Secure Quasi-Realtime Collaborative Editing over Low-Cost Storage Services

Chunwang Zhang[1], Junjie Jin[1], Ee-Chien Chang[1] and Sharad Mehrotra[2]

[1] School of Computing, National University of Singapore, Singapore
{chunwang, jin89, changec}@comp.nus.edu.sg
[2] Department of Computer Science, University of California, Irvine, CA, USA
sharad@ics.uci.edu

Abstract. A realtime collaborative editor facilitates concurrent editing of a document by multiple authors. It is desired that the document be shared only among the authors, and protected from the potentially curious server. Existing approaches have taken two distinct paths – centralized server based approaches that achieve high concurrency and meet real-time requirement but compromise on security and incur high server cost, and peer-to-peer based approaches that support security but compromise on users' convenience and mobility. In this paper, we observe that by relaxing the realtime requirement, we can achieve security, reduce server cost and yet exploit the conveniences of the centralized setting. In particular, we consider generic low-cost storage servers in the cloud that provide storage integrity but do not guarantee low-latency. Essentially, our method breaks the document into small encrypted regions that are stored on the server and coordinates the authors' access. Although two authors are unable to concurrently modify a same region, the system is able to provide "quasi-realtime" experience. By relaxing the requirement to quasi-realtime, the difficulties in achieving document consistency, and the requirement on resources are significantly reduced. We give a proof-of-concept implementation on top of Dropbox, a commercial cloud storage service. Preliminary user studies show that the system is effective.

1 Introduction

A realtime collaborative editing system facilitates concurrent editing of a document shared by a few authors. While such systems improve productivity, it is desired that the document confidentiality be preserved. In particular, if the systems are hosted in the cloud, we want to protect the documents from the potentially curious servers.

There are many realtime collaborative editing systems with centralized public servers such as Google Docs [8] and the now discontinued Google Wave [9]. The centralized setting has a few advantages. With a centralized server, synchronization among the authors and techniques of *operational transformation* [6] can be efficiently and easily carried out, addressing the main technical challenge on concurrency faced by collaborative editing systems. Moreover, reliable

storage can be hosted by the servers. With such reliable storage, authors can readily edit the document from different devices and at different time, and thus facilitate mobility.

A main drawback of the centralized setting is the difficulty in achieving document confidentiality against the potentially curious servers. There are incentives, for example, business competitions, for the centralized servers to actively look into users' sensitive data [15]. In addition, the centralization of data also makes the servers high-value targets for attacks [1, 26]. To guide against the potential security risks, documents and operations must be carefully protected using, for example, proper encryption techniques, before being stored on the servers. However, for techniques of operational transformation, the servers need to know the contents of the operations in order to transform them. Without knowing the actual operations which contain certain information about the documents, it is very difficult for the servers to carry out the transformations.

On the other hand, a peer-to-peer (P2P) collaborative editing system naturally alleviates the above-mentioned issues since no server is involved. For the security requirement, as long as secure channels can be established among the authors, document confidentiality can be achieved. There are also many P2P collaborative editing systems such as SubEthaEdit [28] and CoWord [21]. Although no server is required, resolution of conflicts are now "pushed" to the peers and thus increases the computation load on the peers. Moreover, since there is no reliable storage keeping the latest version of the document, mobility is cumbersome to achieve. For example, consider the situation where Alice and other authors are concurrently editing a document using their desktops under a P2P setting, and later Alice goes offline for a few minutes and then resumes editing on her mobile device. Although possible, it is cumbersome for Alice's mobile device to retrieve the latest version of the document, locate and establish connection with other authors and then carry out collaborative editing.

A centralized system can be adopted in the P2P setting in the following way: after secure communication channels have been set up among the peers, one of the peer takes the role of the server and thus realtime collaborative editing can be carried out. In such adoption, the "super-peer" has to be present throughout the session. This leads to the difficulty in getting seamless recovery when the super-peer fails. It is also possible to have a hybrid setting where the peers employ a storage server to keep the latest version, while collaborative editing is done in the P2P setting. Nevertheless, this requires frequent uploading of the whole file and thus consumes large bandwidth. A recently proposed system SPORC [7] can be treated as a P2P system but with their communications going via a centralized server. The centralized server plays the role of establishing an ordering of the issued operations, which can be easily carried out on the encrypted operations. Such consistent ordering helps to simplify the designs of concurrency control. Nevertheless, it also inherits the requirement of low latency on the server of the centralized system, and high computing cost on the clients of the P2P system.

A main driving force of cloud computing is cost-saving. We know that there are already many commercial file hosting and sharing services in the cloud such as

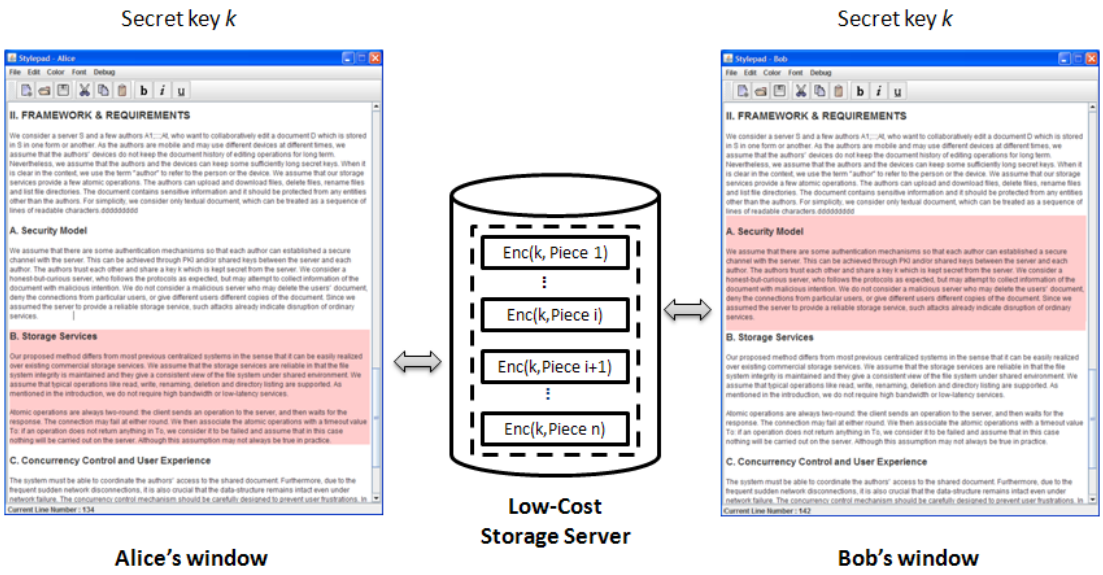


Fig. 1. An illustration of the scenario where there are two authors collaboratively editing a document using our system.

Dropbox[12], SugarSync [13] and Box [3]. With such services, users can reliably store their files there and easily share the files with anyone they like across the Internet. These storage services are low-cost in the sense that they do not guarantee low-latency, provide only low bandwidth, and/or allow limited number of access. Nevertheless, they are reliable in providing storage-integrity. Instead of having a fully trusted dedicated server, we want to leverage such existing storage services to provide a low-cost collaborative editing system where the document confidentiality is preserved. Hence, preferably, no additional new services are required on the servers to facilitate collaborative editing, and no processes are required to keep track the state of an editing session.

We take security, mobility, cost and user experience as the main design criteria. While it is difficult to achieve all of them simultaneously, we observe that by relaxing the realtime requirement to "quasi-realtime", we can have both the conveniences provided by the centralized setting and yet achieve document confidentiality over a generic cloud storage service. While a "true" realtime collaborative editor allows multiple users to edit a same sentence, or even a same word concurrently, in contrast, we do not allow multiple editing in a same *viewport*, but support concurrent editing in different viewports. Such design choice is based on the assumption that users seldom concurrently modify a small region of interest so as to avoid confusion, but yet want to concurrently edit the same document.

Essentially, our proposed method automatically breaks the document into pieces which are encrypted and stored in the server, and ensures that only one author can modify a piece at any given time. Figure 1 illustrates the scenario where there are two authors collaboratively editing a document using our system. Although the proposed method is conceptually simple, there are a few technical issues. Our servers only play the role of providing "shared memory", and do not actively participate in synchronizing the author's operations. Hence, there are still tricky issues in handling concurrency, especially when taking into consideration the frequent connection failures between the authors and the server. Moreover, localized editing to the document may affect global information, which has to be efficiently propagated to other authors .

We implemented a proof-of-concept system over Dropbox. Preliminary user study shows that, in the two-author collaboration scenario, our system can facilitate collaborative editing by saving around 30% of the time of the turn-taking approach, and the communication cost is very low.

The rest of the paper is organized as follows. We state the assumptions and requirements in Sect. 2. In Sect. 3, we describe the proposed method in detail, followed by a brief security and performance analysis in Sect. 4. We implemented a prototype system and conducted a small-scale user study to evaluate its effectiveness in Sect. 5. Related work and conclusion can be found in Sect. 6 and 7, respectively.

2 Framework & Requirements

We consider a server S and a group of authors $A_1, \ldots, A_t$, who want to collaboratively edit a document D which is stored in S in one form or another. The authors are mobile and may use different devices at different time, so the devices are stateless in the sense that they do not keep the record of editing operations after the end of the editing session. Nevertheless, the authors and/or the devices can keep some sufficiently long secret keys. For simplicity, we consider only textual document, which can be treated as a sequence of lines of readable characters and each character is associated with different attributes. An example of such documents is shown in Fig. 1.

2.1 Security Model

We assume that there are some authentication mechanisms so that each author can establish a secure channel (e.g., SSL/TLS) with the server. The authors trust each other and share a key k which is kept secret from the server. We consider a *honest-but-curious* server that follows the protocols as expected, but may attempt to collect information from the document with malicious intentions. The server must not be able to learn the document's content. Since the server is honest, it will not delete the document, deny connections from particular authors, or give authors inconsistent copies of the document. Nevertheless, we believe that it is possible to extend our proposed method to achieve authenticity

against a malicious server without incurring large amount of resources. This extension would be an interesting future work.

2.2 Storage Services

Our system relies on a reliable existing storage service to host the shared document. The service is reliable in the sense that the file system integrity is maintained and it provides a consistent view of the file system under the shared environment. As mentioned in the introduction, we do not require high bandwidth or low-latency guarantees. The servers support the following *atomic operations* over the network.

1. $getFile(name)$: Retrieve the file "*name*" from the server.
2. $deleteFile(name)$: Delete the file "*name*" on the server.
3. $putFile(content,\ name)$: Upload the file "*name*" with the given content. If there is already a file "*name*" on the server, overwrite it.
4. $rename(oldname,\ newname)$: Rename the file "*oldname*" to "*newname*". If two clients happen to rename a same file concurrently, only one of them can succeed.
5. $listDir()$: List the metadata for all files in the directory. The metadata for each file must include at least its filename and last modified time.

Each atomic operation is to be completed over the network in one round-trip: the request, followed by the reply. Note that communication may fail during the operations. Actual storage services may provide different operations. The above set of atomic operations is minimal required for our system.

2.3 Mobility and Cost

Consider the scenario described in the introduction. When Alice switches the editing session from one device to another device, such transition should be easy and convenient for her. In particular, the new device should be easy to obtain the latest version of the document and establish communication with the relevant entities. Note that with the centralized storage server, such requirements on ease-of-use can be easily achieved. Since we consider light-weight mobile devices such as PDAs and smartphones, computations on the client side must not be overly intensive. Furthermore, communications between the clients and the server should not consume large bandwidth.

2.4 Concurrency Control and User Experience

Collaborative editing systems need concurrency control mechanisms to keep all the clients' local copies consistent. While with a centralized server, consistency is not difficult to achieve, it is important to ensure that the control mechanisms do not weaken usability of the system. In particular, below are some situations that the concurrency control mechanism should minimize:

1. The number of *roll-backs*. A roll-back occurs when a few operations issued by an author are deemed to conflict and have to be discarded, and thus already modified document has to be rolled back. The length of a roll-back should also be minimized.
2. The number of user interventions. Certain concurrency control mechanisms (for e.g., some optimistic locking mechanisms) require users' feedback in resolving conflicts. This should be avoided. Furthermore, users' effort in concurrency control (for e.g., button clicks) should also be minimized.
3. Waiting time to modify a region. Clearly, the time to withhold an author from editing the document should be minimized.

3 Proposed method

A common "manual" practice to concurrently edit a document is to have the authors divide the document into large pieces according to the structure of the document, where each piece can only be edited by a single author at any given time. The authors coordinate with each other by communicating through some real-time channels like using phone calls or even via email systems, to lock and unlock a piece. Our system essentially automates this process with the coordination and management of the pieces transparent to the authors.

Essentially, our system automatically breaks the large document into small pieces which are encrypted and stored on the server, and ensures that a piece can only be modified by one author at any given time so as to avoid conflicts. Modified pieces are periodically re-encrypted on the client side and pushed back to the server, overwriting the original ones there. Due to the automation, frequent switches among the pieces can be carried out in a seamless manner, and thus the authors can enjoy "quasi-realtime" experiences by concurrently working on different pieces.

3.1 System Overview

The proposed system has the following components: it (1) employs a simple pessimistic locking mechanism (with timeout) to achieve concurrency; (2) manages the pieces in a smart way by automatically dividing (merging) a piece if it is getting too large (small) as the result of edit, and keeps the document intact even if failures happen during the dividing/merging process; (3) maintains global information in an efficient way, and (4) provides a user-friendly editing experience. As mentioned before, the server is not actively involved in the collaborative session, in the sense that there is no server-side process that is dedicated to keep track of the editing operations and push information back to the authors. The application logic, therefore, must be enforced on the client side.

3.2 Internal Representation

The system breaks the document into a sequence of *sub-files* $F_1, F_2, \ldots, F_n$ which are to be encrypted and stored as individual files in the same directory on the

server. The name of each subfile is also encrypted using format preserving encryptions to produce valid filenames. The filename (in plaintext form) is a 4-tuple (p, i, g, s), where

1. p: A string identifying the document.
2. i: The index indicating the position of the subfile in the sequence. Here, i is a decimal real number (for e.g. "12.15"). A subfile with index i appears before the subfile with index j iff $i < j$.
3. g: The global information. This indicates changes made in the subfile that could affect some global states, like total number of lines. Note that although such information can be derived from the subfile, having it in the filename could potentially reduce communication cost.
4. s: Status of the subfile, that is, whether the subfile is *locked* or *unlocked*. If it is locked, we also include the identity of the author who has locked it.

We use total number of lines as an example to illustrate how global information can be efficiently maintained. Note that (p, i, g, s) is the minimal required information for our system. For example, the subfile ("*part*", 2.0, 100, "*unlocked*") means that it contains 100 lines of content and it is currently not being locked by anyone, while the subfile ("*part*", 3.5, 150, "(*Alice*)*locked*") contains 150 lines of content, and it is currently being locked by Alice. More information might have to be included when extending the system to richer text format.

3.3 Concurrency Control

A successful *listDir* operation will give the information (p, i, g, s) and the last modified time for each subfile in the directory. With such information, together with the atomic operations defined in Sect. 2.2, we can have a simple pessimistic locking mechanism. In this mechanism, the atomic operation *rename* plays the role of locking and unlocking a subfile. In particular, locking (unlocking) a subfile is carried out by changing its name to the *locked* (*unlocked*) status through a *rename* request. Authors can continue to read a locked subfile while it is being updated, but they are not allowed to modify it. They will be continuously notified about the updates.

However, due to frequent network failures, a subfile may be locked forever, if for instance, the author holding the lock is unable to unlock due to such failures. We need a lock timeout to release those subfiles. Let the length of the timeout be T_u. The clock for timeout starts immediately when a subfile is locked and restarts whenever it is updated. If a subfile with its name in the locked status has not been modified for a time period T_u, it is no longer considered as being locked and thus any one can re-lock it. As the authors' time are not synchronized, we further extend the timeout to $T_u + \sigma$ where σ is a bound on the time differences of the authors.

Note that in this locking mechanism, we do not have to explicitly unlock a subfile as anyone can re-lock it after a certain amount of time (i.e., $T_u + \sigma$). However, for better user experience, if it is clear that a user is not working on a piece, the piece will be unlocked immediately.

3.4 Piece Management and Failure Recovery

As a subfile is being edited, it may become extremely large or small. Note that the size of subfiles affects the performance dramatically. While a small piece size will facilitate collaborative editing as the authors can now work on finer pieces, it will generate large network overhead. Our system employs a simple piece management policy: whenever a subfile is getting too large (small), the system will automatically divide it into smaller pieces (merge it into other pieces). In our current implementation, the piece size is controlled by a range $[S_{min}, S_{max}]$ that is setable by the authors.

Dividing/merging a subfile involves a sequence of $deleteFile$ and $putFile$ operations. Network may fail during this process, leaving certain operations not being carried out. For example, consider the case where Alice wants to divide a large subfile by first deleting the original one on the server and then uploading the divided subfiles. Suppose that the network gets disconnected just after she has carried out the deleting operation. Since the new divided subfiles are not uploaded yet, some contents of the document are thus lost. Note that other authors cannot distinguish whether such contents are disappeared due to Alice's deletion or due to network failures, as there is no server side process that is monitoring the connections of the authors. We say that the document is not intact if some of its content are lost or replicated.

Algorithm 1 MERGE(F_1, F_2): Merge two subfiles F_1 and F_2

Input: Subfiles F_1 and F_2, with name (p, n_1, g_1, s_1) and (p, n_2, g_2, s_2) respectively.

1: Create a new subfile F' with name (p, n', g', s') where $n' = n_2$, $g' = g_1 + g_2$ and $s' = s_2$;
2: Copy the contents of F_1 and F_2 to F';
3: $rename((p, n_2, g_2, s_2), (p, n_2, g_2, \text{“}(n')D\text{”}))$;
4: $rename((p, n_1, g_1, s_1), (p, n_1, g_1, \text{“}(n')D\text{”}))$;
5: $putFile(F', (p, n', g', s'))$;
6: $delete((p, n_1, g_1, \text{“}(n')D\text{”}))$;
7: $delete((p, n_2, g_2, \text{“}(n')D\text{”}))$;

Algorithm 1 describes a merging process that ensures intactness. The dividing process is similar and thus it is omitted. The key idea here is to first rename the subfile to be divided/merged in a proper way rather than deleting it directly from the server. Subfiles ending with "$(n')D$" are called the temporary *indicating* subfiles which provide necessary information for failure recovery in case any error happens during the dividing/merging process. Whenever an indicating subfile is found stayed on the server for a long time (e.g., $T_u + \sigma$), the subfile will be either deleted or renamed to the unlocked status depending on whether the target subfile (i.e., the subfile with index n') is already on the server or not, keeping the document always in an intact state.

3.5 Management of Global Information

Localized edit to a piece may affect global information. In our editor, there is a status bar that displays the position of the current cursor w.r.t the start of the document (see Fig. 1, at the bottom of the window). This cursor position, in term of "line number", is global information: if an author deletes or inserts one line in her viewport, the coordinate of the cursor in other authors' viewports should be updated accordingly. For example, suppose that Alice is working on the 5th line of the document while Bob is working on the 1000th line. If Alice deletes one line in her viewport, the coordinate of Bob's cursor should be updated to 999 accordingly. Although such information can be always derived from the subfile by downloading the latest version from the server, doing that, however, will consume the limited network bandwidth.

We address this efficiency issue by adding the required information, that is, the total number of lines of a subfile, into the subfile's name. When a subfile is updated, such information must be updated accordingly. Other authors, by reading only the filename (through the periodical *listDir* request), can know easily how the subfile has been changed. This gives an efficient way to maintain the global information. More data might have to be included when extending the system to support more global information.

However, the filename length restrictions must be taken into consideration. For file systems that limit the length of the filename (e.g., Windows has some odd behaviors with length over 260 characters, including path), information that can be included in the filename could be limited. In such a case, we can have each subfile associate with a small metadata file recording all the necessary global information that we want to maintain, and ensure that each update to the subfile must update the metadata file as well. We would like to explore this possibility in the future.

3.6 User Interface

It would be troublesome to require the authors to manually lock and unlock a piece, especially when the pieces are small and switches between the pieces are frequent. Indeed, in our system, there is no GUI control, like clickable buttons, for them to do so. The authors can start to edit any regions of interest immediately, while the system takes care of the underlying layer locking and unlocking operations. To edit a particular location in the document, the author moves the cursor to that location and then starts editing. Let us denote T_e as the time where the author's editing action (implicitly) generates an locking request, and T_r as the time where the server's response to the locking request reaches the author. In the event where the locking is unsuccessful, whatever editing operations issued by the author during the period T_e to T_r have to be discarded, and we call such event a roll-back. Clearly, from users' point of view, the number of roll-backs, and their period have to be minimized. In our system, fairly intuitive color information is provided for the authors to distinguish between regions with different locking status. By trying not to modify a region that is shown being

used by others, they can significantly reduce the chance of roll-backs (although not totally avoid them). Furthermore, even when a roll-back happens, operations involved between T_e and T_r are usually few, because the *rename* operation typically completes in just a few seconds. In our informal test, most of the *rename* operations complete within 3 seconds. Thus, we totally relieve the authors from the tedious tasks of locking and unlocking a subfile by only paying a price for seldom and short-time roll-backs.

For cases where the authors' devices do not have sufficient resources, for instance, when they are using light-weight mobile devices, they can work on a small part of the total document. When a client is just started, it downloads only the first few (for e.g., 3) subfiles from the server. Whenever an author requires for new contents during her editing, the next subfile will be retrieved and the current first subfile will be removed from the editor. In such a way, large documents are supported even with resource-limited devices.

4 Analysis

With a centralized server, the system design is much simplified. Although the server does not actively help in synchronizing the authors and pushing updates back to them, having it essentially facilitates mobility, and makes it possible to support light-weight mobile devices. Furthermore, as the server is generic, the system can be implemented on a variety of existing storage services most of which provide a certain amount of free storage space. Thus, the cost of having a server is lower as compared to other centralized systems which require dedicated servers.

4.1 Security Analysis

Like SPORC [7], our system bases its security and privacy guarantees on the presence of secure encryption schemes and the security of the authors' cryptographic keys, and not on a trusted or invulnerable server. Since the curious server sees only the encrypted version of the document, it cannot collect any information about the document's content. Furthermore, individual editing operations generated by each author stay only in the client itself, and are invisible to the server. So it is also impossible for the server to perform operation analysis. The document confidentiality is hence achieved.

However, there are still some information that the server can deduce. One particular example is the size of the shared document, since the server is already storing the (albeit encrypted) document. The server also knows how many pieces the document has been cut into. More importantly, the server knows the identities of the authors involved in the collaborative editing session as they must login first to the server, and their action sequences (i.e., who has modified which piece at what time). By comparing the differences in size between two successive updates to a same subfile, the server is able to guess overall what kinds

of operations (e.g., insertion, deletion) have been performed, although the precise operation contents and editing positions are not clear. Although, intuitively, such information may not be useful to an adversary, users of this system should be aware of such possible leakages.

Unlike SPORC, we do not consider a malicious server that may intentionally modify, delete or re-order particular subfiles. We argue that there is little incentive for commercial storage services to perform such malicious actions. Nevertheless, as mentioned before, we believe that it is possible to extend our system to achieve authenticity and integrity against a malicious server without incurring high overhead to the storage services. We leave this for future work.

4.2 Performance-Critical Parameters

The performance of our system can be affected by many parameters, as summarized in Table 1. By performance, we are referring to user editing experience and network overhead. One interesting work is to investigate, comprehensively, how these parameters will affect the system, and to determine the optimal values through, for example, system modeling or large-scale user studies. That could be a separate paper. In this paper, we just discuss the tradeoff between them.

Table 1. Summary of the performance-critical parameters.

S_{min}	Minimum size of a sub-file;
S_{max}	Maximum size of a sub-file;
T_u	Lock timeout;
T_o	Connection timeout;
T_s	Auto update saving period;
T_k	Auto update checking period.

1. S_{min} and S_{max}: The minimum and maximum size of a subfile. Subfiles out of this range will be divided or merged. Thus, in a sufficiently long-term usage, the average size is $(S_{min} + S_{max})/2$ (i.e., assumed a uniform distribution). A small average size facilitates collaborative editing as authors can now work on finer pieces and switches between them are less possible to trigger conflicts, but leads to higher network overhead as more information has to be returned by the periodical *listDir* request and more frequent locking (and unlocking) requests have to be sent. The distance between S_{min} and S_{max} should be reasonably large; otherwise, file dividing and merging may happen too often which wastes a lot of bandwidth. On the other hand, S_{min} (S_{max}) cannot be too small (large) in order to avoid the cases of extremely small (large) subfiles.
2. T_u: The lock timeout, that is, after how much time the lock for a subfile is no longer considered as valid. A larger T_u increases the waiting time for a

subfile to be released, while a smaller T_u increases the number of locking requests. In particular, T_u should be sufficiently larger than the connection timeout T_o in order to tolerate the network delay.

3. T_o: The connection timeout. The larger the average network delay is, the larger the T_o should be. In particular, if T_o is not large enough, an atomic operation may be indeed carried out on the server while the author is unaware of it (and considers the operation as failed). Although this will not cause any problem in concurrency, it frustrates the authors and increases the network overhead (e.g., the author may issue a same operation quickly).
4. T_s: The update saving period. Every T_s seconds, the modified subfile is automatically encrypted and uploaded to the server, replacing the original one there. A smaller T_s can push changes more quickly and reduce the loss when a failure happens. However, it will increase the network overhead and the number of requests. In particular, T_s should be smaller than $T_u - T_o$ so that an active author can work on a subfile steadily without being disrupted.
5. T_k: The update checking period. Every T_k seconds, each client will check the server for new updates through a $listDir$ request. Like T_s, a smaller T_k will improve the "realtime" experience but increase the network cost. Note that T_s and T_k together determine the modification propagation time T, that is, the time between a change is made and the change reaches other authors' devices. In particular, $0 < T < T_s + T_k + T_o$.

In real applications, these parameters must be determined according to, for instance, the real network conditions and possible constraints from the storage services. As an example, if the storage services do not limit the bandwidth usage and/or the number of accesses, we can have a small piece size and small T_s and T_k which in turn improves the authors' editing experience. The best way is to conduct formal experiments to decide the optimal values.

5 Implementation & User Study

We implemented a proof-of-concept system over Dropbox using its API v1.2 (in Java). The API provides all the atomic operations we defined in Sect. 2.2, and functionalities for clients to do authentications. The document is broke into many small files according to its structure. Both the name and content of each file are encrypted using AES-256 (with CBC mode) and the shared secret key. Authors login to the Dropbox server using valid username and password pairs they have, locate the directory and then open the document for editing. The corresponding files are then downloaded to the clients, decrypted and displayed as a single document to them. We provide a simple interface for the authors to edit the document, as shown in Fig. 1. Basic editing operations such as insertion, deletion, copy, cut and paste are provided. Authors can also change the style of any particular characters. Note that there are no control buttons for them to lock and unlock any files. Regions with red background indicate that they are currently being updated by others.

We want to study the effectiveness of the system, that is, whether users can save a significant amount of time by using our system in collaborative editing, and the communication cost. We conducted a few small-scale user studies to achieve such goals. The performance-critical parameters are set as follows: S_{min}=10 and S_{max}=100 (lines), T_o = 30, T_u = 120 and T_s=T_k=10 (seconds).

5.1 Scenarios and Participants

We simulated a scenario where there were two authors collaboratively editing a document. In this scenario, each author had to complete a list of editing tasks that were provided by us. We used a thesis paper (80 pages) as the base document, and created 4 different task lists from it, denoted as T_1, T_2, T_3 and T_4. Each task list contains 25 editing tasks, for example, "*Insert the sentence "Centralized systems are easier to build as compared to distributed systems." at position {1}*", "*Delete the word "collaborative editor" at the first paragraph in section 2.1*", and "*Set the font size of the heading "Chapter 1 Introduction" to 18 and make it bold*". The task lists are carefully designed in a way such that the two authors have to edit some same regions so as to trigger conflicts (and thus roll-backs). Each task list takes around 15 minutes to complete. Due to the space limit, details of the task lists are omitted here. 8 participants took part in the study, denoted as $A_1, A_2, \ldots, A_8$. 4 of them are male, and 4 are female. All of them are undergraduate students from the CS department of the first author's university. All use computers on a regular basis, and have experience in editing various kinds of documents. We randomly grouped them into 4 groups.

5.2 Effectiveness

We compared the effectiveness of our system with the basic *turn-taking* approach: after one author finishes, she passes the document to the other author for editing. While a more natural approach that allows the participants to freely communicate with each other and edit the document in any ways they like could be more interesting, the turn-taking approach is much easier to conduct and control in practice.

The experimental procedure is summarized in Table 2. Each group was required to complete two rounds of tasks, and in each round, they had to complete a given task list in the two-author collaboration scenario using either our system or the turn-taking approach. Group 1 and 3 started with our system in the first round while Group 2 and 4 used it in the second round so as to avoid the possible bias on any particular systems due to the training effect. Note that for each participant, a different task list was used in the second round. Participants in each group sat back-to-back so that they could not see directly each other's screen. We provided WordPad as the editor (i.e., when not using our system) because WordPad is common in Windows and it provides basic editing functionalities which are very close to our editor. We recorded the time that the participants spent in each round.

Table 2. Summary of the experimental procedure.

Participants		1st round		2nd round	
		Task	Editing system	Task	Editing system
Group 1	A1	T1	Our system	T3	Turn-taking
	A2	T2		T4	
Group 2	A3	T1	Turn-taking	T3	Our system
	A4	T2		T4	
Group 3	A5	T3	Our system	T1	Turn-taking
	A6	T4		T2	
Group 4	A7	T3	Turn-taking	T1	Our system
	A8	T4		T2	

The experiment result is summarized in Table 3. We define total editing time as the sum of the time spent by the two participants for turn-taking approach, and the time spent by the participant who finished late for our system. The result of total editing time is also shown in Table 3 (i.e., the 6th and 10th columns) and illustrated in Fig. 2. It is clear that our system can facilitate collaborative editing. Consider Group 1 and 3 where our system was used in the first round. The time spent is only 77.52% (i.e., (22:05+23:06)/(28:10+30:07)) of that of the turn-taking approach. Consider the results of Group 2 and 4, the time reduction is even more: participants in our system used only 61.47% (i.e., (18:54+19:04)/(30:53+30:53)) of the time of the turn-taking approach. Thus, to summarize, our system is effective in the sense that, in the two-author collaboration scenario, our system can clearly improve productivity by saving around 30% (i.e., 1–(77.52%+61.47%)/2) of the time, as compared to the case where they take turns to edit the document.

There are some other observations. First, with our system, each participant spent more time in completing the required tasks: the average time is 20 minutes, while that is 15 minutes in the turn-taking approach. This is partially because of the roll-backs where some already made changes have to be discarded due to

Table 3. Summary of the experiment results.

Participants		The 1st round				The 2nd round			
		Task	Editing system	Time (m:s)	Total time (m:s)	Task	Editing system	Time (m:s)	Total time (m:s)
Group 1	A1	T1	Our system	21:17	22:05	T3	Turn-taking	14:01	28:10
	A2	T2		22:05		T4		14:09	
Group 2	A3	T1	Turn-taking	16:04	30:53	T3	Our system	18:54	18:54
	A4	T2		14:49		T4		15:34	
Group 3	A5	T3	Our system	20:42	23:06	T1	Turn-taking	14:45	30:07
	A6	T4		23:06		T2		15:22	
Group 4	A7	T3	Turn-taking	14:45	30:53	T1	Our system	16:14	19:04
	A8	T4		16:08		T2		19:04	

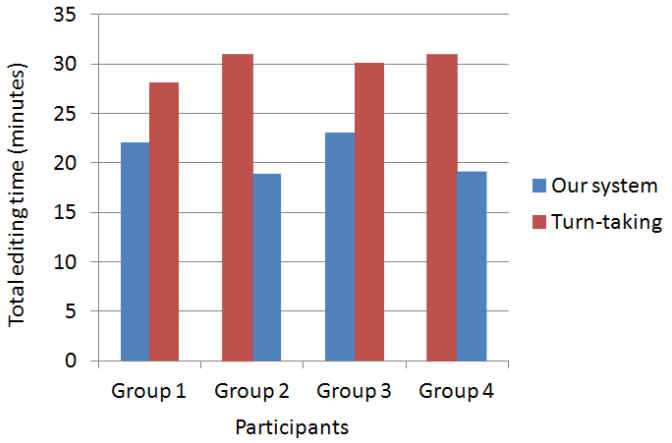


Fig. 2. Total editing time.

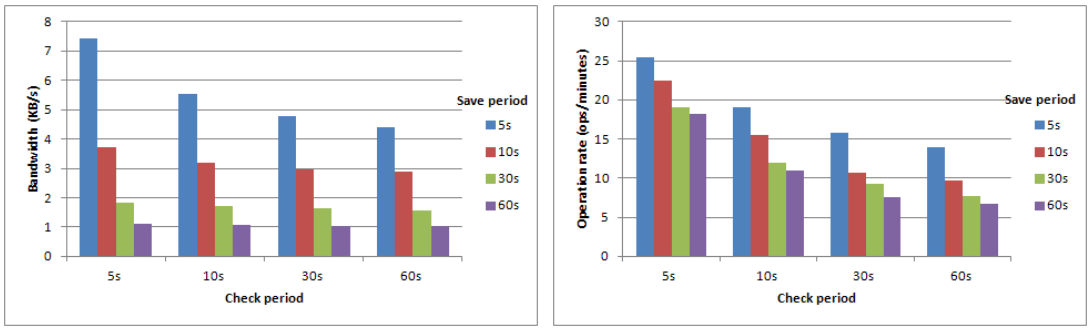
conflicts and the participants have to move back sometime later to continue the failed tasks. Note that our task lists were intentionally designed in such a way so as to trigger conflicts. Another reason is the familiarity with editing tools and the "search" function provided by WordPad. Most of the participants have used WordPad before, and we indeed noticed some of them use keyword search to help in finding the locations of particular tasks. A second observation is that, with a same editing system, participants typically spent less time in their second rounds which indicates that the training effect is still present.

Given that conflicts are rare in practice when the document is large, we expect our approach to continue to perform well when the number of collaborating users increases. In contrast, the turn-taking approach would sequentialize the process of editing preventing individuals from accessing the file concurrently.

5.3 Communication Cost

We measure the communication cost in terms of the bandwidth usage and the operation rate (i.e., number of operations per minute). While the communication cost could also be affected by parameters like average piece size and the lock timeout T_o, in this small experiment, we only consider it as a function of T_s and T_k (i.e., the update saving and checking period, respectively). The parameters are set as follows: S_{min}=10, S_{max}=100 (lines), $T_o = 30$ and $T_u = 120$ (seconds). We change T_s and T_k from 5 to 60 (i.e., 5, 10, 30 and 60, in seconds). For each combination, we asked the participants in each group to complete a given task list under the two-author collaboration scenario using our system, and recorded the amount of data and the number of requests transferred during this period. The task lists here are much shorter, and each takes around 5 minutes to complete.

The results are computed as an average of the 4 groups. Figure 3(a) shows the bandwidth usage as a function of T_s and T_k. As expected, the bandwidth



(a) Bandwidth usage as a function of T_s and T_k (b) Operation rate as a function of T_s and T_k

Fig. 3. Communication cost.

usage decreases when T_s and T_k increase, because data has to be less frequently exchanged between the clients and the server. The figure also shows that the update saving period T_s plays a more important role in the overall cost. For a fixed T_k, when the value of T_s doubles, the communication cost can be reduced by nearly a half. In contrast, the checking period T_k has only limited effect. For example, for a fixed T_s of 30s, the bandwidth usage only decreases from 1.81 to 1.56 (KB/s) when we increase T_k from 5s to 60s. Figure 3(a) also shows that the communication cost is very low. Even in the most loaded case ($T_s = T_k = 5$s), the overall bandwidth usage is only around 7.5 KB/s. While in the least loaded case ($T_s = T_k = 60$s), the bandwidth usage decreases to only 1.0 KB/s.

Figure 3(b) shows the operation rate (ops/minute) as a function of T_s and T_k. Like the bandwidth, the operation rate also decreases when T_s and T_k increase, but the downward trend is more gently. The overall operation rate changes from 25.5 to 6.75 (ops/minute) from the most loaded case to the least case. For storage services like Dropbox that limit the number of accesses per day, these parameters must be carefully set in order not to exceed to maximum values.

6 Related Work

Many real-time collaborative editing systems with centralized servers have been proposed over the past two decades [17, 20, 18, 8, 9]. Early systems like ShrEdit [17] employ simple concurrency control mechanisms such as locking and require the servers to manage the locks. The lock granularity could be a whole document, a section, a sentence or any selected texts. While we also employ a similar locking mechanism, servers in our system are not required to maintain any locking information. Many recent systems, for example, Google Wave [9], employ operational transformation (OT) [24, 6] for consistency maintenance and concurrency control in an intention-preserving manner [20, 9]. With a centralized server, OT can be carried out in the server and thus reduces the clients' workload. There are also systems that leverage transactional techniques in database to ensure con-

currency [29]. However, all these systems are designed based on the assumption of dedicated and trusted servers.

On the other hand, there are also many peer-to-peer collaborative editing systems [6, 14, 27, 28, 21] and research efforts on improving the concurrency control in the P2P setting [19, 10, 22, 25, 24]. Typically, P2P systems require the documents to be replicated on each involved device and are sometimes cumbersome for session switches when the authors are mobile. Although it is also possible to have a hybrid setting [2, 30], such systems still require active involvements of the servers.

A recent system SPORC [7] proposed a generic group collaboration framework using untrusted cloud servers where the document confidentiality and integrity are preserved. The centralized server plays the role of assigning a total ordering for the submitted operations, which can be easily carried out on encrypted operations. Although simple, there are still requirements of a dedicated server and resources on the server to keep track of the editing session. Moreover, since the workload of performing operational transformation are now pushed to the clients, the computation cost on clients is high and thus this approach is not suitable for mobile applications. In contrast, our system can be applied on any existing cloud storage services and we require no control of the servers. Our system is also carefully designed to support light-weight mobile devices.

Gabriele et al. [5] and Huang et al. [11] proposed methods to protect document privacy in Web-based (collaborative) editing applications such as Google Docs [8] and Microsoft Office Live [4]. The general idea is to encrypt the document contents before uploading them to the cloud server. Although these solutions can achieve certain degree of document privacy, applying them to the current Web-based document editing applications will partially disable the collaborative editing feature. In addition, two very recent systems, Venus [23] and Depot [16], allow clients to use cloud resources without having to trust them. Venus provides strong consistency in the face of a potentially malicious server, but requires the majority of a "core set" of clients to be online in order to achieve most of its consistency guarantees, and does not support applications other than key-value storage. Depot, on the other hand, does not rely on the availability of a "core set" and supports varied applications. Moreover, it allows clients to recover from malicious forks. However, unlike our system, Depot is not designed for real-time collaborative applications.

7 Conclusion

Many existing real-time collaborative editing systems employ operation transformations to achieve consistency. However, supporting such techniques with the additional goals on security, mobility and low services cost is difficult. In this paper, we argue that a relaxed quasi-realtime requirement could significantly simplify the system design. Preliminary user-studies on our proof-of-concept implementation showed that that such system is effective.

Acknowledgments. The authors would like to thank Veronica Hu He, National University of Singapore, for her help in implementing the system and conducting the user studies. Chang and Zhang are partially supported by grant TDSI/09-003/1A.

References

1. Debian investigation report after server compromises. `http://www.debian.org/News/2003/20031202`, December 2003.
2. R.M. Baecker, D. Nastos, I.R. Posner, and K.L. Mawby. The user-centered iterative design of collaborative writing software. In *Proceedings of the ACM CHI'93 Human Factors in Computing Systems*, pages 399–405. ACM, 1993.
3. Box. Box: Simple Online Collaboration. `http://www.box.com/`.
4. Microsoft Corporation. Microsoft Office Live. `http://www.officelive.com/`.
5. G. D'Angelo, F. Vitali, and S. Zacchiroli. Content cloaking: preserving privacy with google docs and other web applications. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 826–830. ACM, 2010.
6. C.A. Ellis and S.J. Gibbs. Concurrency control in groupware systems. *ACM SIGMOD Record*, 18(2):399–407, 1989.
7. A.J. Feldman, W.P. Zeller, M.J. Freedman, and E.W. Felten. Sporc: group collaboration using untrusted cloud resources. In *Proceedings of the 9th USENIX conference on Operating Systems Design and Implementation*, page 1, 2010.
8. Google. Google Docs. `https://docs.google.com`.
9. Google. Google Wave. `https://wave.google.com/wave`.
10. S. Greenberg and D. Marwood. Real time groupware as a distributed system: concurrency control and its effect on the interface. In *1994 ACM conference on Computer Supported Cooperative Work*, pages 207–217. ACM, 1994.
11. Y. Huang and D. Evans. Private editing using untrusted cloud services. In *The 31st International Conference on Distributed Computing Systems (ICDCS 2011)*, pages 263–272. IEEE, 2011.
12. Dropbox Inc. Dropbox: Simplify your life. `https://www.dropbox.com/`.
13. SugarSync Inc. SugarSync. `https://www.sugarsync.com/`.
14. M. Koch. The collaborative multi-user editor project iris. *Technical Report*, 1995.
15. J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (sundr). In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation-Volume 6*, pages 9–9. USENIX Association, 2004.
16. P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. Depot: Cloud storage with minimal trust. In *ACM Transactions on Computer Systems (TOCS)*, volume 29, page 12. ACM, 2011.
17. L.J. McGuffin and G.M. Olson. *ShrEdit: A Shared Electronic Work Space*. University of Michigan, Cognitive Science and Machine Intelligence Laboratory, 1992.
18. Raphael A Finkel Mullick, Sachin. MUSE: A Collaborative editor. `http://www.cs.engr.uky.edu/~raphael/studentWork/muse.html`, 1998. Masters Project. University of Kentucky.
19. C.M. Neuwirth, D.S. Kaufer, R. Chandhok, and J.H. Morris. Issues in the design of computer support for co-authoring and commenting. In *1990 ACM conference on Computer Supported Cooperative Work*, pages 183–195. ACM, 1990.

20. D.A. Nichols, P. Curtis, M. Dixon, and J. Lamping. High-latency, low-bandwidth windowing in the jupiter collaboration system. In *Proceedings of the 8th Annual Symposium on User Interface Software and Technology (UIST'95)*, pages 111–120. ACM, 1995.
21. Advanced Collaborative Technology Research. Codoxware: Connecting people and documents. `http://www.codoxware.com/`.
22. M. Ressel, D. Nitsche-Ruhland, and R. Gunzenhäuser. An integrating, transformation-oriented approach to concurrency control and undo in group editors. In *1996 ACM conference on Computer Supported Cooperative Work*, pages 288–297. ACM, 1996.
23. A. Shraer, C. Cachin, A. Cidon, I. Keidar, Y. Michalevsky, and D. Shaket. Venus: Verification for untrusted cloud storage. In *Proceedings of the 2010 ACM workshop on Cloud computing security workshop*, pages 19–30. ACM, 2010.
24. C. Sun, X. Jia, Y. Zhang, Y. Yang, and D. Chen. Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. In *ACM Transactions on Computer-Human Interaction (TOCHI)*, volume 5, pages 63–108. ACM, 1998.
25. C. Sun and P. Maheshwari. An efficient distributed single-phase protocol for total and causal ordering of group operations. In *3rd International Conference on High-Performance Computing (HiPC'96)*, page 295. IEEE Computer Society, 1996.
26. Owen Taylor. Intrusion on www.gnome.org. `http://mail.gnome.org/archives/gnome-announce-list/2004-March/msg00114.html`, 2004.
27. GH ter Hofte and HJ van der Lugt. Cocodoc: a framework for collaborative compound document editing based on opendoc and corba. In *Proceedings of the IFIP/IEEE international conference on Open distributed processing and distributed platforms*, pages 15–33. Chapman & Hall, Ltd., 1997.
28. TheCodingMonkeys. SubEthaEdit: Collaborative text editing. `http://www.codingmonkeys.de/subethaedit/`.
29. Q. Wu and C. Pu. Modeling and implementing collaborative editing systems with transactional techniques. In *Proceedings of the 6th International Conference on Collaborative Computing: Networking, Applications and Worksharing*, pages 1–10. IEEE, 2010.
30. A.A. Zafer. Netedit: A collaborative editor. Master's thesis, Master of Science, University de Virginia, USA, 2001.