Web and File Based Document Collaboration via Aysnchronous and Synchronous Updates

Daniel Yule
Dalhousie University
Halifax, Canada
yule@cs.dal.ca

James Blustein
Dalhousie University
Halifax, Canada
jamie@cs.dal.ca

ABSTRACT

Collaborative documents can be worked on synchronously (collaborators see updates as they happen) and asynchronously (collaborators see updates after they are shared). Synchronous updates are useful when collaborators are working together in real-time on the details of a document. Asynchronous updates are more appropriate for sharing larger sets of changes or for publishing a particular version. Furthermore, although web-based collaboration is ideal for real-time updates, often authors wish to use software installed on their local computer to edit documents. As such, this note will outline an approach for merging synchronous and asynchronous updates across both the WWW and the user's local file system using Operational Transforms and Conflict-Free Replicated Data Types.

1. INTRODUCTION

Remote file synchronization is the process of replicating a file system across multiple devices. Recently, services such as Googe Drive, Dropbox and Microsoft OneDrive have made the concept widely available, but the literature shows work as far back as the early 1980s [18].

The system we will demonstrate is both synchronous, so that close collaborators can see and respond to updates in real time, and asynchronous, so that the files can be propagated to external locations on the user's request. This system is useful for example in an academic publishing model where authors can write a paper while collaborating in realtime on a departmental server, then push it to a server provided by a journal. Reviewers can see the file, add comments, and after the review period is over, the commented version can be sent back to the authors for revisions. The authors can again collaborate on their revisions in real-time, before pushing them back to the journal for final review.

A synchronous solution means that any changes made to the files must be propagated as quickly as possible to all clients. The CAP Theorem [2] states that we cannot have consistency, availability and tolerance to partitions, but for a synchronous file system, we need all three. To achieve this, we use a weaker form of consistency called "eventual consistency," meaning roughly that if all users stop making edits to files, eventually the files on all clients will match. Saito and Shapiro [12] give a more formal definition.

The approach we take is optimistic, in that it assumes that conflicts (where two users edit the same part of the same file) are rare, and can be fixed after they happen [12]. Thus, users can edit any part of file at any time, and if two users happen to independently edit the same part of the same file simultaneously, the result is arbitrary, but eventually consistent.

The solution we propose is in user space, meaning that it runs as an application on the user's device, rather than being integrated directly into the operating system. It also allows for synchronization between a web interface and local files.

DChanges'16, September 13 2016, Vienna, Austria

ISBN 978-1-4503-4409-8/16/09…$15.00

DOI: http://dx.doi.org/10.1145/2993585.2993587

2. BACKGROUND

In our work, we bring together two approaches for achieving eventual consistency: Conflict-Free Replicated Data Types (CRDT) [16] and Operational Transformation (OT) [4].

2.1 Conflict-Free Replicated Data Types

Conflict-Free Replicated Data Types are constructed in such a way that operations on them cannot produce conflicts, regardless of their order, and so can be easily proven to result in consistent replicas for every user.

Shapiro et al. [16] have suggested several practical data types with this property, in particular, the Observed-Remove Set (OR-set) which will form the backbone of our file tree replication component. A set S has two operations, **add(e)** and **remove(e)**. If e is already in S then **add(e)** has no effect. If e is not in S then **remove(e)** has no effect.

Set operations seem like they might be commutative, but inconsistencies can arise. For example (borrowed from Shapiro et al.[16]), suppose user α adds the element 5 to the set, then removes it, resulting in the operation [**add(5)**, **remove(5)**]. Meanwhile user β has also added the element 5 to his set, resulting in the operation [**add(5)**]. Now, when α and β swap their operations, alpha will have [**add(5)**, **remove(5)**, **add(5)**] resulting in $S_\alpha = \{5\}$, whereas β will have [**add(5)**, **add(5)**, **remove(5)**] resulting in $S_\beta = \{\}$.

OR-Sets circumvent this problem by applying a unique token to each element. Operation **add(e)** generates a unique token and appends it to the element. This pair is then what is transmitted to all replicas. Operation **remove(e)** on the other hand removes all pairs containing e and trans-

Client α		Client β	
Operation	**Result**	**Operation**	**Result**
Initial	abc	Initial	abc
delete(1, 1)	bc	**insert(4, "ba")**	abcba
insert(3, 'e')	bce	**delete(3, 1)**	abba
transmit operations			
insert(4, "ba")	bceba	**delete(1, 1)**	bba
delete(3, 1)	bcba	**insert(3, 'e')**	bbea

(a) Without OT

Client α		Client β	
Operation	**Result**	**Operation**	**Result**
Initial	abc	Initial	abc
delete(1, 1)	bc	**insert(4, "ba")**	abcba
insert(3, 'e')	bce	**delete(3, 1)**	abba
transmit and transform operations			
insert(4, "ba")	bceba	**delete(1, 1)**	bba
delete(2, 1)	beba	**insert(2, 'e')**	beba

(b) With OT

Figure 1: Example of operations with and without Operational Transformation

mits those pairs to the other replicas. So, in the above example, the operations that α performs would be [**add((5,** t_1**))**, **remove((5,** t_1**))**, **add((5,** t_2**))**] resulting in $S_\alpha = \{(5, t_2)\}$, whereas β will have [**add((5,** t_2**))**, **add((5,** t_1**))**, **remove((5,** t_1**))**] resulting in $S_\beta = \{(5, t_2)\}$, where t_1 and t_2 are randomly generated tokens.

Although it is possible to apply a CRDT approach to editing a text buffer (e.g. [11]), this approach generates a lot of overhead and is not appropriate for binary files, as it involves assigning each byte a unique ID. There have been attempts to reduce the amount of bookkeeping CRDT requires in these cases (e.g. [9]), but a substantial amount of overhead remains. So, for operations on files themselves, we now consider Operational Transformation.

2.2 Operational Transformation

Operational Transformation was first proposed by Ellis and Gibbs in 1985 [4], but has seen renewed focus in the last decade and a half. Operational Transform allows for data types with non-commutative operations, but achieves consistency by transforming the operations when they are commuted.

For example, suppose that each user has a buffer B of data they wish to keep in sync. This buffer defines two operation **insert(pos, data)**, which inserts *data* at position *pos*, shifting the buffer after *pos* up by the length of *data* and **delete(pos, length)**, which removes *length* data from the buffer at position *pos*, shifting the buffer beyond *pos*+*length* down by *length*.

In Operational Transformation, the operations from a remote replica are transformed to take into account all prior operations before being applied. Figure 1 shows the difference between applying the operations as they are received (Figure 1a) or transforming them prior to applying them (Figure 1b). In particular, note how the position of each insert and delete maintains the intended meaning of the operation (e.g. the delete operation from Client α is intended to delete the 'c'.)

OT is more flexible than CRDT, but much harder to guarantee consistency. In 2006, Imine et al. [7] showed that many OT algorithms (in fact, all the algorithms they tested, including the original by Ellis and Gibbs [4]) were not guaranteed to converge to a consistent state. However, since then work has been done on achieving an algorithm with a provable guarantee on consistency. In particular, we shall use the Admissibility Based Sequence Transformation (ABST) algorithm proposed by Shao, Li and Gu [15] because of its linear running time and proof of correctness.

ABST is too complex to explain in detail here, but the fundamental principle is to maintain the operations performed in order first by type (inserts then deletes) and then by their position in the buffer. This allows for a transformation of operations similar to a two way merge, resulting in an algorithm linear in the number of operations to process and the number of operations in the history, as opposed to the original OT algorithm, which is quadratic.

1. Divide F_B into M equally size blocks of size n.
2. For each block at byte F_B^j, compute signatures R^j and H^j (where R is an easy to compute signature with many possible collisions, and H is a more complex signature with a lower probability of collisions).
3. Transmit the signatures to A.
4. At each byte offset i in F_A, compute R^i on the block starting at i
5. Compare R^i to each R^j. If there is a match, compute H^i and compare it to H^j.
   (a) If H^i matches H^j then record a block match between i and j.
   (b) Otherwise, if either R^i does not match R^j or H^i does not match H^j record the byte at F_A^i
6. Transmit the recorded matches and bytes to B
7. F'_B is constructed from merging the block matches from F_B and the new bytes transmitted from A.

Figure 2: The rysnc algorithm (based on Tridgell [17])

2.3 rsync

The rsync [17] algorithm can be used to keep two files on separate machines in sync with each other. The idea of rsync is to divide each file into blocks of a particular size and then compare the signatures of each block across the two files to see what changes have been made.

Suppose we have two computers, A and B, which each have a copy file F on them (F_A and F_B). Then if we wish to update F_B based on F_A, follow the steps in Figure 2

We will use this algorithm for finding how files have been updated on a local machine in Section 4.3.

3. RELATED WORK

The first attempts at remote file synchronization were based around networked file systems, such as Sun's Network File System (NFS) [13] and IBM's Server Message Block (SMB) [6], both of which are still in operation today, al-

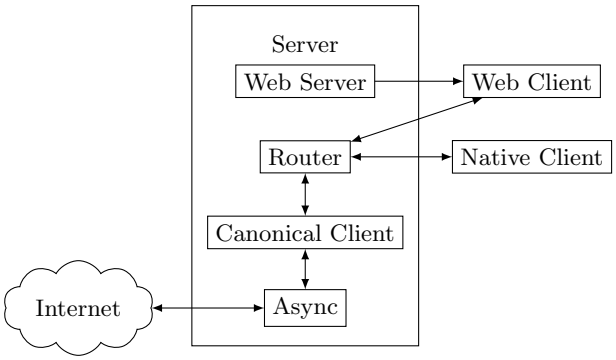
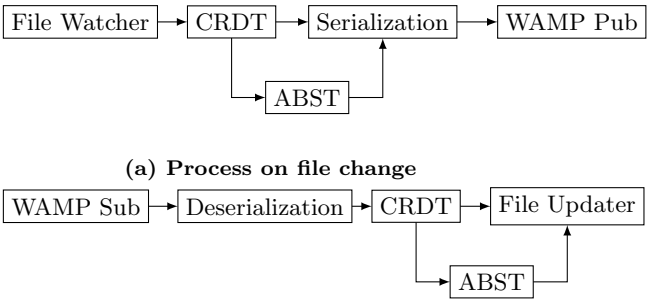


Figure 3: The architecture of the system



(a) Process on file change

(b) Process on receiving remote changes

Figure 4: Flow of data

though NFS has largely been replaced by (also created by Sun) zFS.

Other research has built on these systems, such as Coda [14], which allows workstations to continue working on remote files in the event of a disconnect and then synchronizes later, or Bayou [3], which allowed for a more peer-to-peer synchronization strategy.

Although rsync was and remains popular, other user space file synchronization approaches, such as Unison [10], or more recent commercial products such as Dropbox (which uses a modified rsync algorithm) and Google Drive (which uses Diff-Merge-Patch [5]) have cropped up and are widely in use.

Mehdi et al have suggested merging the Operational Transform and CRDT algorithms [8], but they use OT for keeping files synchronized between the client and the server, and CRDT for propagating changes between servers. Ahmed-Nacer et al. discuss applying CRDT to a file system [1], but focus on methods for resolving conflicts, and do not give more than a high level overview of how the file system might work. Our approach of treating Operational Transform as a commutative operation within CRDT is novel, and is the first contribution of this demo.

Furthermore, although the idea of version control (RCS, CVS, SVN, etc.) and even distributed version control (Git, Mercurial, Bitkeeper, etc.) is well established, combining real-time synchronization with asynchronous version control in this way is the second contribution of this demo.

4. APPROACH

4.1 Architecture

The system is comprised of five components: a web client, a web server, a native client, a real-time router and asychronous versioning (see Figure 3).

The web client provides web based editing and collaboration on files and folders as well as control over the asychronous versioning, while the native client runs on users' machines to broadcast changes to files made locally to other users.

The real-time router receives updates from the web and native clients, and distributes them to other users, while also keeping the most recent version of each file and folder updated and available.

At the user's request, the asynchronous versioning component uses the most recent version of the files and folders from the real-time router to create a snapshot of the files as they currently are, along with the difference between this and the previous snapshots. This history of file revisions can then be pushed or pulled to other servers, allowing for asychronous updates to be shared.

The web server is a standard web server that simply serves the static files used for the web client.

Communication between the clients and the realtime router, as well as between the asynchronous versioning components is provided by the Web Application Messaging Protocol (WAMP)[1]. WAMP is a protocol for both remote procedure call (RPC) and publish subscribe (pubsub) communications. It is available over websockets, and so provides two way communication for both the web and native clients.

4.2 Process

The file system is represented as an OR-Set, with each file having a unique token assigned to it that is independent of its filename or path. The token is generated from a locally unique value, alongside the ID of the client that created the file. The system is implemented as a map from the unique token to its corresponding file and meta-data. Alongside its binary octet stream, a file F has associated meta-data MD_F, represented as an Last Write Wins (LWW) Set [16]. This meta-data contains information such as filename, file path, and file type. Changes to the meta-data are made according to a simple timestamp based overwrite algorithm, described in Section 4.2.4.

The web client, when starting up, will subscribe to all updates for the object it is currently looking at (either a folder or a file F) from the real-time router, and also request the most recent version of F (using RPC).

The native client, when starting up, will scan all the files in its fileset, subscribe to changes to any file in the set, and request all remote changes to all files from the real-time router since the latest version it has received (using RPC). Upon receiving the remote changes, it will merge them with the local changes, and then publish all of the local changes for other users to see.

From here, both clients will track any changes to any file or folder F made locally (in either the web interface or to the file from an application on the local system). Any changes that are made will be published to the real-time router. Figure 4 describes the flow of data through the clients when the file changes. At present, all files are treated based on their

[1]See http://wamp-proto.org/

binary content, regardless of their file type.

Running alongside the router is a special client which keeps track of the canonical form of the object. This client is identical to a native client except that it exposes a procedure through RPC that allows other clients to retrieve the list of transactions since a partcular time.

When the user requests the version control component to take a snapshot of the current file, it consolidates all changes made to the file since the last snapshot, and stores those changes. If the user wishes, those changes can be sent to another server to apply to its own copy of the file. These changes are applied exactly as if they had come from another local machine, and so follow the same procedure as below.

We shall now discuss the operations on the files in more detail. There are four possible operations on a file system: create, delete, update and move.

4.2.1 Creating

A file F is created differently on a web and native client. On a web client, the user requests the file's creation, whereas on the native client, the user creates the file using some external application. In either case, a new, globally unique token t_F is generated for that file (as outlined above), and an **add(t_F)** operation is published to the real-time router, along with the relevant meta-data (name and path).

The token t_F and its meta-data are stored in the file set locally, all subscribers add it to their local set and any native clients will physically create the file. If two users independently create a file with the same name on different systems, they will not overwrite each other, since each file will be created with a unique token. However, when the client receives the **add** command, if a file with the same name already exists, the filename will be modified locally to indicate that it is a different file that happens to have the same name. For example, if client A and client B each created a file called "info.txt" before synchronizing, then after the operations had propagated, client A would have two files, named "info.txt" and "info (from B).txt" and client B would have two files, named "info.txt" and "info (from A).txt".

Each client is responsible for keeping track of the mappings between actual filenames and what is displayed.

4.2.2 Delete

When a file F with token t_F is deleted, the client will remove it from its local set, then publish a **remove(t_F)** message. Upon receiving it, the router and every subscriber will physically delete F from the file system and remove t_F from its local map.

4.2.3 Move

Moving a file on the file system can be represented either as a change to the meta-data map if they remain within the tracked folder or a delete if it is leaving the tracked area.

4.2.4 Update Meta-data

When changes to the meta-data take place, an **updateMD(t, k, v, ts)** message is published, where t is the token corresponding to that file, k is the meta-data key (such as 'filename' or 'path'), v is the new value for that meta-data element, and ts is the local timestamp. When the router or any other client subscribing to that file system receives the update message, the meta-data entry k in t is changed, so long as the timestamp ts is more recent than the timestamp on the previously stored value. In this way, the value stored in the meta-data map will be the one with the most recent timestamp. This may cause some strange behaviour when the clocks on the computers are out of sync, but most computers are close enough that it shouldn't be a problem.

If the meta-data update changed the path or filename of a file, then that file is renamed or moved, unless there is already a file in the new location. If that is the case, then the file name is changed according to the rules outlined in section 4.2.1.

4.2.5 Update Data

As mentioned earlier, when updating data within files, we use Operational Transform to maintain consistency. Since the goal of Operational Transform is to end with the same result regardless of the relative order of operations, we can treat Operational Transform as a kind of Conflict Free Replicated Data Type.

When updating a file, there are two possible operations: **insert** and **remove**. **change** is a possible operation as well, but can be expressed as a **delete** followed by an **insert**.

For each file F, on every client and on the router, each operation since the previous snapshot is stored. Following the ABST algorithm [15], we store the operations in file order, rather than the order they were received, first the inserts, then the deletes. This means that each operation happens earlier in the file than the one that follows it.

When receiving a series of operations from a remote site, the file is locked locally, the remote operations are compared against the local operations, and 'merged in,' so that the new sequence of actions retains the property of being stored in the order they occur in the file. The merged operations are performed on the file itself, and the file is unlocked. Shao, Li and Gu [15] give the precise details of the merge.

4.3 Detecting Changes

On the web interface, changes can be detecting by tracking events, but this is not possible in the native client when the changes are made in external applications. Instead, we apply a modification of the rsync algorithm outlined above to find the changes. For each file, we store block hashes of the latest version. When a file is modified, the client treats the existing hashes as F_B according to the algorithm in figure 2, except line 4.(a) is modified to read If H^i matches H^j and $i \geq j$ then record a deletion of $i - j$ bytes.

This change removes the ability to handle re-ordering within the file gracefully, but the OT algorithm does not deal well with re-ordering (treating it instead as a delete and insert).

5. CONCLUSION

The system as described brings together two consistency approaches (OT and CRDT) to synchronize changes across local files and a web client. It is also able to propagate those changes to other remote servers.

Before the system can be truly useful, several more problems must be resolved. The large size of operation history, more appropriate difference algorithms for tree and text based files, and improved conflict resolution for the asynchronous component are all areas of future improvement.

6. REFERENCES

[1] Mehdi Ahmed-Nacer, Stéphane Martin, and Pascal

Urso. File system on CRDT. Technical report, INRIA Lorraine - LORIA, jul 2012.

[2] Eric A. Brewer. Towards robust distributed systems. In *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing - PODC '00*, page 7, 2000.

[3] a. Demers, K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and B. Welch. The Bayou Architecture: Support for Data Sharing Among Mobile Users. *1994 First Workshop on Mobile Computing Systems and Applications*, pages 2–7, 1994.

[4] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems, 1989.

[5] Neil Fraser. Differential Synchronization. *Proceedings of the 9th ACM symposium on Document engineering*, pages 13–20, 2009.

[6] Christopher R. Hertel. *Implementing CIFS: The Common Internet File System*. Pearson Education, Inc., Upper Saddle River, New Jersey, 2004.

[7] Abdessamad Imine, Michaël Rusinowitch, Gérald Oster, and Pascal Molli. Formal design and verification of operational transformation algorithms for copies convergence. In *Theoretical Computer Science*, volume 351, pages 167–183, 2006.

[8] Ahmed-Nacer Mehdi, Pascal Urso, Valter Balegas, and Nuno Perguiça. Merging OT and CRDT algorithms. In *Proceedings of the First Workshop on Principles and Practice of Eventual Consistency*, pages 8–11, New York, New York, USA, 2014. ACM.

[9] Brice Nédelec, Pascal Molli, Achour Mostefaoui, and Emmanuel Desmontils. Concurrency Effects Over Variable-size Identifiers in Distributed Collaborative Editing. In Gioele Barabucci, Uwe M. Borghoff, Angelo Di Iorio, and Sonja Maier, editors, *Proceedings of the International workshop on Document Changes: Modeling, Detection, Storage and Visualization*, Florence, Italy, 2013. CEUR Workshop Proceedings.

[10] Benjamin C. Pierce and Jerome Vouillon. What's in Unison? A Formal Specification and Reference Implementation of a File Synchronizer. Technical report, Dept. of Computer and Information Science, University of Pennsylvania, 2004.

[11] Nuno Preguiça, Joan Manuel Marquès, Marc Shapiro, and Mihai LeÅčia. A commutative replicated data type for cooperative editing. In *Proceedings - International Conference on Distributed Computing Systems*, pages 395–403, 2009.

[12] Yasushi Saito and Marc Shapiro. Optimistic replication, 2005.

[13] R. Sandberg, R. Sandberg, D. Goldberg, D. Goldberg, S. Kleiman, S. Kleiman, D. Walsh, D. Walsh, B. Lyon, and B. Lyon. Design and implementation of the Sun network filesystem, 1985.

[14] Mahadev Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39:447–459, 1990.

[15] Bin Shao, Du Li, and Ning Gu. A fast operational transformation algorithm for mobile and asynchronous collaboration. *IEEE Transactions on Parallel and Distributed Systems*, 21(12):1707–1720, 2010.

[16] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Convergent and Commutative Replicated Data Types. *Bulletin of the European Association for Theoretical Computer Science*, 104:67–88, 2011.

[17] Andrew Tridgell. Efficient Algorithms for Sorting and Synchronization. *Doktorarbeit Australian National University*, (February), 1999.

[18] Benjamin W. Wah. File placement on distbuted computer systems. *Computer*, 17(1):23–32, 1984.