ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Access Control in a Decentralized Collaboration Platform

Nicolas Ritter

School of Computer and Communication Sciences

Decentralized and Distributed Systems lab

Semester Project

January 2018

Responsible
Prof. Bryan Ford
EPFL / DEDIS

Supervisor
Kirill Nikitin
EPFL / DEDIS

Contents

1 Introduction

Online editing platforms such as Google Docs are extremely popular and useful tools for sharing documents and collaborating on them with other people. The ability for multiple users to work on the same document simultaneously in real time and the simplicity of not having to deal with some kind of version control software are extremely attractive. However there are concerns which come with the usage of such services:

- Having to share potentially sensitive data with a third party which may or may not be trustworthy
- Having to rely on a central server, which is a single point of failure
- Not having local control and ownership of the data

These concerns are often a matter of principle, coming from an objection against giving private information to third party corporations such as Google for them to use as they please. But these concerns can also be pragmatic: online services can experience downtime, get shut down, or attacked. They can get blocked by states or internet service providers, or on the contrary help governments spy on their citizens. The bottom line is, if a centralized service can be made decentralized there are many reasons that can make doing so an interesting proposition.

The Peerdoc platform attempts to provide a Google Docs-like environment in an entirely decentralized manner. In particular, the central task of this project is to implement a form of access control to the existing application, as well as improving the overall functionality and usability of the system to bring it closer to a fully functional application.

2 Goals and motivations

The overall goal of this project is to devise access control mechanisms and implement them into the existing Peerdoc program. Access control is a crucial component of any real-world collaboration platform; a user of such a platform will only want explicitly authorized users to access and modify their document. However enforcing access control on a peer-to-peer network isn't trivial as there is no central authority that can be relied on.

A goal of the system is that it should be able to recover from network partitions. In a decentralized system, we cannot guarantee that the network remains connected uninterruptedly or that there aren't significant delays. In fact, it is even expected that users will usually be offline and only come online sporadically. So if there is a fork in the state of the access control it is important that all peers eventually converge to the same state again.

Finally one goal of this project is also to improve certain general aspects of the Peerdoc platform (which is still very much in development) by moving its database from the front end to the back end, allowing nodes to dynamically join the document and catch up on its content, allowing documents to be opened and closed, and so on.

3 Background

Peerdoc is a collaborative platform which is currently in development. It consists of a browser-based user interface written in HTML/Javascript and a back end written in Go, which comprises several modules. The architecturally central module is called Management and serves as the link between front end and back end, notably the ABTU module which implements the algorithm which integrates local and remote operations to the local copy of the document. This algorithm was based on existing research papers [1] [2] and was implemented by another student as part of a previous semester project.

The workings of the ABTU algorithm are based on operational transformations. Each operation represents one change made to the state of the document (either the insertion or deletion of a character), and multiple operations are ordered by using vector clocks. The idea of this project is to translate this notion of operation from the ABTU to an access control module where an operation corresponds to a change of permissions.

Multiple papers have been written on decentralized collaborative platforms, as well as on access control in such platforms. Peerdoc takes inspiration from optimistic algorithms, that is algorithms which apply operations locally first and are able to roll back or modify the local state if a conflict arises. Some designs such as SPORC [3] rely on an (untrusted) central server while others use operation validation based on time thresholds [4], both of which we decided to avoid as the platform is meant to be truly asynchronous and decentralized.

4 Design and implementation

4.1 Overall architecture

A schema of the application is shown in figure 1. Most of it hasn't changed from the previous implementation, with the exception of the addition of the access control module and of a SQLite database in the back end (to replace the previous front-end CouchDB database). Communication between management and front end is done through a websocket, and communication with ABTU and access control is done through Go channels as both modules run independently on separate threads.

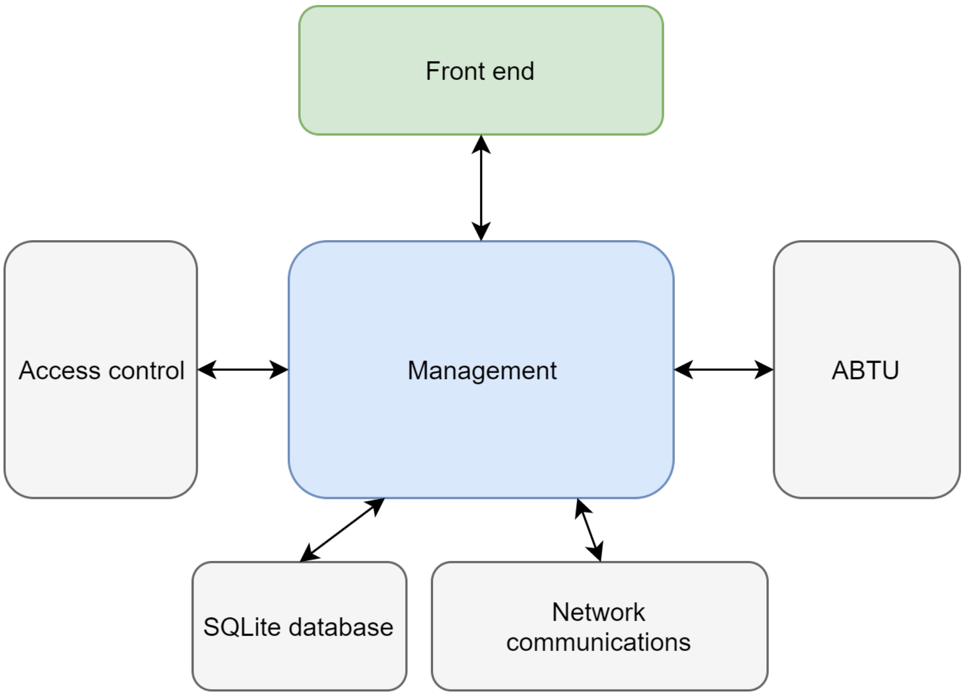

Figure 1: Architecture of the program

4.2 Basics

The basic workings of the system is that each peer maintains a state for each document which associates users with UNIX-style permissions (where 4 is read-only, 6 is read/write and 7 is administrator privileges). When receiving an operation (either text or access control) a peer checks whether it is allowed according to its local state of permissions.

4.3 Access control operations

An access control operations is similar to a text operation in that it uses vector clocks for its ordering, originates from a specific peer (this identifier is assumed to eventually also serve the function of a public cryptographic key) and applies to a specific document. The difference of course is that instead of carrying a character and its position, it carries a user and their permissions. There exist only one type of access control operation as everything can be described as a change of permissions. Adding a user to the network is done by granting them permissions, and removing a user by setting their permissions to 0. For this reason access control operations can also contain the IP address of the target peer, so that the program can set up communications with a new peer when it has been added.

Access control operations also contain a special timestamp which points to the place in the history of text operations at which it was issued. This pointer is needed to order a given access control operation relative to the history of text operations, since text operations and access control operations run on separate vector clocks concurrently. The idea is that we do not want the ABTU algorithm to have to wait on access control operations, but when an access control operation occurs we need to know the point in time at which it becomes effective since it affects the legality of text operations. This is needed so that we can then cancel any illegal text operation after it has been applied.

4.4 Ordering of access control operations

Unlike text operations which are heavily dependent on each other (e.g. the position of an inserted character depends on characters that were inserted before it), access control operations are fairly independent. They do not need to be altered to account for other operations before being applied; they only need to be ordered properly, that is respecting the order in which they were issued. Thus the handling of access control operations is much simpler than the ABTU algorithm. Unless the operations are concurrent (i.e. they branch from a common state independently), it is simply a matter of respecting the order defined by the vector clock. Behavior in case of a fork will be explained in section 4.6.

4.5 Canceling text operations

4.5.1 Description

Text operations are only ordered with respect to each other and are applied optimistically without waiting for access control operations. Since access control operations are expected to be much less frequent than text operations this usually doesn't cause any issues. However this means that sometimes an access control operation will make one or more already-applied text operations illegal. In such cases, these illegal operations need to be rolled back. We will call this behavior the *cancellation* of operations, as it is different from the ABTU algorithm's *undoing* mechanism.

The difference lies in that canceling an operation must be invisible, in the sense that operation history must then look as if the operation had never been applied. In contrast, undoing an operation is actually an operation of its own. The reason for this difference is that some peer P_i might have applied a text operation from site s and increased its vector clock at $SV_i[s]$ while another peer P_j might have applied the access control operation first, dropped the illegal operation from s and thus never incremented $SV_j[s]$. Eventually we want these vector clocks to converge.

4.5.2 Mechanism

When an access control operation which removes a user's permissions is received, it is forwarded to the ABTU thread. Since the access control operation carries a pointer to the time in the history of text operations at which it was issued, the ABTU thread can check whether any subsequent text operations are now illegal. To do so it reads the operation history backwards as long as the current operation has a higher timestamp that the pointer. If an operation originating from the user which had their permissions removed is found, it is deleted. Then every subsequent operation has its character position shifted and timestamp decreased to account for the removal of the illegal operation.

The cancellation process also goes through the buffer of incoming operations and drops any operation which has become illegal, as we don't want them to be stuck in the buffer forever.

4.6 Catching up and recovering from partitions

4.6.1 Catching up

What happens when a node joins (or rejoins) a document that has been worked on during its absence? In the ideal case this isn't a problem; nodes can simply exchange their respective statuses on the history of operations (a status simply contains the local vector clock corresponding to the document). Then, upon receiving a status, a node sends all operation which are ahead of it, if any. However in case of a network partition (or simply when two peers issue an operation simultaneously) it is possible that both sides of said partition have independently added operations to their respective histories. When this happens it is necessary that both nodes can again converge to the same state from their different histories.

4.6.2 Priority rules

It is crucial that concurrent access control operations (that is operations whose timestamps can't be ordered relative to each other) can still be ordered according to a set of deterministic rules, so that all nodes can converge to the same state after a partition.

In the case of text operations, it is possible to transform conflicting operations and apply them all sequentially, which is what the ABTU algorithm does. For instance when an operation inserts character "a" and another inserts character "b" in the same position, both characters can be inserted one after the other according to a defined order. With access control operations however it is clear that some operations will simply override others when applied. For instance if one operation sets some peer's permissions to read-only while another sets them to read/write, then whichever is applied second will override the first. Therefore it is important that concurrent access control operations are ordered in a way that is not only deterministic but also sensible.

For two concurrent operations Op_a and Op_b we use these rules one after the other, until one is applicable:

- $Op_a < Op_b$ if $distance(Op_a, 0) < distance(Op_b, 0)$
- $Op_a < Op_b$ if $Op_a.peer = Op_b.peer$, and $Op_a.permissions > Op_b.permissions$ (this includes cases where a peer is removed, i.e. permissions = 0)
- $Op_a < Op_b$ if $Op_a.source = Op_b.peer$ and $Op_b.permissions \neq 7$
- If no other rule apply, $Op_a < Op_b$ if $Op_a.source < Op_b.source$ according to the lexicographic order.

Where $Op_a < Op_b$ means that Op_b overrides Op_a, *peer* denotes the peer subject to the operation, *permissions* the permissions which are set to this peer, *source* the source of the operation and $distance(Op_i, 0)$ the sum of distances between each value of Op_i's vector clock and 0.

After a concurrent operation has been inserted in the access control history, the algorithm checks if it has an effect (i.e. no subsequent operation in history affects the same peers permissions, and the operation has full permissions at this point in time). If it does, management is notified so that it can apply the change to the local access control state.

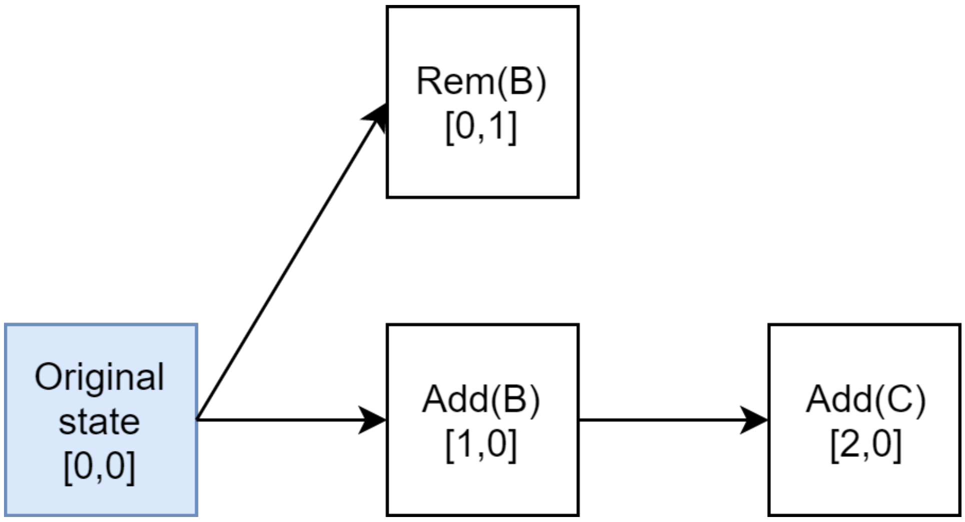

Figure 2: A concurrent operation is received

Figure 2 shows a fork created by a concurrent operation. ”Rem” and ”Add” correspond to the removal of a peer and the granting of some permissions to a peer, respectively. The numbers represent the vector clock of each operation. Figure 3 shows the state of operation history after the concurrent operation has been inserted according to the priority rules.

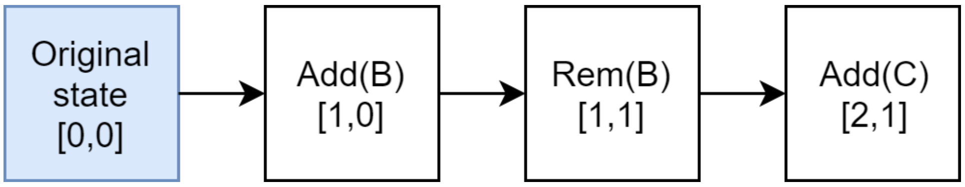

Figure 3: The concurrent operation has been inserted

4.7 Miscellaneous work

4.7.1 Database

Previously the Peerdoc app stored its data on a CouchDB database which was connected directly to the Javascript front end. It was replaced with a SQLite database and moved to the back end, which is easier to work with and makes more sense.

The amount of stored data has also been improved. Previously Peerdoc would only store documents, their content and the list of collaborators but now the program also stores the history of operations of each document and the local timestamps, which are then loaded when the document is re-opened. This allows the collaboration to continue seamlessly across executions.

4.7.2 Improving robustness and usability

Peerdoc was still in a prototypical stage (and to an extent still is) so some work was spent on general improvements and fixes to the program. Namely the app is now aware of which document is open, and allows the user to switch documents. The TCP communication code using the Peerstore library was replaced by a simpler UDP protocol, and it now allows the set of peers to be changed during execution. In practice this means that new peers can now join or leave the network, and that the application will no longer crash as a result of not being able to establish communication with a particular peer.

The peers' identifiers were changed from integers to strings. This change means that timestamps are now represented as maps instead of integer-indexed arrays, and are no longer required to all keep the same constant length (which allows new peers to join the system). Currently this string is arbitrary but in the future (when cryptography is implemented) it could be synonymous with (or derived from) the peer's public key.

5 Limitations

5.1 Practical limitations

The program serializes data as JSON to pass it to channels between the ABTU (or access control) and the front end. The assumption was that the central management would mostly just forward this data between channels and sockets and wouldn't need to access its content. This assumption, however, has proven less and less true as the complexity of the program increased. As it stands, using JSON byte arrays to communicates between modules provides little in the way of usefulness and on the contrary makes it easy to make mistakes when writing code. Replacing these byte channels with channels of plain structs would be a huge improvement to the code.

The current system only processes operations pertaining to the currently open document, as opposed to being able to receive background updates on multiple documents. However the catch-up mechanism will fetch all required operations as soon as a document is opened, assuming another peer has them.

Certain front end features such as deleting or renaming documents aren't yet adapted to function with the new back end database. The top menu is still a placeholder and could be used for notifications, account management, etc.

5.2 Theoretical limitations

Priority of operations cannot be cleanly defined in certain corner cases. For instance if there are two administrators who demote each other concurrently, these operations are mutually exclusive and there is no way for an algorithm to decide which one to follow (without being unfair and arbitrary). These cases might require some kind of prompt to let the user decide which operation to accept.

It is also impossible to differentiate between an actual network partition and a malicious peer creating a fork on purpose (for instance to deliberately ignore their own loss of privileges). Again choosing whether to accept an outdated operation might require user intervention in some form.

6 Results

The project was successful in implementing access control to the platform as well as improving the overall functionality of the program (although there is still work to be done and not all corner cases are covered). The section below will detail the performance achieved by some of the new mechanisms added to the program.

6.1 Performance evaluation

6.1.1 Catch-up mechanism

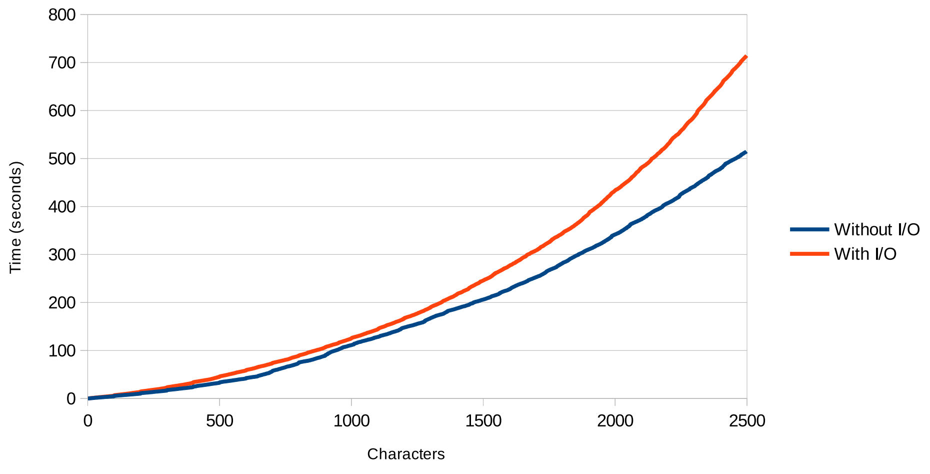

Figure 4: Catch-up performance

The setup for the catch-up mechanism consists of one Peerdoc instance P1 which has written a string of 2500 characters to the document (which can be considered to be at least one decent page of text), and another instance P2 running on the same machine which joins afterwards (and which sends its status upon connecting to the document).

To measure the performance of the catch-up system, time was first recorded in P2 right after it sent its first status, then when each remote operation reached P2's management after having been processed by the ABTU algorithm. One test was run with the program as is and another with writes to the SQLite database removed. The results are shown in figure 4.

The first important conclusion to make is that catch-up time isn't quite linear in the number of characters, which isn't ideal, and overall the process is rather slow. The program makes a lot of writes to the database in its current state since it doesn't have a way to exit gracefully. It is clear that removing most of these writes in favor of only saving the data when the program is closed would make the process of integrating operations faster.

The biggest bottleneck probably comes from the fact that the ABTU implementation has to send every remote operation to the Javascript front end and wait for its acknowledgement before proceeding to the next operation. This is a slow process, and the increasing cost of inserting or deleting a character in the front-end document as it becomes longer most likely plays a big part in the progressive decrease in performance.

Another possible cause for the relative slowness of the process is that operations are sent individually. If they were packaged into bigger UDP packets or if some sort of snapshot of the entire document was shared in some cases instead of operations this could speed up the process considerably.

6.1.2 Operation canceling

A test was done on a node that had a history of one text operation from peer A and 2500 text operations from peer B. The node then received an access control operation which made the operation from peer A illegal and thus required a cancel. It took 0.02 second for the algorithm to scroll through history backwards, remove the illegal operation and shift all subsequent operations in history. This first step was very fast since all operations from B were immediately detected as legal, and scrolling through history again to shift their positions and clocks isn't much more expensive either.

However it took another 96 seconds to send the changes to the front end. These changes involve rolling back the document to the point before the illegal operation was applied then rewriting it, because the position of the character that corresponds to the illegal operation isn't trivially known as it can be affected by every subsequent operation.

It is clear from both these experiments that the protocol between back end and front end works well to send individual operations at a reasonable pace but that it isn't quite adapted to receiving large batches of operations at once. In the future implementing a system where the state of the document is computed in the back end then sent to the front end as one chunk of text might be more efficient at handling such cases.

7 Future work

There is still work left to be done on the Peerdoc platform to make it truly usable as an actual tool for collaborative work.

The most important task will be to implement encryption mechanisms into the program, as access control in a system which uses unencrypted, unauthenticated messages is hardly effective. The system will have to allow for the cryptographic key to be changed whenever a user has their access to the document removed, since they then must not be able to read any following updates to the document. In the current state of the program nothing prevents the forging of operations. On a related note there will also need to be an actual secure and user-friendly way to log in to the system.

Cursor position sharing is not implemented yet. This feature would be similar to Google Docs and would allow users to see what other people are working on by displaying their cursor on the document.

Finally there will need to be practical ways to share a document, which wouldn't require the user to input the recipient's IP address. Also as it stands a document has to be open for a peer to receive operations on it, which means that the document has to already exist on each node for them to start sharing updates. Ideally new documents should be shared with peers automatically for the system to be really usable.

8 Installing and running the program

- The program is written in the Go programming language:

  https://golang.org/dl/

- The source code is available at the following GitHub repository:

  https://github.com/nikirill/peerdoc

- It also require the SQLite driver library found here:

  https://github.com/mattn/go-sqlite3

- As well as the Gorilla websocket library:

  http://www.gorillatoolkit.org/pkg/websocket

- Finally SQLite3 needs to be installed on the machine. It can be found here:

  https://www.sqlite.org/download.html

To run the program, execute the following command:

```
go run main.go
```

To specify a local identifier "id", use the command-line argument

```
-siteId="id"
```

And to specify the local IP address, use

```
-address="adress:port"
```

If these arguments aren't used, the program will default to identifier "abc" and address 127.0.0.1:1234.

The "UIPort" flag can also be used to specify the port used to communicate with the Javascript front end (which defaults to 5050) but changing requires the Javascript code to be modified accordingly (in index.js and editController.js).

References

[1] Du Li and Rui Li. *An admissibility-based operational transformation framework for collaborative editing systems.* 2010.

[2] Bin Shao, Du Li, and Ning Gu. *An algorithm for selective undo of any operation in collaborative applications.* 2010.

[3] Ariel J. Feldman, William P. Zeller, Michael J. Freedman and Edward W. Felten. *SPORC: Group Collaboration using Untrusted Cloud Resources.* Princeton University, 2010.

[4] Asma Cherif, Abdessamad Imine and Michal Rusinowitch. *Optimistic Access Control for Distributed Collaborative Editors.* French Institute for Research in Computer Science and Automation, 2011.