billion-cloud-market-growing-25-annually

[7] J. Hale, "Amazon cloud drive forensic analysis," *Journal of Digital Investigation*, vol. 10, pp. 259–265, October 2013. [Online]. Available: http://dx.doi.org/10.1016/j.diin.2013.04.006

[8] B. Martini and K.-K. R. Choo, "Cloud storage forensics: ownCloud as a case study," *Digital Investigation*, vol. 10, pp. 287–299, 2013. [Online]. Available: http://dx.doi.org/10.1016/j.diin.2013.08.005

[9] “Nist cloud computing forensic science challenges,” June 2014.

[10] D. Quick and K.-K. R. Choo, “Dropbox analysis: Data remnants on user machines,” *Journal of Digital Investigation*, vol. 10, pp. 3–18, June 2013. [Online]. Available: http://dx.doi.org/10.1016/j.diin.2013.02.003

[11] ——, “Google drive: Forensic analysis of data remnants,” *Journal of Network and Computer Applications*, vol. 40, pp. 179–193, April 2014. [Online]. Available: http://dx.doi.org/10.1016/j.jnca.2013.09.016

[12] RightScale. Cloud computing trends: 2016 state of the cloud survey. [Online]. Available: http://www.rightscale.com/blog/cloud-industry-insights/cloud-computing-trends-2016-state-cloud-survey

[13] V. Roussev, A. Barreto, and I. Ahmed, “Forensic acquisition of cloud drives,” in *Advances in Digital Forensics XI*, G. Peterson and S. Shenoi, Eds. Springer, 2016.

[14] V. Roussev and S. McCulley, “Forensic analysis of cloud-native artifacts,” in *Proceedings of the Third Annual DFRWS Europe (DFRWS-EU)*, 2016, pp. S104–S113, doi: 10.1016/j.diin.2016.01.013.

[15] J. Somers. How I reverse engineered google docs to play back any documents keystrokes. [Online]. Available: http://features.jsomers.net/how-i-reverse-engineered-google-docs/

Vita

Shane McCulley is a graduate research assistant at the University of New Orleans pursuing a Ph.D in Computer Science. His research interests include cyber security, digital forensics, privacy, CSCW, and artificial intelligence. He is currently developing forensic tools to extract and process cloud-native artifacts.