FACULTY OF ENGINEERING AND ARCHITECTURE

Real-Time Collaboration in Jupyter Notebooks

Jan Mariën
Student number: 01507966

Supervisors: Prof. dr. ir. Filip De Turck, Prof. dr. Bruno Volckaert
Counsellors: ing. Merlijn Sebrechts, Sander Borny

Master's dissertation submitted in order to obtain the academic degree of
Master of Science in Computer Science Engineering

Academic year 2020-2021

GHENT UNIVERSITY

FACULTY OF ENGINEERING AND ARCHITECTURE

Real-Time Collaboration in Jupyter Notebooks

Jan Mariën
Student number: 01507966

Supervisors: Prof. dr. ir. Filip De Turck, Prof. dr. Bruno Volckaert
Counsellors: ing. Merlijn Sebrechts, Sander Borny

Master's dissertation submitted in order to obtain the academic degree of
Master of Science in Computer Science Engineering

Academic year 2020-2021

GHENT UNIVERSITY

Preface

I would like to thank my counsellors ing. Merlijn Sebrechts and ing. Sander Borny for guiding me through this process and providing extensive feedback. In addition I would like to thank my girlfriend Elise for her unconditional support, Ronald for providing feedback and finally everyone that participated in the usability study.

Mariën Jan
May 30, 2021

Abstract

Computational notebooks are more popular than ever. Many courses such as data science and artificial intelligence use these tools for illustration and assignments. In times where students are distanced due to an unforeseen pandemic and forced to work from home, it is even more important that these students receive proper assistance. Support for Real-Time Collaboration (RTC) could be very beneficial to both teachers and students when illustrating course material or working on remote assignments. However, open source solutions such as Jupyter notebooks lack such functionality. This thesis researches the possibilities of implementing such features in Jupyter notebooks. A comparative study is conducted between two potential RTC libraries: Y.js and Microsoft Fluid. By analysing both usability and performance aspects, Y.js is selected as the best candidate. Using this library a proof-of-concept prototype, which brings RTC capabilities to JupyterLab, is developed. Based on the results of this thesis it is concluded that both the current technology and notebooks such as Jupyter's are ready for Real-Time Collaboration. It is not a question if, but when Jupyter notebooks will offer RTC to the public.

Keywords— Computational notebooks, Real-time Collaboration, Jupyter, Y.js, Fluid framework

Real-Time Collaboration in Jupyter Notebooks

Mariën Jan
Supervisors: Prof. dr. ir. Filip De Turck, Prof. dr. Bruno Volckaert
Counsellors: ing. Merlijn Sebrechts, ing. Sander Borny
Faculty of Engineering and Architecture
Ghent University

***Abstract*—Computational notebooks are more popular than ever. Many courses such as data science and artificial intelligence use these tools for illustration and assignments. In times where students are distanced due to an unforeseen pandemic and forced to work from home, it is even more important that these students receive proper assistance. Support for Real-Time Collaboration could be very beneficial to both teachers and students when illustrating course material or working on remote assignments. However, open source solutions such as Jupyter notebooks lack such functionality. This thesis researches the possibilities of implementing such features in Jupyter notebooks. A comparative study is conducted between two candidate RTC libraries, Y.js and Microsoft Fluid. By analysing both usability and performance aspects, Y.js is selected as the best candidate. Using this library a proof-of-concept prototype, which brings RTC capabilities to JupyterLab, is developed. Based on the results of this thesis it is concluded that both the current technology and notebooks such as Jupyter's are definitely ready for Real-Time Collaboration. It is not a question if, but when Jupyter notebooks will offer RTC to the public.**

I. Introduction

In recent times, online lectures and exercises have become more and more prevalent. This new way of educating, however, comes with many challenges. Data science courses, for example, make use of computational notebooks which guide students through an assignment. Nonetheless, assisting students while teaching remotely is often a struggle. Real-Time Collaboration (RTC) could be a valuable addition to this process for both students and teachers. Currently computational notebooks have limited support for RTC and at the time of writing only commercial solutions seem to have such functionalities. Open-source alternatives such as Jupyter notebooks, however, lack support for RTC at this time.

So what are the difficulties when sharing a notebook in real-time? And how can an open-source solution such as Jupyter implement Real-Time Collaboration in their notebooks? The first section of this paper gives a deeper understanding of computational notebooks and Real-Time Collaboration. The algorithms behind the technology of RTC are discussed and a state-of-the art survey is conducted. The second part of this thesis works towards a proof-of-concept extension that implements RTC in Jupyter notebooks. Firstly, a comparative study between two candidate libraries is conducted. Both performance and usability aspects of these frameworks are tested and based on these results, a library is selected to build the prototype upon. In what follows, the proof-of-concept extension, which is developed as part of this thesis, is thoroughly discussed. A final section summarises the results and concludes this paper.

II. Computational Notebooks

Computational notebooks are often compared to a science lab report. In those reports, data, calculations and figures are written down but the most important aspects of such reports are the result and understanding of the experiment. Computational notebooks try to achieve the same by combining executable code and markdown, a markup language often used for documentation, into one visual narrative. Such visual narratives can easily be read by users of a different background without them having to understand the specifics of the code in this notebook.

Throughout the years, such notebooks have become increasingly popular. Research by P. Parente [1] indicates that the estimate amount of Jupyter notebooks circulating on GitHub is currently over nine million.

III. Real-Time Collaboration

Real-Time Collaboration can take many forms. Examples vary from live messaging to real-time text editing software like Google Docs. For the remainder of this thesis, RTC is referred to as the ability to perform simultaneous coding and text editing

A. Challenges

Several challenges pose themselves when multiple users are editing the same document. Documents being out of sync or changes by certain users not being saved is to be avoided at all time. In addition to this, permissions need to be considered as well. Which user has access to which document and does this user have write or read-only access? The next section lists the most common algorithms which deal with synchronisation of real-time changes. Challenges regarding security are considered out of scope for this thesis.

IV. Algorithms

Three main algorithms can be distinguished. **Operational Transformation** (OT) originated in 1989 as underlying algorithm for the GROVE editor by Ellis et al [2]. It was originally called dOPT and relies on a transformation function to ensure convergence. This initial algorithm has several

problems which resulted in many adaptions and variations which try to fix these [3]. Examples are REDUCE [4] and the Jupiter algorithm [5].

Although CRDTs already existed way earlier [6], Shapiro et al. [7] formalised this concept in 2011. A **Conflict-Free Replicated Data Types** (CRDT) is defined by the authors as a data structure for replicating data in a distributed system.

Differential Synchronisation is a difference based synchronisation algorithm that was invented by N. Fraser [8]. One of its well know implementations is Google's diff-match-patch[1] which formed the early basis for Google Docs' real-time collaboration.

V. State-of-the-art

In this section, some state-of-the-art libraries and software solutions are listed. Microsoft Fluid and Y.js are thoroughly described as they are the subject of the comparative study in the following section.

A. Microsoft Fluid

The Microsoft Fluid Framework [9] is a set of TypeScript (JavaScript) libraries that allow developers to build collaborative web applications. It was only recently made open-source and available to the public. Fluid offers RTC through the use of so-called Distributed Data Structures (DDSs). They are very similar to regular objects (e.g. arrays and strings) used when programming. However, these objects allow to be edited by multiple users and automatically synchronise their state across all clients. The algorithm behind these shared structures is neither CRDT nor OT but the model is *more similar to CRDT than OT* according to the developers themselves [10]. Figure 1 provides an overview of the architecture [11] of this framework. Three components can be distinguished:

- The **Fluid Loader** which is responsible for loading a Fluid Container.
- A **Fluid Container**, which consists of at least one Fluid Object which holds one or more DDSs. The main task of this container is to implement the application logic at client side.
- The **Fluid Service**, which orders incoming operations and distributes them to all clients. Note that this service does not process any of these operations and thus is not aware of the contents of a Container.

B. Y.js

Y.js is a JavaScript library that implements a CRDT to provide users with *shared types*. It supports several shared types such as text, arrays and maps. Contrary to Microsoft Fluid, Y.js is purely P2P and does not necessarily rely on a server component. The CRDT implementation is based on the YATA

[1]https://github.com/google/diff-match-patch

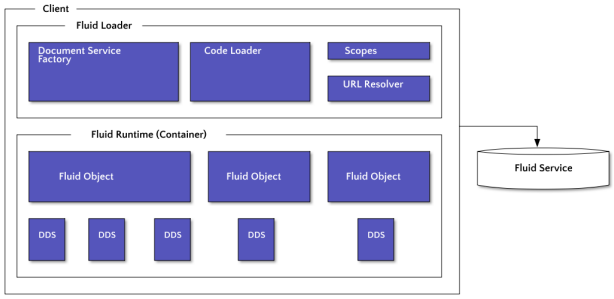


Fig. 1. Architecture of the Microsoft Fluid Framework [11].

(Yet Another Transformation Approach) by P. Nicoleascu et al. [12]. However, K. Jahns, the author of Y.js, added several modifications which makes the library more efficient. The changes are precisely explained in his blogpost [13]. Together with Y.js the author also provides several bindings for existing text editors and server implementations to distribute changes.

C. Others

1) Libraries: In addition to Microsoft Fluid and Y.js, many other JavaScript RTC libraries are available. The **AutoMerge** library implements a CRDT based on the JSON-CRDT by Klepmann et al. [14] [15]. **ShareDB** implements a real-time database that is very convenient for usage with the *Operational Transformation*. It was originally developed to store and synchronise JSON documents, but currently allows for all sorts of documents using so called *shared-types*.

2) Notebook software: As mentioned before, open-source solutions for notebooks are scarce. Commercial solutions such as **CoCalc** and **Apache Zeppelin** do have support RTC in computational notebooks. **Google's Colab** offered RTC functionality but removed this feature after the API driving this was deprecated by Google.

3) Text-editing: Rich-text editors with support for RTC are ubiquitous. In the shadow of famous examples such as **Google Docs** and **Microsoft Office**, open-source software such as **EtherPad** and **FirePad** target themselves specifically as RTC editors.

4) Coding: Several popular IDEs already have support for RTC. **Visual Studio Live Share**[2] is part of Microsoft's Visual Studio and can also be downloaded as an extension to Visual Studio Code. JetBrains, known for IDEs such as IntelliJ and PyCharm, recently introduced **Code With Me**[3] and Atom already supports RTC for several years using **TeleType**[4].

[2]https://visualstudio.microsoft.com/services/liveshare/
[3]https://www.jetbrains.com/code-with-me/
[4]https://teletype.atom.io/

VI. Comparison of Y.js and Microsoft Fluid

In this section, the comparison between the Microsoft Fluid Framework and Y.js is described. These two frameworks are chosen based on several aspects. First of all, because Jupyter's front-end is written in TypeScript (a superset of JavaScript) this is the preferred language. Secondly, Microsoft Fluid is a quite recent library and thus makes an interesting candidate for a performance evaluation. Lastly, Y.js has already been tested and came out as best comparing to current CRDT JavaScript libraries. It thus makes sense that we compare the results of Fluid to one of the currently most-performant libraries.

A. Performance

As part of this thesis, the current benchmark for Y.js, developed by Jahns[5], is extended to include the Fluid framework. The benchmark measures several performance aspects such as synchronisation time, document size, update size and average memory usage. The benchmark and results can be found on GitHub[6]. In addition to this benchmark, some specific scenarios are tested and discussed below.

1) Insert/remove N words (2 clients): One common scenario is the case where a teacher is guiding a student through the notebook by means of example. In more technical terms this means that two clients are connected but only one of them is writing to the shared document. Results show that Y.js outperforms Fluid except for the document size.

2) Insert/remove N words (100 clients): Consider a scenario where a teacher is walking students through a notebook as an illustration during the lecture. In this case, one client is writing and a whole classroom (e.g. 100 students) is observing the notebook. Once again, Y.js seems to perform best.

3) Two clients simultaneously edit different cells of a notebook: Research [16] has shown that it is highly unlikely that two users concurrently edit the same part of a shared document. Consequently, a scenario is simulated where two users edit a different part of the document (in the case of notebooks this means two different cells). The results are more or less the same comparing to the previous two scenarios.

B. Usability

To obtain an idea of the ease-of-use of the two frameworks, a usability study has been conducted as part of this thesis. Participants were asked to create a simple web application, that uses RTC, in both frameworks. Afterwards they were presented with a questionnaire to evaluate different aspects related to the usability of these libraries.

Eight users from different fields participated in this study. The different study fields or professions of the participants are summarised in Table 6.4. Out of these eight participants, four of them failed to complete both assignments. The other four were all able to complete the assignments using Y.js and three of those four also successfully completed the assignment using Microsoft Fluid. Y.js receives an average documentation score of 3.57 out of 5 while Fluid receives a 2.43 out of 5. The participants that successfully completed the assignment using Y.js spent on average 1 hour and 20 minutes on the assignment. For Fluid, the average time spent is 3 hours and 22 minutes.

TABLE I
Field of study / profession of the study participants.

Study Field / Profession	# Participants
Computer Science Engineering	3
Software QA Team Lead	1
Software Developer	3
Software Engineer	1
Data Scientist	1

VII. Prototype

A. Introduction

This section describes the development of a proof-of-concept prototype that implements RTC functionality in Jupyter notebooks. The prototype is developed as an extension to JupyterLab, Jupyter's next-generation notebook interface. As this thesis stems forward from the perspective of RTC in an educational environment, JupyterHub is used as entry point for the user. JupyterHub provides authentication and the ability to provide users with a unique account that can for example be linked to their institutional account. Whenever a user logs in to the hub, a JupyterLab instance containing the developed RTC extension is spawned.

B. Architecture

Figure 2 shows the architecture of our prototype setup. Alongside JupyterHub an instance of the Y.js WebSocket server is running. JupyterHub itself is configured to spawn a JupyterLab instance instead of the regular notebook interface. Notebooks that are shared through this instance using the RTC extension are synchronised by means of the Y.js server. Figure 3 illustrates how the RTC extension interacts with the Jupyter notebook and the WebSocket server.

C. Operations

1) Sharing a notebook: To share a notebook, a user presses the share button. Consequently the shared structure is initialised and a connection to the WebSocket server is established. The shared structure consists of a JavaScript class that contains a Y.js document and handles all local and remote changes. The following data is kept in the shared document:

- *rtc-id*: A unique id that identifies the shared notebook.

[5]https://github.com/dmonad/crdt-benchmarks
[6]https://github.com/janmarien/crdt-benchmarks

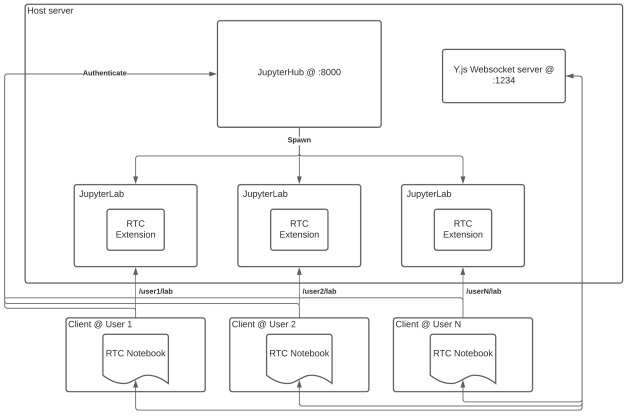


Fig. 2. Architecture of the prototype setup

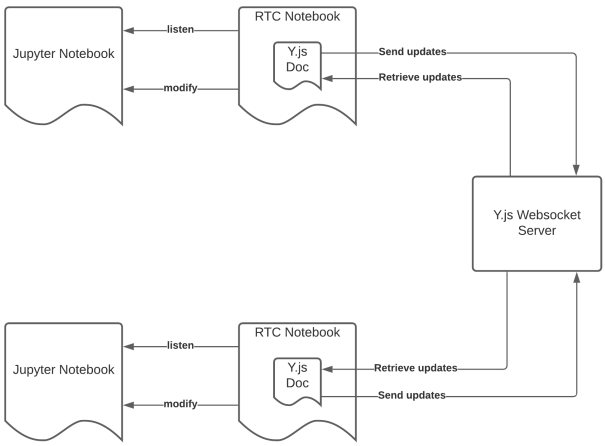


Fig. 3. The RTC notebook, which processes local and remote document updates, is kept alongside the original notebook and holds the shared document. Using this document, updates are distributed to all clients.

- *Shared text*: A text entry in the Y.js doc which holds the current contents of a cell and that can be accessed by the cell's rtc-id.
- *Shared cell metadata*: A map structure that stores additional data for every cell. The cell's type, output, execution count and position in the notebook are stored inside this map.

2) Connecting to a shared notebook: Upon connecting to a shared notebook through the provided rtc-id, a connection is established to the WebSocket server and the contents of the shared document are fetched. A new notebook on the user's instance is then populated using the shared content. Each cell is inserted according to its position in the metadata and the cell is consequently bound to the shared text. The shared structure then initialises remote and local listeners to handle changes.

3) Cell editing: Cells are bound using the Y.js CodeMirror binding[7]. This ensures automatic synchronisation of the cell's content across all clients.

4) Adding and removing cells: When adding/removing a cell, the local listener adds/removes an entry from the metadata map. This change is synchronised and all other clients receive an update indicating a change to the metadata. The remote listener at these clients handles this change and act accordingly. In case of an addition a new cell is created and bound to its accompanying shared text. Upon deletion, the remote listener removes the cell from the notebook and discards all bindings.

5) Moving cells: Moving a cell triggers an event indicating the old and new position of the cell. The local listener intercepts this event and consequently updates the cell's position in the metadata map. Again, remote clients receive this update and act accordingly by moving the respective cell to it's correct position.

6) Executing cells: The execution of cells is currently implemented such that only the client hosting the notebook is bound to a kernel and effectively executes a cell. Whenever a client that is not the host (the one that initiated the sharing of the notebook) executes a cell, the execution count value in the metadata is incremented. The host receives this update and executes the respective cell. Upon completion of this execution, the result is propagated to all clients by setting the output value for the executed cell in the metadata map. Remote clients receive this change and consequently update the output of the cell in their notebook.

7) Changing cell type: Changing the cell type (e.g. from code to markdown) is done by updating the *type* field in the metadata map.

8) Saving the notebook: Saving a notebook automatically triggers the serialisation of the shared Y.js document. The document is then stored in the notebook file as metadata. This serialised form can later be restored when reconnecting to a shared notebook.

D. Future Work

Several options regarding the notebook **kernel** are possible. Currently only one of the clients, the host, is connected to a kernel. This means that all executions take place in this user's notebook. However, other possibilities such as separate kernels with shared execution or just separate kernels with local execution exists and whether or not to synchronise execution state heavily depends on the use case.
Currently, no **security** measurements have been taken. This means that everyone that has access to the JupyterHub server

[7]https://github.com/yjs/y-codemirror

can access any document being shared on the server. Even worse, the Y.js WebSocket server needs to be available externally as well. A fine-grained permission system to secure document access is required for a production-ready extension. **Integration** of the Y.js WebSocket and other components into the Jupyter ecosystem could be beneficial for both security and the end-user experience. To provided complete RTC functionality to the user, support for **offline editing** should be implemented. This can be achieved by storing and restoring the Y.js shared document.

VIII. Conclusion

In this thesis, research is conducted into the possibilities of bringing Real-Time Collaboration to Jupyter notebooks. The literature study shows that RTC is an active and well researched domain and many solutions are available nowadays. As part of this thesis, a study that compares two RTC libraries, Y.js and Microsoft Fluid, is conducted. Based on both performance and usability aspects, it is concluded that Y.js currently is the best candidate to implement RTC in Jupyter notebooks. Although Microsoft Fluid is a decent framework and shows great potential, the current version is not deemed stable enough and continuous (breaking) changes make this library less fit for production-ready implementations. Based on this conclusion, a proof-of-concept extension for JupyterLab was build using Y.js. This prototype demonstrates the possibilities of RTC in Jupyter notebooks and proves that the current state-of-the-art technology on RTC is up for the task. Currently, the JupyterLab community is working on the development of fully integrated RTC functionality in their product through the use of the Y.js framework. After a long year of planning and several discussions in the community on how to implement RTC, it seems that the actual development recently kicked off and is making good progress. It is thus expected that RTC is coming to JupyterLab sooner rather than later.

The work and research on this specific topic seems complete. However, future research could focus on other related e-learning problems. For example tools which provide an easy way of deploying notebooks and providing students with a develop-ready environment without having to distribute all the required data manually (which can be a lot when working with big data) could really improve remote learning assignments.

References

[1] P. Parente, "Estimate of public jupyter notebooks on github," 2014.

[2] C. A. Ellis and S. J. Gibbs, "Concurrency control in groupware systems," *SIGMOD Rec.*, vol. 18, no. 2, p. 399–407, Jun. 1989. [Online]. Available: https://doi.org/10.1145/66926.66963

[3] C. Sun and C. Ellis, "Operational transformation in real-time group editors: Issues, algorithms, and achievements," in *Proceedings of the 1998 ACM Conference on Computer Supported Cooperative Work*, ser. CSCW '98. New York, NY, USA: Association for Computing Machinery, 1998, p. 59–68. [Online]. Available: https://doi.org/10.1145/289444.289469

[4] C. Sun, Y. Yang, Y. Zhang, and D. Chen, "A consistency model and supporting schemes for real-time cooperative editing systems," 1996.

[5] 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 ACM Symposium on User Interface and Software Technology*, ser. UIST '95. New York, NY, USA: Association for Computing Machinery, 1995, p. 111–120. [Online]. Available: https://doi.org/10.1145/215585.215706

[6] M. Shapiro and N. Preguiça, "Designing a commutative replicated data type," INRIA, Research Report RR-6320, 2007. [Online]. Available: https://hal.inria.fr/inria-00177693

[7] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-free Replicated Data Types," in *SSS 2011 - 13th International Symposium Stabilization, Safety, and Security of Distributed Systems*, ser. Lecture Notes in Computer Science, X. Défago, F. Petit, and V. Villain, Eds., vol. 6976. Grenoble, France: Springer, Oct. 2011, pp. 386–400. [Online]. Available: https://hal.inria.fr/hal-00932836

[8] N. Fraser, "Differential synchronization," in *DocEng'09, Proceedings of the 2009 ACM Symposium on Document Engineering*, 2 Penn Plaza, Suite 701, New York, New York 10121-0701, 2009, pp. 13–20. [Online]. Available: http://neil.fraser.name/writing/sync/eng047-fraser.pdf

[9] Micrososft. (2020) Fluid framework. [Online]. Available: https://fluidframework.com/

[10] Microsoft. (2020) Fluid framework - frequently asked questions. [Online]. Available: https://fluidframework.com/start/faq/

[11] ——. Microsoft fluid framework - architecture. [Online]. Available: https://fluidframework.com/docs/concepts/architecture/

[12] P. Nicolaescu, K. Jahns, M. Derntl, and R. Klamma, "Near real-time peer-to-peer shared editing on extensible data types," 11 2016, pp. 39–49.

[13] K. Jahns. (2020, Dec) Are crdts suitable for shared editing? [Online]. Available: https://blog.kevinjahns.de/are-crdts-suitable-for-shared-editing/

[14] M. Kleppmann and A. R. Beresford, "A conflict-free replicated json datatype," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 10, p. 2733–2746, Oct 2017. [Online]. Available: http://dx.doi.org/10.1109/TPDS.2017.2697382

[15] M. Kleppmann and A. R. Beresford, "Automerge: Real-time data sync between edge devices," 2018. [Online]. Available: https://www.mobiuk.org/abstract/S4-P5-Kleppmann-Automerge.pdf

[16] G. D'Angelo, A. Di Iorio, and S. Zacchiroli, "Spacetime characterization of real-time collaborative editing," *Proc. ACM Hum.-Comput. Interact.*, vol. 2, no. CSCW, Nov. 2018. [Online]. Available: https://doi.org/10.1145/3274310

Contents

List of Figures

List of Tables

List of Abbreviations

CmRDT ..... Commutative Replicated Data Type

CRDT ..... Conflict-Free Replicated Data Type

CvRDT ..... Convergent Replicated Data Type

DC ..... Definition Context

DDS ..... Distributed Data Structure

dOPT ..... Distributed Operational Transformation

EC ..... Execution Context

ET ..... Exclusion Transformations

GOT ..... General Operational Transformation

IDE ..... Integrated Development Environment

IT ..... Inclusive Transformations

JSON ..... JavaScript Object Notation

LUB ..... Least Upper Bound

OT ..... Operational Transformation

P2P ..... Peer-to-Peer

RTC ..... Real-Time Collaboration

SEC ..... Strong Eventual Consistency

Listings

Chapter 1

Introduction

Throughout the years live collaboration functionalities have become ubiquitous. Using examples such as Google Docs and Microsoft Office many people nowadays are utilising these features, knowingly and unknowingly.
Real-Time Collaboration (RTC) is not restricted to text documents or chat services but also different forms of coding are now the subject of this technology. Whereas Version Control System tools like Git[1] still provide an easy way to share code between different contributors, developers would like to see features such as RTC coming to their favourite tools and IDE's. Computational notebooks, narrative documents in which code and illustrative text are fluently interleaved, have become increasingly popular among developers, scientists and students. Therefore it is not a question if but when RTC functionality will come to these narrative coding tools as well.

Recently, online lectures and exercises have become more and more prevalent. Even more right now, when both students and teachers have to deal with an unforeseen way of teaching as a result of the COVID-19 pandemic, online classes and assignments are the new standard. This new way of educating, however, comes with many challenges. In data science courses, for example, students often receive a notebook in which several small assignments guide them towards learning a new tool or concept. When on campus, a student can easily ask for help directly from a tutor or assistant which then immediately have access to his or her code. Using distant-learning, direct access to code is less obvious and thus real-time feedback is not that straightforward. This is were RTC functionality in notebooks can really be a valuable addition. It would allow direct access to a students assignment and the ability to provide real-time feedback and guidelines.

Currently, computational notebooks have limited support for Real-Time Collaboration. Although many notebook products exist, only the commercial ones seem to be able to deliver such functionality. Open source alternatives such as Jupyter notebooks, which is widely used e.g. at universities for teaching data science, do not provide RTC features at this time. However having such capabilities will benefit both students and teachers when teaching remote courses or labs using computational notebooks. So what are the difficulties when sharing a notebook in real-time? What are the user's expectations when collaborating in live documents? How can an open source solution such as Jupyter implement RTC in their notebooks?

[1] https://git-scm.com/

In the first part of this thesis, a deeper dive into the specifics of computational notebooks and Real-Time Collaboration is made. The popularity of these narrative documents is illustrated and one of the most prevalent implementations, the Jupyter project[2], is thoroughly discussed. Many forms and use-cases for RTC exist and are zoomed in upon in a dedicated section followed by a brief discussion of possible issues and caveats when using collaborative features. Successively, the different sorts of algorithms that can be used are explained in detail. Many RTC examples and implementations already exist and to end this section, an overview of the state-of-the-art applications and libraries is given.

In a following section, this paper describes the study between two different RTC frameworks. In a comprehensive comparison of the Microsoft Fluid and the Y.js framework, both research into the performance as well as the usability is conducted. To measure the performance, an existing benchmark for RTC-libraries, that already includes Y.js, is adapted for the Microsoft Fluid Framework. Results of this benchmark are discussed and various aspects of both libraries are compared.
To evaluate the usability, a study was conducted in which participants were asked to create an example application in both frameworks. The order in which they did was randomised beforehand. Afterwards, every contributor was posed a set of questions related to ease-of-use, the time they spent on the assignment and the clarity of the documentation. These evaluations are then reviewed and compared for both frameworks.

As a final part of this study, a proof-of-concept application implementing RTC in Jupyter Notebooks is developed. The prototype is developed as an extension to JupyterLab using the Y.js framework for the underlying shared structure.

To complete this thesis, a final section summarises the results and discusses potential future work.

[2]https://jupyter.org/

Part I

Literature Study

Chapter 2

Computational notebooks

Computational notebooks can best be compared to a science lab report. They contain some data, calculations and figures but in the end the message of the scientist, e.g. the result and conclusion of an experiment, remains the most important. Through a combination of executable code and markdown, a markup language often used for documentation, visual narratives are formed that are much more easy to read than plain code. These notebooks mostly come in the form of an easily shareable digital document that can be edited through a web interface. Figure 2.1 displays a computational notebook which is accessed through the user's browser. The notebook consists of different cells which can all be executed separately or, in case of markdown cells, rendered as rich-text.

2.1 Popularity

The popularity of computational notebooks has significantly increased in the past decade. J. Singer [8] states three different causes as to why these notebooks are so prominent nowadays.

1. **Simplicity and accessibility** are two key characteristics that are favoured by many users. The structure one gets by interleaving different blocks of code with blocks of text makes such documents very comprehensible.

2. Secondly, the paper mentions the **modern-day coding approaches** as being one of the stimulants. Present-day developing is characterised by "a stack overflow mentality", according to Singer. Programmers nowadays very often scour the internet for code and combine different fragments to create new code. Due to its easily distributable format, notebooks are often published online and thus form a great source for such code snippets.

3. Lastly, the **attractiveness of these notebooks to many different types of developers** is overwhelming. In the article, notebook users are categorised into three main groups. First of all we have the more inexperienced developers or the so called novices. The simplicity of the notebooks together with its narrative style provide a comfortable manner to introduce these beginners to coding. A second group consists of the scientific developers. For them, these documents are more a tool to communicate their findings and thus pure code is less relevant. The narrative style and the

ease with which they can integrate descriptions, plots and graphics makes computational notebooks the favourite tool for scientific developers.

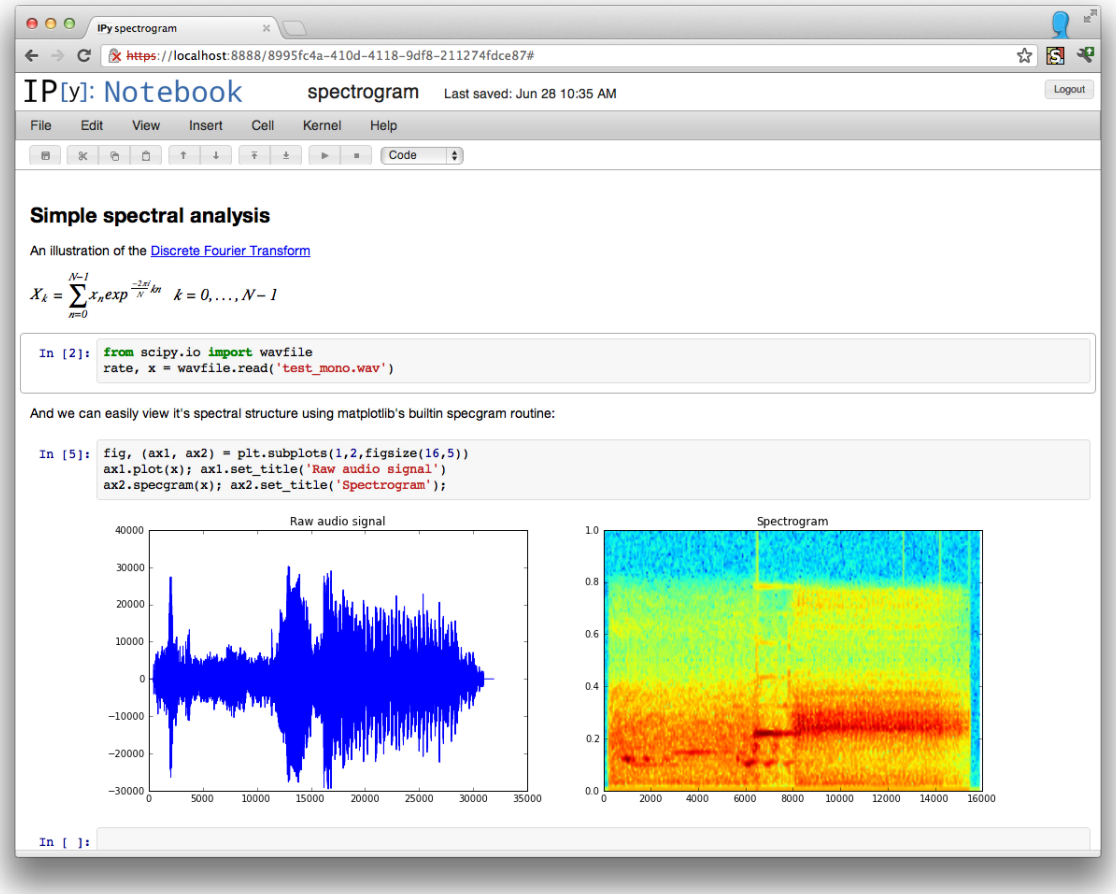


Figure 2.1: A computational notebook consists of multiple executable code cells interleaved with descriptive text [1].

To get an idea of how the popularity of computational notebooks has increased in the past decade, P. Parente [2] made an estimate on the amount of publicly available Jupyter notebooks, which is discussed later, on GitHub. Figure 2.2 shows that, currently, over nine million public notebooks have been published to GitHub. Coming from around sixty thousand hits in 2014, it is clear that the popularity of Jupyter notebooks has skyrocketed over the years.

2.2 Jupyter

Project Jupyter[1], which is based on the IPython project, is an open-source project that aims to provide software and services for interactive development. Currently, more than 40 programming languages are supported including the original languages, Julia, Python and R, to which this project thanks its name.

The grant proposal by F. Perez et al [9] explains how this project originated and which problems Jupyter tries to solve. Three important properties of computational narratives are mentioned:

[1] https://jupyter.org/

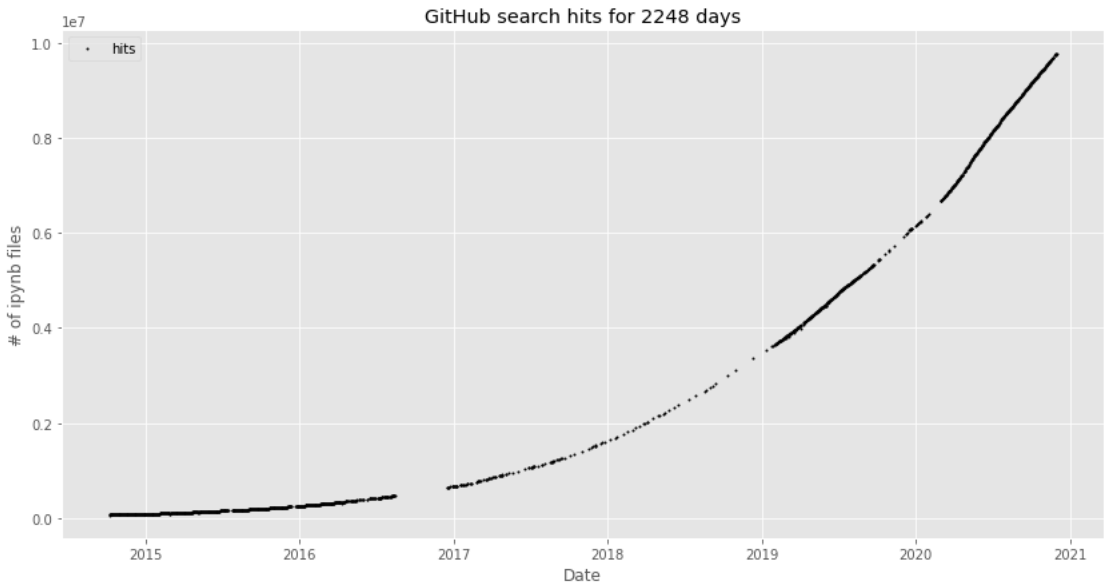


Figure 2.2: The number of public notebooks on Github has skyrocketed over the years [2].

1. **Variety of audience and contexts**
   A computational notebook that tells a certain narrative is likely to be spread across many different audiences, one more technical than the other. In some cases, even non-scientific or non technical people walk through or collaborate on such notebooks. It is important that, across all these contexts and audiences, the essence of the narrative is preserved.

2. **Reproducibility**
   When e.g. a certain scientist creates a computational narrative, it should be clear, for both himself and other people, how to understand this narrative and how it can be exactly reproduced.

3. **Collaboration**
   Collaboration is one of the key aspects of computational narratives. It is important that people can easily share and work together on this narrative. Real-time Collaboration plays a big role in achieving such a collaborative workflow.

The Jupyter project has several products of which **Jupyter Notebook** is definitely the most popular. Jupyter Notebook is a web application that provides an interface to create, edit and share computational notebooks. As mentioned in the previous section, its popularity has dramatically increased over the last years and the notebooks are being widely used by a diverse audience such as data scientists, students and researchers.

Along with their Notebook product, Jupyter also offers **JupyterHub**. JupyterHub aims to bring the notebook interface to multiple users in a way that is scalable and easy to manage, without the actual users having to struggle with any setup. It runs in the cloud and has support for multiple environments, authentication and provides several schemes for deployment. A typical scenario where the hub could be useful is a university which sets up a hub in the cloud. Hence, students and teachers have access to a notebook interface without having to set one up for their own.

While Jupyter Notebook only allows a user to edit notebooks and provides a rather simple interface,

JupyterLab offers a configurable and customisable front-end which supports multiple document formats and allows to open several notebooks and files in the same window. JupyterLab is said to be *Jupyter's Next-Generation Notebook Interface*. One of its great advantages is that JupyterLab is developed in a way such that many of the components are easily extendable. As a result, third-party developers can straightforwardly contribute to JupyterLab and distribute a variety of extensions.

2.3 Notebook structure

What follows is a description on how a computational notebook is structured. The description is based on Jupyter notebook but generally, other computational notebooks have a similar structure.

A computational notebook consists of different cells. Each cell consists of input that can either be code, such as Python, markdown or raw (just plain text). Code cells have an additional output which will hold the result of an execution. Whenever a cell is executed, the input of the cell is passed as code to the kernel. The kernel is a process that runs in the background and is specific to the programming language that is being used in the notebook. Currently, several languages are supported but the most commonly used in such notebooks is Python. Upon execution of a cell, the kernel receives the code and executes it in the background. When the kernel has finished executing, it will pass the result back to the notebook. The notebook then renders this result into the output area of the specific cell.

The cells of a notebook are commonly structured in a way that no cell i above a certain cell j depends on the result of cell j. This means that code only depends on the code that is in cells above and that a user should be able to run the whole notebook from top to bottom without having any dependency error. While these notebooks allow the user to place a cell anywhere he likes and there is no such thing as dependency checks, it is considered bad practice when a user for example puts a cells containing $a + b$ above the cells defining a and b.

Chapter 3

Real-time collaboration

3.1 What is real-time collaboration?

Real-Time Collaboration (RTC), is a broad term that mostly refers to the practice where two or more users simultaneously work together and/or communicate through the use of dedicated software. Examples vary from live messaging to real-time text editing software like Google Docs. In this thesis RTC is referred to as the ability to perform simultaneous coding and text editing.

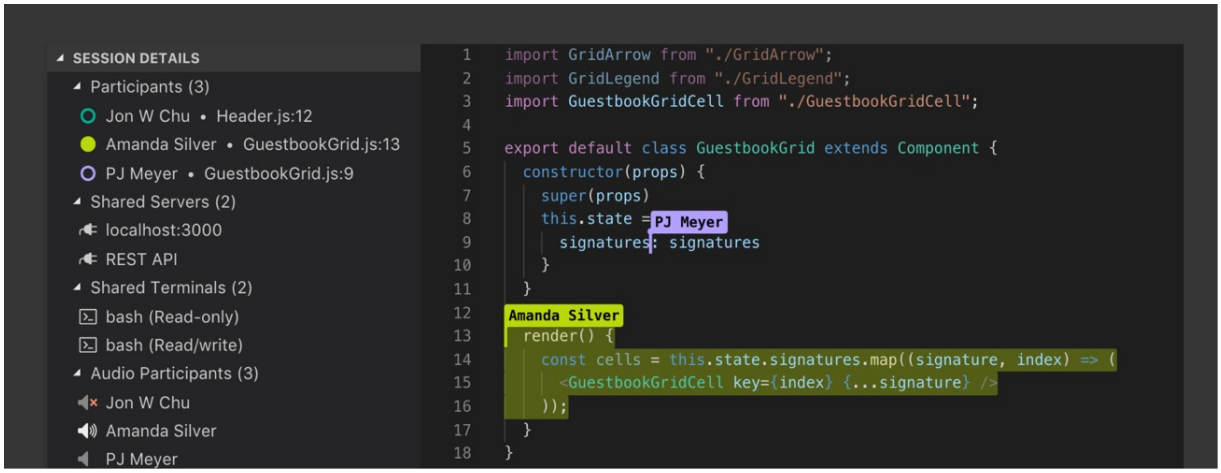


Figure 3.1: RTC in Visual Studio Code. In this image multiple users are editing the same document and their positions in the code are indicated along with their name tags [3].

RTC in coding environments such as computational notebooks is not as straightforward as for example in plain-text editors. Along with the textual information (e.g. the code or markdown), many additional aspects need to be considered when synchronising two documents. Figure 3.1 illustrates the use of RTC in a coding environment.

3.2 Challenges

When multiple users edit the same document, several challenges pose themselves which need to be taken into account when developing RTC functionality in an application. Problems such as documents being

out of sync or changes not being saved need to be avoided at all time. In addition to synchronisation, security of shared documents also needs to be considered. Which user can access another user's document or and can he change this document? The next chapter on algorithms goes deeper into the synchronisation challenges and translates some of these to requirements for these underlying algorithms.

3.3 Jupyter RTC

As mentioned in the grant proposal [9], support for collaborative features is highly requested by the Jupyter user base. At the time of publishment, Jupyter was focusing on bringing Real-Time Collaboration to their notebooks and joined forces with Google Research to do so. The main idea was to use Google Drive and Google Doc's architecture to bring RTC features to Jupyter. Unfortunately there is still no decent solution available. However, Google has proven with Google Colab that it is definitely possible to have RTC features in Jupyter notebooks. The question thus arises as to why the standalone Jupyter products still do not have any support for RTC despite earlier collaboration with Google.

Early 2020, JupyterLab started a working group[1] which is dedicated to bringing RTC functionality to JupyterLab. This group focuses on developing a data model to support RTC. Several design aspects and specifications are listed on their site and the original goal was to release RTC functionality together with JupyterLab 3.0 in June 2021[2]. With JupyterLab 3.0 released in January 2021, this goal was unfortunately not reached but work is still in progress. At the time of writing, the JupyterLab RTC community is developing a first working version based on the Y.js JavaScript library.

[1] https://github.com/jupyterlab/rtc

[2] https://jupyterlab-rtc.readthedocs.io/en/latest/organisation/czi-2020.html

Chapter 4

Algorithms

Software implementing Real-Time Collaborative features in order to maintain consistency across all clients, require a correct algorithm. Such algorithms highly relate to synchronisation algorithms used in a distributed system. Currently, two concepts, *Operational Transformation (OT)* and *Conflict-free Replicated Data Types (CRDT)*, are widely used in present-day collaborative software. In this section the requirements such algorithm needs to fulfil are discussed together with a thorough description of the earlier mentioned concepts. In addition to these two widely-used concepts, a third algorithm *Differential Synchronisation* is discussed.

4.1 Requirements

Applications with RTC features have to meet several expectations to provide a decent experience towards the user.

- Whenever another user, or the user himself in another client, is viewing or editing the same document, the user(s) expect(s) these documents to eventually become identical.
- The user(s) expect(s) that any change to the document is applied in the exact order as committed and that the result of this change is exactly as intended by this user(s).

These requirements can be formalised to the overarching concept called **consistency**, which is discussed in a following section.

4.2 Causality

4.2.1 Partial causal ordering

Before explaining the concept of consistency, a causal ordering relation is defined. This relationship is used to describe consistency requirements. In Lamport's *Time, Clocks, and the Ordering of Events in a Distributed System* [10] such ordering is thoroughly described. To be able to order certain events, a *happened before* relation (denoted as $\rightarrow$) is defined. More precisely, $a \rightarrow b$ means that event a happened before event b. This however does not necessarily mean that a caused b but one can be absolutely certain that b did not cause a. This relation is also called a potential causal ordering relation.

4.2.2 Logical Clocks

For different clients in a distributed system to be able to reason about this causality an additional data structure is added to messages passed between clients. To this end, Lamport introduces logical clocks. A logical clock simply adds a number to a certain event to be able to order this event in relation to other events. This means that if a relation $a \rightarrow b$ exists between event a and b that the clock value assigned to event a is strictly smaller than the clock value of event b, shortly notated as $a \rightarrow b \implies C(a) < C(b)$. This relation is called the clock consistency condition. However, this relation only holds in one direction. This means one cannot strictly order events based on their clock value and thus the potential causal ordering cannot be ensured. To do so, a more comprehensive data-structure, such as a vector clock is used.

4.2.3 Vector Clocks

The idea of vector clocks was formalised by Fidge [11], Mattern [12] and Schmuck [13]. However a first mention is found in a paper by Liskov et al. [14] where it was named *multipart timestamps*. Multipart timestamps are timestamps in which not one logical clock value is stored, but one for every client in a distributed system. A vector clock can thus be represented as $\vec{t_{p_i}} =< c_1, c_2, ..., c_n >$ for which a process p_i can only modify the value c_i. Vector $\vec{t_{p_i}}$ thus contains the event count for each p as seen from p_i. The following rules exist to compare vector clocks.

$$\vec{t_a} = \vec{t_b} \text{ if } \vec{t_{a_i}} = \vec{t_{b_i}} \; \forall i \tag{4.1}$$

$$\vec{t_a} <= \vec{t_b} \text{ if } \vec{t_{a_i}} <= \vec{t_{b_i}} \; \forall i \tag{4.2}$$

$$\vec{t_a} < \vec{t_b} \text{ if } \vec{t_a} <= \vec{t_b} \text{ and } \exists j \text{ such that } \vec{t_{a_j}} < \vec{t_{b_j}} \tag{4.3}$$

Equation 4.3 forms the base for defining the potential causal ordering. The following relation holds:

$$a \rightarrow b \Leftrightarrow \vec{t_a} < \vec{t_b} \tag{4.4}$$

4.2.4 Concurrency

Two events a and b are said to be concurrent (denoted as $a||b$) if both $a \not\rightarrow b$ and $b \not\rightarrow a$ hold.

4.3 Consistency

A variety of models to reach consistency exists. Below, a commonly used model for RTC editing systems by C. Sun et al [15] is discussed.
To reach a consistent system, three properties must be met.

- **Convergence**

  Whenever the system is at ease, meaning that all incoming and outgoing operations have been processed, all clients have an identical state (e.g. the content of a document is identical at all clients).

- **Causality-preservation**

  The causality-preservation implies that if $a \rightarrow b$, a should always be executed before b at all clients.

- **Intention-preservation**

  For two operations a and b, if $a||b$ holds then, regardless of the execution order of these operations, the intentions of both operations must be preserved.

4.4 Operational Transformation (OT)

4.4.1 History

Operational Transformation originated in 1989 as underlying algorithm for the GROVE (GRoup Outline Viewing Editor) system by Ellis et al [16] and was named dOPT (Distributed Operational Transformation). However, this algorithm suffers from some issues which resulted in the existence of different algorithm variants that try to solve these problems. The establishment of a Special Interest Group of Collaborative Editing (SIGCE) in 1998 led to a succeeding decade of intensive research and the expansion of OT. [5]

4.4.2 dOPT

The authors of GROVE describe the general functioning of their dOPT algorithm as follows. When a client performs a certain operation, it is executed immediately at this client and afterwards distributed to all other clients. Upon receipt of an operation from client b at client a, it examines the operation and determines whether or not client b has already applied operations which are not yet received by a. In this case, the operation will be queued. If not, it will be immediately applied at client b. Client a also checks whether it has already carried out operations which client b did not yet receive and may transform the operation if this is the case.

Transformation function

In order to achieve convergence, the algorithm requires a transformation function T which holds following property for two operations a, b and the resulting transformations $a' = T(a, b)$ and $b' = T(b, a)$:

$$a \circ b' \equiv b \circ a' \tag{4.5}$$

Equation 4.5 implies that when executing a before b, the resulting state is identical to the scenario where b gets executed before a. [16]

Consistency

dOPT is build upon the two following consistency properties explained in 4.3:

- Convergence
- Precedence (= causality-preservation)

Algorithm

Following data structures are used throughout the algorithm's description:

- Transformation function T (as defined above)

- State vector s (cfr. 4.2.3)
- Requests: a request $r =< i, s, o, p >$ contains the client's identifier (i), the client's state vector (s), the operation (o) and the priority for the included operation (p).
- Request Queue: A queue in which every client stores the requests which have not yet been executed.
- Request Log: A log containing every request that is executed at the client. Every client keeps such log.

The algorithm is described below from the viewpoint of a specific client identified by its identifier i.

During initialisation, the client creates an empty queue, an empty log and a state vector containing the value 0 for every client in the distributed system. When receiving an operation through the user's input, the client creates a new request and broadcasts this request to all other clients. Upon reception from a request that originated at another client, the client's stores this request in the request queue.

Whenever the request queue is non-empty, the client actively checks whether it is possible to execute requests. Three cases exist, depending on the value of the clients state vector s_i and the value of the request's state vector s_j.

1. The originating client has applied operations which have not yet been applied at the current client. $(s_j > s_i)$

   In this case, the request is ignored and left on the queue.

2. Both state vectors are equal. $(s_i = s_j)$

   The request is applied immediately and appended to the request log. State vector s_i is updated accordingly.

3. The current client has applied operations which have not yet been applied at the originating client. $(s_j < s_i)$

   The operation is transformed based on every request that has not yet been executed at client j. These requests are determined by scanning the request log at the current client. State vector s_i is incremented and the request is appended to the log.

Illustration

Figure 4.2 illustrates how convergence is achieved by *Operational Transformation*. Both clients initially are in an equal state consisting of the string *abc*. Both clients execute an operation at the same time: client 1 appends an x to the string ($ins[0, x]$) while client 2 removes c from the string ($del[2, 1]$). Both state vectors are equal when the clients create their operation and thus both operations are executed immediately. Upon reception of the operation at the other client, both clients end up in scenario 3 as described above. The operations are thus transformed depending on the previously executed operations at that client, resulting in operation $del[3, 1]$ at client a and operation $ins[0, x]$ at client b which remains the same as before. As end result, both clients end up with the string xab.

4.4.3 Problems

The dOPT puzzle

In their discussion on the algorithm's correctness, Ellis et al. mention a scenario for which dOPT ends up in an inconsistent state. This problem, known as the dOPT puzzle, is summarised in the paper by Sun et al [4]. Consider a scenario in which three clients perform actions on an initially empty document. The following rule for priority is used in case of concurrent insert operations that have the same position: the position of the operation with the lowest priority will be shifted. In this example, the priority equals the site's identifier. Client 3 performs an insertion [z, 1] noted as O_3. Client 2 executes an insertion [y, 1] noted as O_2. After arrival of O_3 at client 1, it inserts [x, 1] into the document. Note that $O_2||O_3$ and $O_2||O_1$. Upon arrival of O_2 at client 3, this operation is first transformed to $O_2' = ins[y, 2]$ and then to $O_2" = ins[y, 3]$. The document at client 3 now contains xzy. The same transformations occur at client 1 resulting in the same document content. At client 2, however, first an insertion of y is performed. Upon arrival of O_3 at this client, no changes are made to this operation due to the fact that O_3 has the higher priority. The document at client 2 now contains zy. O_1 now arrives at this client and is also transformed due to $O_1||O_2$. The result of this transformation is $O_1' = ins[x, 2]$ (O_1 has a lower priority than O_2). Due to $O_3 \rightarrow O_1$ no further transformations are required for O_1' and this operation is executed. The final result at client 2 is now zxy. This does not equal the document content at client 1 and 3 resulting in an inconsistent state of the system which is not desirable.

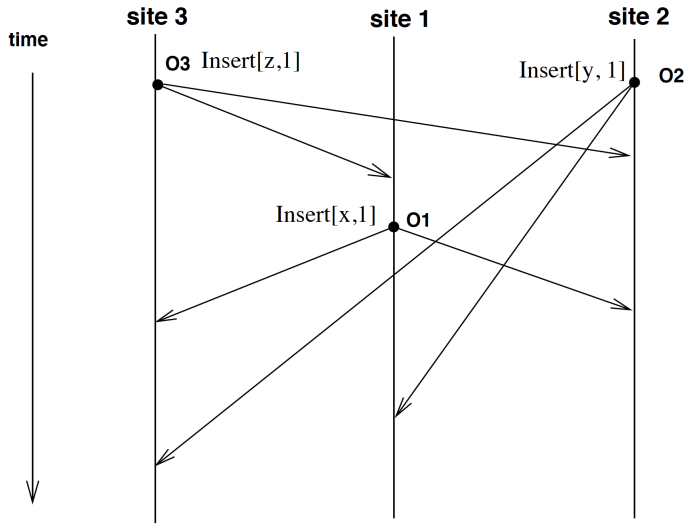


Figure 4.1: Scenario illustrating the dOPT puzzle. Three clients are performing operations for which the dOPT algorithm converges to an inconsistent state. [4]

4.4.4 Variants

In search for a solution to the dOPT puzzle, many adaptations of the original algorithm were created. Some of those are shortly summarised below with emphasis on their solution to the dOPT puzzle.

REDUCE

The REDUCE (REaltime Distributed Unconstrained Cooperative Editing) project [15] formed the basis for the intention-preservation property mentioned in 4.3. Along with convergence and causality-preservation, this resulted in the consistency model mentioned earlier in this paper. To achieve convergence and intention-preservation, REDUCE distinguishes two schemes. It uses *Operational Transformation* to reach convergence and an undo/do/redo scheme for the intention-preservation. REDUCE removes the responsibility of the transformation function to ensure convergence. The GROVE algorithm uses so called Inclusive Transformations (IT) which ensures that when an operation a is transformed against an operation b, the impact of operation b is included in the transformation. In addition to this type of transformation, REDUCE also uses Exclusion Transformations which exclude the impact of operation b from the transformation. For the algorithm to know when to use which type of transformation, the concept of *operation context* is introduced. For an operation O, two contexts exist:

- **Definition Context (DC):** context of the document state on which O is defined.
- **Execution Context (EC):** context of the document state on which O is to be executed.

In addition to these context definitions two relations are defined:

- **Context equivalence**: Two operations a and b are context equivalent, noted as $a \sqcup b$, if $DC(a) = DC(b)$.
- **Context precedence**: Two operations a and b are context preceding, noted as $a \mapsto b$, if $DC(b) = DC(a) + [a]$.

Based on these relations, a precondition is specified for both types of transformation functions. For the Inclusive Transformation $IT(a, b)$ it is required that $a \sqcup b$ while for the Exclusive Transformation $ET(a, b)$, $a \mapsto b$ must be satisfied.

To obtain intention-preservation both contexts must be equal: $DC(O) = EC(O)$. This is achieved by means of the *General Operational Transformation* (GOT) algorithm which uses both Inclusion- and Exclusion Transformations. The specific details are omitted in this thesis but below an illustration on how REDUCE solves the dOPT puzzle is given.

The scenario from figure 4.1 is repeated. Steps for client 2 and 3 remain the same as before. However, when O_3 arrives at client 1, that currently has a document state of y due to O_2, this previous operation is undone because of $O_3 \implies O_2$ (meaning O_2 totally follows O_3). Consequently O_3 is applied to the document and O_2 now has to be transformed using an Inclusion Transformation (as stated in the GOT algorithm). This results in an operation $O_2' = ins[y, 2]$ which forms the document state zy. Upon arrival of O_1, O_2" also has to be undone because of $O_1 \implies O_2'$. O_1 is applied without transformation ($O_1 \rightarrow O_3$) and O_2' is once again transformed using Inclusive Transformation. This results in O_2" $= ins[y, 3]$. The final result of the document at client 2 is xzy which is consistent with clients 1 and 3.

Jupiter

Different from dOPT, the Jupiter adaption [17] adds a centralised server to the system which keeps track of the shared documents. Communication is now restricted to a two-way client-server connection. An incoming operation on the server is transformed if necessary, applied on the server's copy of the document

and than broadcast to all other clients. If a client receives an operation from the server it transforms this operation if needed and executes it on its local copy. In addition to this, Jupiter also keeps track of valid transformation paths through at 2-dimensional graph. Due to communication being restricted between client and server this suffices to solve the dOPT puzzle.

adOPTED

In addition to the transformation property defined by formula (4.5), the adOPTED algorithm defines a second requirement which must be met by the transformation function.

$$T(T(o,a),b) \equiv T(T(o,b),a') \tag{4.6}$$

This newly added property ensures that whatever path is used to transform an operation o, the end result will be the same equivalent operation. By using an N-directional graph, the algorithm can keep track of all valid paths which provide a solution to the dOPT puzzle.

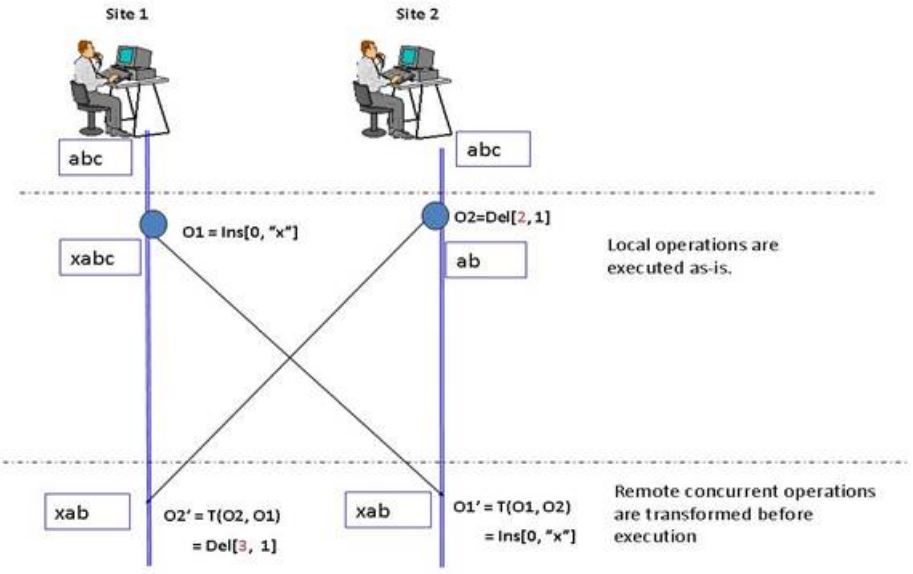


Figure 4.2: Two clients concurrently execute an operation. Operational Transformation ensures a consistent state (xab) at both clients. [5]

4.5 Conflict-free Replicated Data Type (CRDT)

The acronym CRDT was first mentioned by Shapiro et al. in 2007 [18] but was then defined as Commutative Replicated Data Type (CmRDT). Later in 2011, the concept of Conflict-free Replicated Data Type was formalised by the same authors [19] as a data structure for replicating data in a distributed system. Since then CRDTs have been the subject of research and many adaptations.

4.5.1 Model

Shapiro et al. distinguish two types of replicated objects: state-based and operation-based. Both models are used and an object based on the state representation is equivalent to an operation-based object

as proven by the authors. For certain scenarios, e.g. formal reasoning, states are preferred whereas implementations mostly follow the operation-based model. In both models a set of processes is considered. A process p of this set is deemed correct if this process has not crashed.

State-based model

Consider several replicas of an object $o_1, o_2, ..., o_n$ which are all in a state $s_i \in S$ also called the payload of this object. An object at process p_i is than defined as (s_i, s^0, q, u, m), with s^0 the initial state and q, u, m functions for respectively querying the object, updating the object and merging the state with the state of another replica. Following definitions and rules are defined:

- The states of two objects are identical if querying the objects returns an identical value and this for all possible queries.

$$s \equiv s' \text{ if } q_s(t) = q_{s'}(t)\ \forall t$$

- Querying an object does not affects it state.

$$(s \bullet q) \equiv s$$

- A method is defined as enabled when its precondition is met.
- Method execution is ordered numerically: $f_i^k(a)$ denotes the k^{th} execution of method $f \in (q, u, m)$ with arguments a at object replica i.
- A object is said to have state s^k after execution of $f^k(a)$ and state s^{k-1} before.

Operation-based model

The tuple (s_i, s^0, q, t, u, P), in which s_i, s^0 and q are identical to the state-based representation, defines an operation-based object. Instead of a merge function m, this model consists of a prepare-update method t and the effect-update method u. P denotes the protocol which will be used to distribute u to all other replicas. The same definitions as above apply for this representation as well, with an addition that the prepare-update method t leaves the object's state intact or more formally: $(s \bullet t) \equiv s$.

4.5.2 Causal History

The causal history for an object is defined below for both the state- as the operation based representation.

State-based

A causal history is defined as $C = [c_1, c_2, .., c_n]$ for which each c_i traverses states $c_i^0, ..., c_i^k, ...$ Following rules are defined for the k^{th} method execution at replica i:

- $c_i^0 = \emptyset$
- A query does not change the causal history ($c_i^k \equiv c_i^{k-1}$).
- An update $u_i^k(a)$ is appended to the causal history as follows: $c_i^k = c_i^{k-1} \cup \{u_i^k(a)\}$
- A merge $m_i^k(s_{i'}^{k'})$ with a remote state $s_{i'}^{k'}$ is added to C by taking the union of both replica's histories: $c_i^k = c_i^{k-1} \cup c_{i'}^{k'}$

Using this definition for causal history, a happened-before relation for two updates u and u' can be defined. An update u happened-before u' if at the moment of executing u' at a certain replica, u is included in the causal history of that replica ($u \rightarrow u'$ if $u \in c_j^{k-1}$). Following the definition from 4.2.4, two updates said to be concurrent if $u \not\rightarrow u' \wedge u' \not\rightarrow u$.

Operation-based

In the case of an operation-based object representation, both query and prepare-update (t) methods leave the history unaltered. Effect-updates are appended to the history in an identical way as updates in the state-based scenario. The happened-before relation can then be defined in a similar way: $(t, u) \rightarrow (t', u')$ if $u \in c_j^{k-1}$.

4.5.3 Strong Eventual Consistency (SEC)

Eventual Consistency intuitively means that, if no more updates are being conducted, a consistent state is eventually reached. It is defined by following properties:

- **Eventual delivery**
  If a correct replica delivers an update, this update should be delivered at all other correct replicas.
- **Convergence**
  Two replicas that have applied the same update(s) should eventually reach an identical state.
- **Termination**
  The execution of a method should always terminate.

To reach *Strong Eventual Consistency*, the convergence property is required to be more strict.

- **Strong Convergence**
  Two replicas that have applied the same update have an identical state.

4.5.4 CRDT

Monotonic semilattice object

Following concepts are introduced:

- Least Upper Bound

  The element m is a Least Upper Bound (LUB) for $\{x, y\}$, denoted as $m = x \sqcup y$, under the relation $\leq$ if and only if $\forall m', x \leq m' \wedge y \leq m' \implies x \leq m \wedge y \leq m \wedge m \leq m'$. LUB has the following properties:

  - Commutative: $x \sqcup y = y \sqcup x$
  - Idempotent: $x \sqcup x = x$
  - Associative: $(x \sqcup y) \sqcup z = x \sqcup (y \sqcup z)$

- Upper semilattice

  A join- or upper semilattice is a partial order $\leq$ that possesses a LUB m for all pairs $\{x, y\}$.

Based on above definitions a **monotonic semillatice** is defined as an object in state-based representation that possesses the partial order $\leq$, noted as $(S, \leq, s^0, q, u, m)$, and following properties:

- S is a semilattice ordered by $\leq$.
- Applying the merge function m to a remote state s' is equal to calculating the LUB between the local and remote state, $s \bullet m(s') = s \sqcup s'$.
- A state is monotonically non-decreasing across different updates, $s \leq s \bullet u$.

Convergent Replicated Data Type (CvRDT)

Under the assumption of eventual delivery and termination as stated in 4.5.3, a state-based object that assures the monotonic semilattice condition is SEC.

Commutativity

Two updates (t, u) and (t', u') are said to be commutative if and only if for a state s where u and u' are enabled, u (resp. u') is still enabled in state $s \bullet u'$ (resp. $s \bullet u$) and $s \bullet u \bullet u' \equiv s \bullet u' \bullet u$

Commutative Replicated Data Type (CmRDT)

An object in operation-based representation is said to be SEC if, under the assumption of causal delivery and termination, it assures the commutative condition for all concurrent updates and for which the precondition P is satisfied by causal delivery.

CvRDT and CmRDT

For both object representations, a data type that converges is now defined. Shapiro et al. furthermore prove that both CvRDT and CmRDT are equivalent. The proof will be omitted here as this is out of scope for this thesis.

4.5.5 Example: Integer Vectors

A vector of integers can be described as $v = (\mathbb{N}^n, [0, .., 0], \leq^n, [0, .., 0], value, inc, max^n)$ using the state-based object representation. Calling the *value* method returns the payload $s \in \mathbb{N}^n$. The *inc* function takes an argument i and increments the payload at index i, more precise $s \bullet inc(i) = [s'[0], .., s'[n-1]]$ for which $s'[j] = s[j] + 1$ if $i = j$ and $s'[j] = s[j]$ if $i \neq j$. The results of merging two vectors s and s' with each other is a vector containing the maximum value of both vectors for each index: $s \bullet max^n(s') = [max(s[0], s'[0]), .., max(s[n-1], s'[n-1])]$. If a client c_i is only allowed to increment the value at index i, a vector clock (4.2.3) is obtained.

4.6 Implementations

Several types of CmRDTs and CRDTs exist nowadays. A first Commutative Replicated Data Type (CmRDT) appeared in 2005, before CRDTs were formally defined, in a paper by Oster et al. and was given the name **WOOT** (WithOut Operational Transformation) [20]. Karayel et al. later proved [21]

that WOOT is indeed a CRDT.

In 2009, Preguiça et al. [22] introduced **TreeDoc**, a CmRDT designed for collaborative text editing. Treedoc considers a document as a linear succession of *atoms* which each have a unique position identifier (PID). Atoms can be seen as the smallest part which, combined with others, make a document. Examples of such atoms are a character or an image inserted in the document. The TreeDoc algorithm is based on a binary tree-structure which is build using the unique identifiers.

Logoot is defined by Weiss et al [23] as an algorithm for Peer-to-Peer (P2P) collaborative editing. It is an adaption of the original TreeDoc structure which uses lists of integers to order the PIDs instead of a tree structure.

Many other CRDTs and related algorithms exist. M. Klepmann et al provide a site [24] which contains interesting literature and material regarding CRDTs.

4.7 Differential Synchronisation

Differential Synchronisation is described in the paper by Fraser [25]. A clear summary of this paper can be found on the author's website[1]. Googles *diff-match-patch*[2], which formed the early basis for Google Docs, is a well known example implementation.

4.7.1 Basic Algorithm Overview

Figure 4.3 is used to describe the operation of the symmetric algorithm which repeatedly goes through the steps described below. Note that although one of the texts is called server text, both texts are located on the same client. It is assumed that the algorithm starts off in a state where client text, server text and common shadow are all equal.

1. The difference (diff) between the client text and common shadow is calculated (1a + 1b).
2. This so called diff will result in a sequence of edits that have been applied to the client text.
3. The client text which was used to calculate the diff is now copied over to the common shadow.
4. Edits as a result of the diff calculation are now used together with the server text to calculate a patch on best-effort basis.
5. This patch is now applied to the server text.

As this is a symmetric algorithm, it is now applied in the other direction so the client text gets updated with the changes made to the server text.

To be able to use this algorithm in a client-server environment the common shadow needs to be divided in a client shadow and server shadow which are updated separately. However, as patches are delivered on best-effort basis, a network error could cause client and server to go out of sync which can result in the loss of changes applied since the previous synchronisation. The following section describes an addition to the algorithm which avoids such data-loss.

[1] https://neil.fraser.name/writing/sync/
[2] https://github.com/google/diff-match-patch

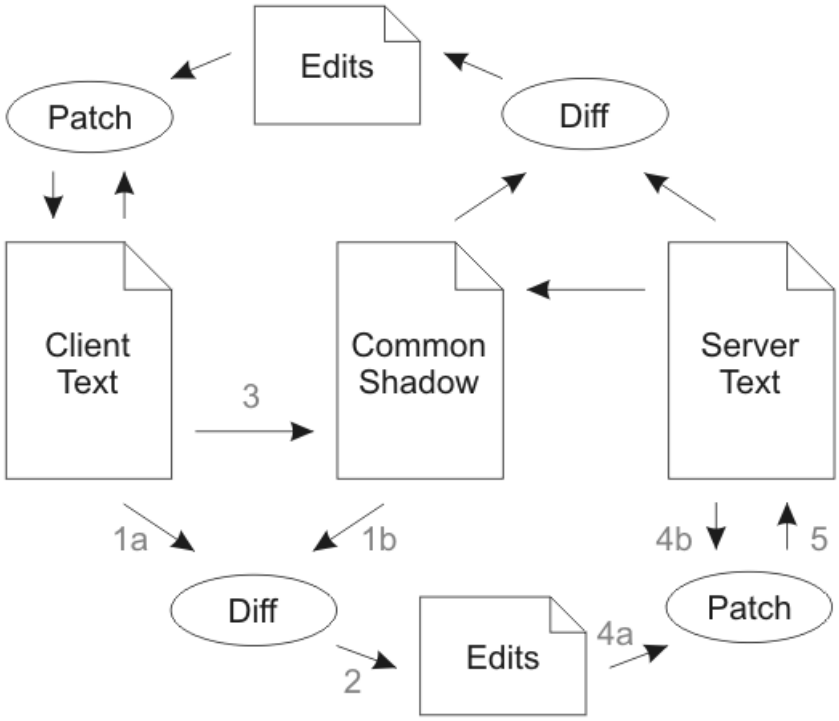


Figure 4.3: Illustration of the *Differential Synchronisation* algorithm. The algorithm repeatedly loops through steps 1 to 5 [6].

4.7.2 Guaranteed Delivery

In addition to the common shadow being split in a client- and server version, the server now has an extra backup shadow as shown in figure 4.4. When no network errors occur, the algorithm remains more or less the same. The client generates a diff, updates its version number and sends its previous version number n along with the edits and the version number of the previous server state m as an acknowledgement. Upon receiving the edits, the server verifies that both n and m match the current server shadow and then performs the same steps as the client in addition to creating a backup of the server shadow.

Following cases of network failures are considered:

- **Duplicate packet**
  The server will notice that it already applied edits with client version n and as there is no incentive for the server to do this twice, the double packet is simply ignored.

- **Lost outbound packet**
  If the server never receives a set of edits with client version n, the client will not receive an acknowledgement for these edits. In this case, the client will store the edits on a stack and will send a non-empty stack along with every future set of edits until the changes on this stack are acknowledged by the server.

- **Lost return packet**
  In case the server did receive the edits sent by the client but the server's acknowledgement got lost the client will remain in the same illusion as above. It thinks that his edits were not received correctly by the server. The server will thus receive the next edits together with the ones that

remain on the clients stack. This will cause a mismatch for the server version number m that is included in the edits send by the client. However, the server will detect a match between m and the version number of the backup shadow. The server then restores the backup shadow and ignores the first edit (as this has already been applied by the server) following a normal operation of the algorithm.

- **Out of order packet**
  One of the above scenarios of a lost packet will occur along with the duplicate scenario whenever the lost packet arrives.

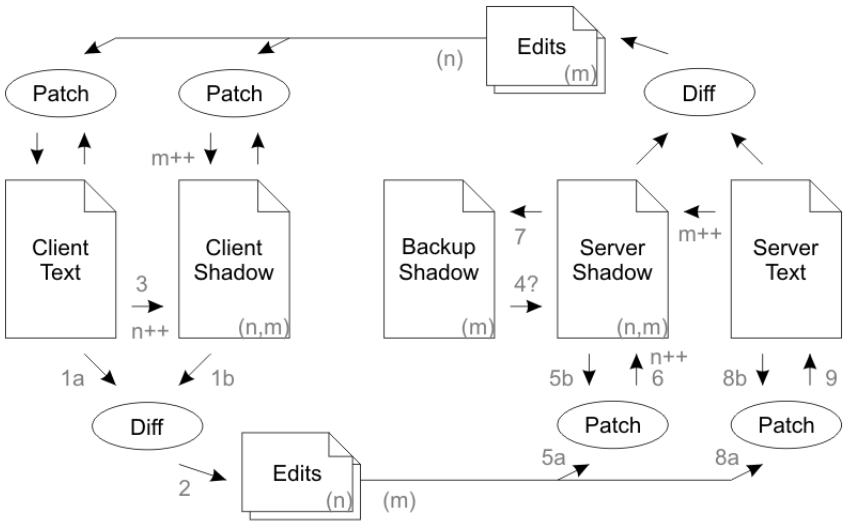


Figure 4.4: The *Differential Synchronisation* algorithm with guaranteed delivery. The server now has a backup shadow to be able to cope with network errors [6].

4.7.3 Issues

- **Scalability**
  The algorithm has quite expansive operations which will most likely cause performance issues in case of numerous clients.

- **No continuous stream**
  By design, a client or server needs to wait for a reply before sending a new packet. Consequently only one packet can be en route at the same time. This can cause performance issues on high-latency networks.

- **No metadata**
  The server merges all data from different clients together without any extra information such as which edits came from which client.

Chapter 5

State-of-the-art implementations

5.1 Libraries

5.1.1 Microsoft Fluid Framework

The Fluid Framework [26] is a set of libraries that allow a developer to build collaborative web applications. In September 2020, Microsoft announced that they made the framework available as open source [27]. Based on the code that is used for collaborative experiences in Office 365, Microsoft now offers these functionalities through a separate, publicly available framework.

While the original goal is to provide an easy way to build low-latency collaborative web applications, B. Posey [28] provides an extra argument why the public release of this framework might be worth following up on. The author of this article envisions the open source release of the Fluid framework as the beginning of something bigger. Microsoft's preview video [29] for this framework shows a future where components can be shared across all different Office applications just by simple dragging it from one app to another (e.g. a table made in excel included in an outlook email). This while retaining all real-time collaboration features for every component. Posey, however, thinks it might even go beyond this. If Microsoft continues on this momentum and makes future, related projects open source as well, we are one step closer to an environment where not only Office applications have fully shareable components but third-party applications also have the option to do so. Although this project is still in its early stages and not yet ready for production, it is clear that this framework has quite some potential and Microsoft sees a bright collaborative future.

Fluid offers RTC functionality through so called Distributed Data Structures (DDSs). Those DDSs are very similar to regular objects (e.g. an array) used when programming an application. However, these structures allow for editing by multiple users and automatically sync their state across all clients. Examples such as a *SharedString* or a *SharedMap* are only a small sample of the many data structures this framework has to offer. According to the documentation, Fluid uses neither an OP nor a CRDT implementation but their *model is more similar to CRDT than OT* [30].

Figure 5.1 shows the architecture[1] of the framework in which three main components can be distinguished.

[1] https://fluidframework.com/docs/concepts/architecture/

1. **Fluid Loader**

   The Fluid Loader is responsible for loading a Fluid Container. It does so by means of a URL. Through this URL, the loader looks up to which service the Container is bound. This service is then requested to prepare everything for setting up a new Container. Finally, the Loader fetches the required Container code which it uses to create a new Container.

2. **Fluid Container**

   A Fluid Container exists of at least one Fluid Object which contains one or more Distributed Data Structures. The main task of this container, which resides client side, is to implement the application logic and to perform operation merges. The DDSs receive operations from the Fluid Service and merge the changes to a consistent state.

3. **Fluid Service**

   One of the key goals of the framework is to make the server as lightweight as possible. The main purpose of the Fluid service is to order incoming changes (operations) and distribute them to all clients. Contrary to many OT-implementations, it does not process any of the operations and is not at all aware of the contents of a Container.

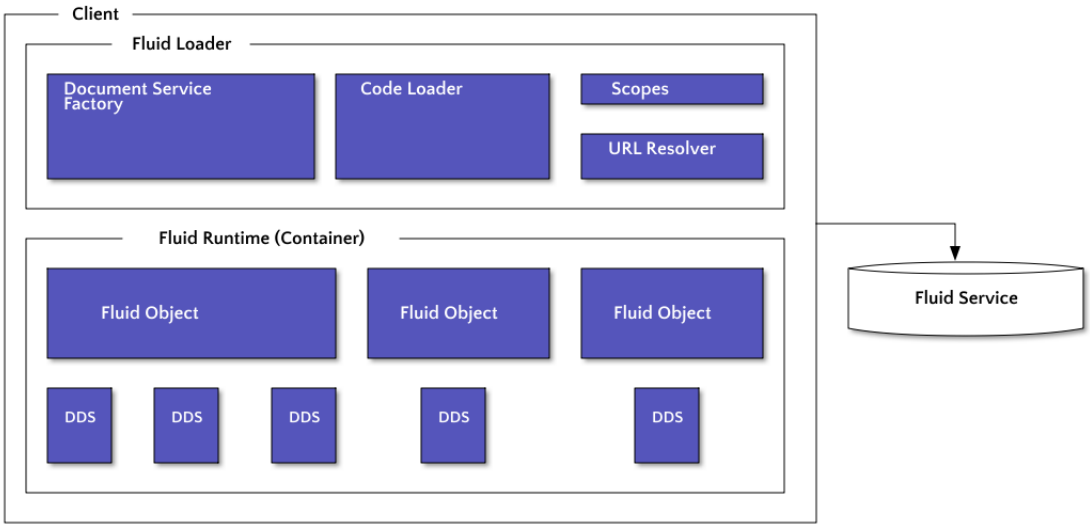


Figure 5.1: Architecture of the Microsoft Fluid Framework [7].

5.1.2 Y.js

Y.js[2] is a Javascript library that implements a CRDT to provide users with *shared types*. It supports several shared types such as text, map, array, and XMLElement. Contrary to the Fluid Framework, Y.js is Peer-to-Peer and does not rely on any server component.

[2]https://github.com/yjs/yjs

The CRDT implementation is highly based on the Yet Another Transformation Approach (YATA) mentioned in the paper by P. Nicoleascu et al. [31]. The algorithm's finest details are written down in this document but K. Jahns' blog post [32] greatly summarises everything you need to know about Y.js. The author of this framework starts by describing how data is represented in the CRDT algorithm. Y.js makes use of so called *Items* which form a linked list. Each Item contains the data (e.g. text), some metadata related to the algorithm and finally a unique ID. This ID is obtained by means of a *Lamport Timestamp* (4.2.2) which contains a unique client identifier and a logical clock to ensure the correct ordering of events.

In addition to the original algorithm, Y.js also implements the ability to merge and split Items. These operations preserve all metadata. As mentioned in the blog, both insert and delete operations have a maximum cost of two Item creations. Consequently, the size of all metadata is only proportional to the amount of operations conducted by the user and not at all to the number of characters inserted. The performance of this library is measured and compared to others in the benchmark developed by Jahns[3]. Based on these numbers, it is clear that Y.js outperforms the other libraries on almost all of the tested aspects. The benchmark itself will be more thoroughly described in a later section of this document.

Besides the library itself, the author also provides several bindings for existing text editors such as ProseMirror [33] and CodeMirror [34]. To distribute operations across multiple clients, various server implementations using protocols like WebRTC and WebSocket are available.

5.1.3 Automerge

Automerge[4] is a Javascript library that implements a CRDT based on the JSON-CRDT [35] by Klepmann et al. However the specific details differ from the original algorithm and are mentioned in a later paper by the same author [36]. Automerge can be used to build collaborative software by representing the state of the application as JSON objects. These objects can than easily be synced across different clients.

The library is originally developed to support software that is based on the *local first* principle by Klepmann et al.[37]. Local first tries to establish a new balance between data ownership and real-time collaborative applications by means of seven proposed ideals. The paper describes an interesting vision on how software should be developed from a perspective which prioritises the user's local storage before cloud storage.

5.1.4 delta-crdts

Delta-crdts[5] is a CRDT implementation based on the paper by P.S. Almeida et al [38]. Delta-crdts are based on so called delta-states which are much smaller in size than a full state. The idea of such delta-states is to send the result of certain updates as a state-fragment so that one can apply changes to remote states but without having to send the whole state, which can get quite big, to other clients. While this seems straightforward, ensuring causality is much simpler when merging a complete state but not when applying deltas. To this end the authors address the different challenges and propose the required concepts and conditions to ensure consistency. While delta-based CRDTs greatly improve the size of

[3] https://github.com/dmonad/crdt-benchmarks
[4] https://github.com/automerge/automerge
[5] https://github.com/peer-base/js-delta-crdts

messages that need to be distributed across clients, it do comes at a performance cost as shown by Jahns' CRDT benchmark[6].

5.1.5 ShareDB

ShareDB [39] is a JavaScript library which provides a real-time database that is convenient for usage with *Operational Transformation*. It was originally developed to store and sync JSON documents, but allows for other sorts of documents using shared types. While ShareDB provides the backend database, the shared types themselves are responsible for the actual OT implementation.

5.1.6 Google Wave Federation Protocol

The Google Wave Federation Protocol, which is build upon *Operational Transformation* [40], formed the base for Google Wave which was founded in 2009. Google Wave provides documents, called *waves*, which are stored on the server and can be edited in real-time. These waves had a lot resemblances with regular emails but add many extra features such as editing of spreadsheets and live chat. Instead of sending a new email whenever someone wanted to reply, a conversation could be made directly in the wave document. Although this is an excellent example of real-time collaboration and Google was one of the early companies to really use this technology to the fullest, Google Wave was unfortunately shut down in 2012 due to lack of interest. The federation protocol, which was made open source from the beginning, however continued to exist (in other Google products among others) and the project was further developed by the Apache Software Foundation before it was once more abandoned in 2018.

5.2 Existing software

5.2.1 Notebooks

Several solutions that bring RTC features to computational notebooks already exist. However for Jupyter notebooks, there is currently no decent solution available (anymore). **Callisto** is developed by Wang et al. [41] for their research into real-time collaboration in notebooks [42]. It is developed as an extension to Jupyter Notebook and is build upon ShareDB (5.1.5). While this seems to be a great extension and the kind of solution that is needed to enable RTC in notebooks, it is no longer usable. The client source code is available on GitHub[7] but unfortunately the server code has been removed and the extension can no longer be used.

CoCalc [43] is a commercial solution which aims at online course teaching and providing a collaborative environment. It offers a custom Jupyter Notebook implementation, support for LaTeX and many other tools such as chat rooms and a Linux terminal. As this is a commercial product there is no information available on how the RTC is implemented in their notebooks.

Released in 2015, **Apache Zeppelin** [44] is currently incubated as an Apache Software Foundation project. Zeppelin offers web-based notebooks and has support for several technologies such as SQL, Python and Apache Spark. Both single-user and multi-user solutions are provided and the notebook also offers Real-Time Collaboration features. It is however not mentioned which technology drives these

[6] https://github.com/dmonad/crdt-benchmarks
[7] https://github.com/littleaprilfool/callisto

collaborative functionalities.

Google Colab[8] is Google's wrapper around Jupyter Notebook. It provides a complete Jupyter environment in the cloud and offers access to Graphic Process Units (GPUs) and Tensor Processing Unit (TPUs). Due to the environment running on Google's infrastructure, it is robust, flexible and allows for easy sharing of notebooks (e.g. through Google Drive). Google Colab also offered Real-Time Collaboration functionality but as of November 2017, despite being one of the key features of this product, the API driving Colab's RTC has been deprecated and shut down by Google[9]. As of now there is still no replacement.

In an answer to Google Colab's RTC functionality being removed, **Deepnote** [45] was developed as an alternative solution. Similar to Colab it provides a environment for Jupyter Notebook which runs completely in the cloud. Unlike Colab, it has RTC capabilities and presents itself as the new tool for data scientists.

5.2.2 Text Editing

Text editors were one of the first applications in which RTC is brought to the end-user. Famous suites such as **Google Docs** and **Microsoft Office** all offer sharing capabilities and support the live co-authoring of documents. Many other editors exist which target themselves specifically as real-time collaborative editors. Examples are **Etherpad** [46] and **Firepad** [47].

Etherpad's RTC functionality is an OT implementation based on so called *changesets* [48]. In addition to collaborative editing, the software is easily extendable using various plug-ins such as real-time audio- and video chatting. Firepad is a collaborative text- and code-editor which uses Google's Firebase Realtime Database[10] as server component to store and synchronise all the data. The project is however currently no longer being maintained.

5.2.3 Coding

Along with text-editing, collaborative coding, often referred to as pair programming, has also become increasingly popular. While this is originally defined as two developers sitting behind one computer and programming code together, nowadays technology allow for collaborative coding in a remote setting. Several popular IDE's already support live sharing and coding amongst users.

Visual Studio Live Share [49] is added as feature for Microsoft's popular Visual Studio IDE and can be downloaded as an extension to its lightweight brother Visual Studio Code. It allows real-time collaboration in both IDE's with additional functionality such as voice chat and shared terminals.

Visual Studio Code's most famous competitor Atom also has support for live coding. **Atom Teletype** [50] is an extension to the Atom editor which allows users to collaborate in real-time. Teletype uses WebRTC to communicate in a P2P-fashion between the collaborating users.

[8] https://colab.research.google.com/
[9] https://developers.google.com/realtime/deprecation
[10] https://firebase.google.com/docs/database

JetBrains, known for its collection of IDE's such as IntelliJ and PyCharm, introduced **Code With Me** [51] in September 2020. Code With Me is a plugin that enables collaborative features such as pair programming in existing JetBrains products. It provides features such as audio- and video-chatting and a *following* functionality which can be very useful in classroom scenario.

CodeCollab [52] is an online code editor which supports more than 17 coding languages so far. It has support for sharing and real-time collaboration but lacks documentation on any other features.

Part II

Evaluation

Chapter 6

Comparison of Microsoft Fluid and Y.js

6.1 Introduction

This chapter describes a comparative study that is conducted as part of this thesis. Two libraries that can be used by developers to implement RTC functionality in web applications are compared. The libraries in question are Microsoft Fluid and Y.js. These frameworks are chosen based on several aspects. First of all, JavasScript is the preferred language as Jupyter's front-end is developed TypeScript which is a superset of JavaScript. Secondly, Microsoft Fluid has been made public only quite recently and thus makes an interesting candidate to evaluate as there are not many examples and information regarding performance available yet. Lastly, Y.js has already been compared to several other Javascript RTC libraries and came out convincingly as most efficient one. It thus makes sense that the newly Fluid is compared to the most performant state-of-the-art implementation. However, several side notes need to be made regarding this comparison. It is important to know that while Y.js is a P2P CRDT implementation which does not necessarily require a server component, Microsoft Fluid is not. Although, according to the authors, Fluid leans more towards CRDT than OT, it is neither pure CRDT and neither P2P as is the case with Y.js. It requires a server component and this should be taken into account when interpreting this evaluation.

6.2 Performance

6.2.1 Benchmark

For the performance study, an already existing CRDT benchmark by Kevin Jahns[1] is extended to acquire a notion of the performance of the Microsoft Fluid Framework. The benchmark compares several CRDT Javascript libraries and the code of this original benchmark is adapted to include Fluid. The benchmark consists of four different sets of tests in which synchronisation time, document size, (average) update size, parse time and an estimation of memory usage are all measured.

The benchmark is developed in Javascript and run using Node.js. Figure 6.1 shows how the different Fluid Containers that are used in the tests are connected to each other. Fluid's testing package (test-utils) is

[1] https://github.com/dmonad/crdt-benchmarks

used to create Containers. The Local Loader loads the container by connecting to the Local Service which in this case is a LocalDeltaConnectionServer. A second setup consists of replacing the local Fluid Service with the Tinylicious server, which comes much closer to a reference implementation of a Fluid Service.

Contrary to Fluid, Y.js does not necessarily require a server component. In the benchmark by Jahns, the operations on different documents are performed in a P2P-fashion by applying update patches from one document to the other document. In an RTC scenario one would obviously require a server to dispatch these updates to other clients. As mentioned before, although the local server or Tinylicious being run on the same host minimises latency, it should still be taken into consideration when comparing benchmark results.

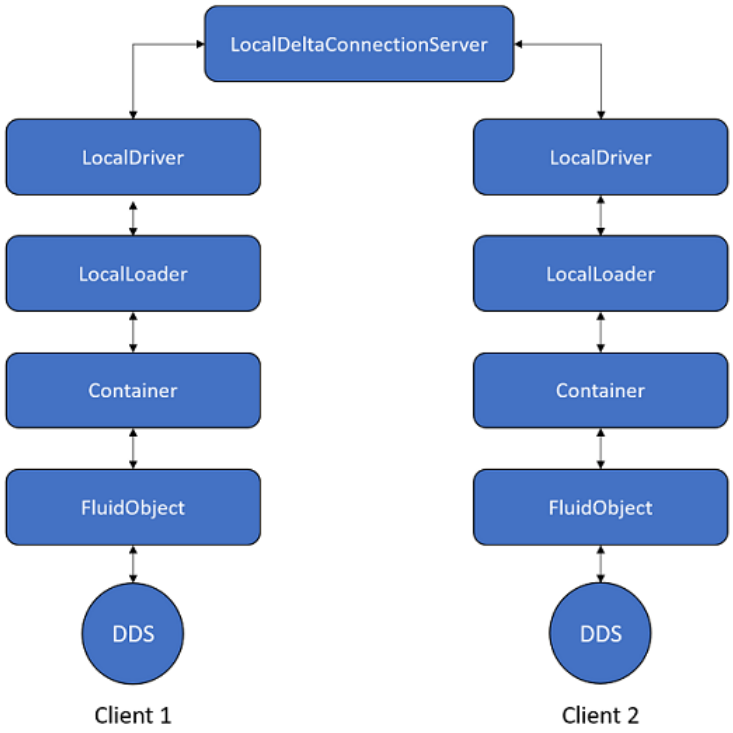


Figure 6.1: A local server, *LocalDeltaConnectionServer*, is used as Fluid Service to easily perform tests using multiple clients[2].

Synchronisation time

To measure synchronisation time, namely the time needed to have both clients in an equal state, we make use of the test-tools provided by the Fluid framework itself. These tools include a so called *OpProcessingController*, which can be used to await until all operations, both in- and outgoing, have been processed. This way, it can be exactly measured how long it takes for both objects to have the same content.

Document size

In Fluid, the Container comes the closest to the definition of a document. To have an idea of the size of the container, it is signalled to generate a summary using a built-in function. Upon the completion of

[2]https://github.com/microsoft/FluidFramework/tree/main/packages/test/test-utils

this summary, it is converted to a JSON string and the length of this string is considered the container size.

Update size

When a certain shared object is updated, the identical object at all other clients receives update operations from the Fluid service. A listener is attached to the shared object and each captured update is converted from JSON to a string. The lengths of all these strings is added up and forms the global update size for a single client. To calculate the average update size, the global update size is divided by the total number of received operations at a single client.

Memory usage

An approximate of the used memory is obtained by calculating the difference in heap-size before and after the benchmark. As mentioned by Jahns[3], this however can cause incorrect results when the heap is highly fragmented so these numbers should be interpreted with caution.

Parse time

Parse time is the time it takes for a new client to load a document and to fetch an up-to-date version. To measure this, a new Loader and Container are created and the shared object is obtained from the Container. The time it takes to complete this process is the parse time.

6.2.2 Description of the different benchmarks

Benchmark set 1

The first benchmark set consists of two groups: benchmarks that use textual input and ones that use numerical input. For the textual tests, Fluid's *SharedString* class is used contrary to the numerical tests for which the *SharedNumberSequence* class is used. The documentation of Fluid[4] on distributed data structures provides extensive information on both classes. The string-based benchmarks combine different ways of inserting a piece of text into the *SharedString* object. Numerical tests simulate the same operations by inserting numbers in to an array instead of using string. Different tests go from inserting the whole string at once to inserting characters one by one at random positions. Table 6.1 lists all the different benchmarks contained in this set.

Benchmark set 2

Benchmark set 2 covers scenarios where two clients modify a shared object at the same time. Table 6.2 describes all the different benchmarks contained in set 2.

Benchmark set 3

The third set treats cases where a high number of clients concurrently write to a shared object. These situations are the most demanding and require a lot of memory to simulate on a single system. Table 6.3

[3] https://github.com/dmonad/crdt-benchmarks
[4] https://fluidframework.com/apis/

Benchmark name	Description
B1.1 Append N characters	Insert all characters one after another into the SharedString
B1.2 Insert string of length N	Insert the whole string at once
B1.3 Prepend N characters	Insert all characters one by one at position 0 in the SharedString. The final result is the reversed string.
B1.4 Insert N characters at random positions	Insert all characters one by one at a random position in the SharedString
B1.5 Insert N words at random positions	Instead of inserting characters one by one, complete words are now inserted at random positions
B1.6 Insert string, then delete it	Insert a string of length N and remove it afterwards
B1.7 Insert or delete strings at random positions	Combine both insertion and deletion of words at random positions
B1.8 Append N numbers	Insert an array of N numbers at once into the sequence
B1.9 Insert Array of N numbers	Insert an array of N numbers at once into the sequence
B1.10 Prepend N numbers	Insert all numbers one by one at position 0 in the sequence
B1.11 Insert N numbers at random positions	Insert all numbers one by one at a random position in the sequence

Table 6.1: List of benchmarks contained in set 1.

Benchmark name	Description
B2.1 Concurrently insert string of length N at index 0	Two clients simultaneously insert a string of length N into the SharedString object.
B2.2 Concurrently insert N characters at random positions	Two clients simultaneously insert N characters one by one at a random position into the SharedString object.
B2.3 Concurrently insert N words at random positions	Two clients simultaneously insert N words one by one at a random position into the SharedString object.
B2.4 Concurrently insert & delete	Two clients simultaneously insert and delete N words at random positions in the SharedString object.

Table 6.2: List of benchmarks contained in set 2.

Benchmark name	Description
B3.1 250 *clients concurrently set number in map*	A high number of clients simultaneously set a value to the same key in a SharedMap.
B3.2 250 *clients concurrently set object in map*	A high number of clients simultaneously set an JSON object (e.g. {name: 'id', address: 'here'}) in a SharedMap.
B3.3 250 *clients concurrently set string in map*	A high number of clients simultaneously set a string of length $20\sqrt{N}$ in a SharedMap.
B3.4 250 *clients concurrently insert text in array*	A high number of clients simultaneously insert a character into a SharedObjectSequence

Table 6.3: List of benchmarks contained in set 3.

describes all the different benchmarks contained in set 3.

Benchmark set 4

The fourth and last benchmark comes much closer to real-world scenarios. It consists of replaying an editing trace that includes 182,315 character insertions and 77,463 deletions. Although this is a realistic scenario for someone writing a large document such as a paper or master thesis, this case is not that representative for someone who is coding in a computational notebook. However, for the sake of completeness and to get an indication on how Fluid performs in this scenario, the benchmark is added as well. The results can be found in the bottom of Table A1.

6.2.3 Test setup

All benchmarks are performed on a single device. Using a Node.js script clients are simulated by creating a container (Fluid) or shared document (Y.js) for each of them. This means that all these documents or containers, run on the same device. This is of course a simplification of a real-world situation and should be considered when interpreting results. Factors such as network delays or disruptions are not considered in this benchmark as Y.js is tested in a P2P-fashion and Fluid using a local Fluid service. The device on which the benchmark is run has the following specifications:

- 12 core AMD Ryzen 3900X @ 3.8 GHz
- 32GB 3200 MHz DDR4 RAM
- Node version 14.16

For the extra benchmark cases described in section 6.2.4, the Node.js *worker_threads*[5] package was used. To make the simulation somewhat more realistic in these tests, every connecting client is spawned using a separate worker.

6.2.4 Results

Table A1 included in the appendix contains the results of all benchmarks for Y.js and Microsoft Fluid using both the LocalServer and Tinylicious as Fluid Service. In general it is concluded that Y.js outperforms Fluid on most aspects. Synchronisation time is significantly faster and both average update size

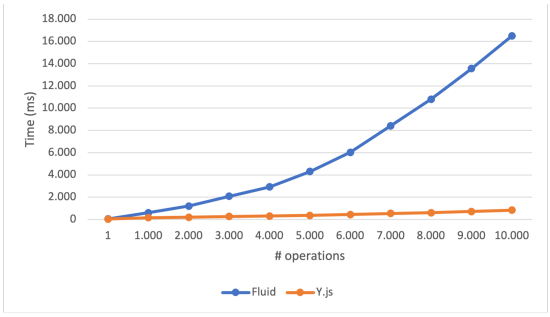
[5] https://nodejs.org/api/worker_threads.html

and memory usage are drastically lower. However when comparing document sizes, Microsoft Fluid has a compacter representation although some remarks (see 6.2.5) need to be taken into consideration.

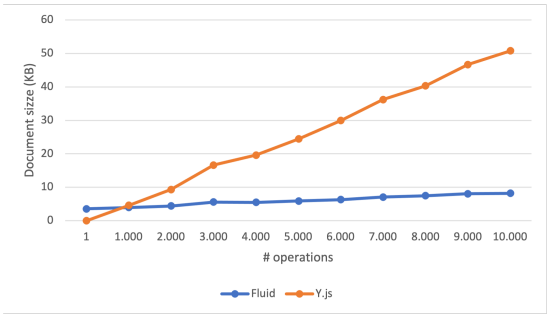
Below, some extra tests are discussed in the context of using RTC in computational notebooks for teaching purposes. Note that the memory usage here represents the average memory usage on a single client. In these benchmarks, Tinylicious is used as Fluid Service.

Insert/remove N words (2 clients)

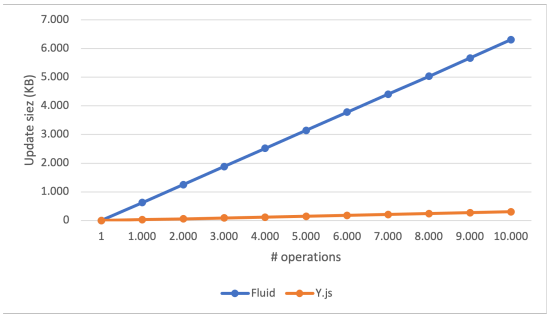
One of the most common scenarios is the case where a teacher is guiding a student through means of example. In this case, two clients are connected to the same document but only one of them is writing to it. This client performs N operations which either insert or delete a word between 3 and 10 characters. Figure 6.2 displays the benchmark results for this specific scenario. The results show that Y.js clearly outperforms Fluid except for the document size.



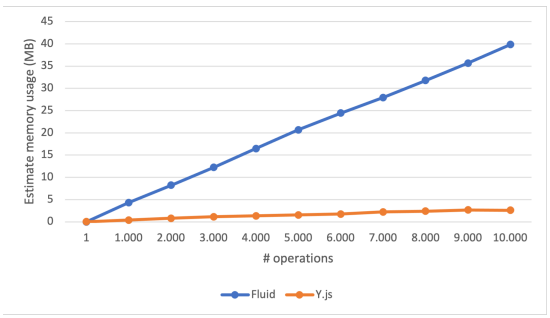
(a) Microsoft Fluid has a higher synchronisation time.



(b) The document size of Y.js is slightly bigger than Fluid.



(c) The update size of Fluid operations is significantly bigger than Y.js.

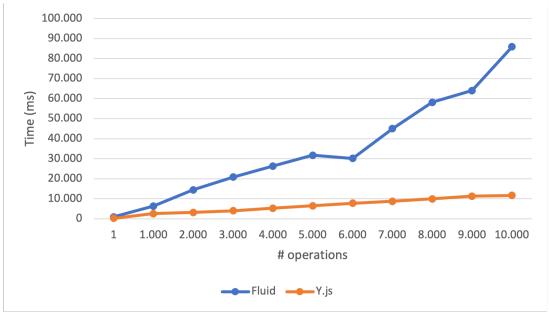


(d) Y.js has a much smaller estimated memory usage.

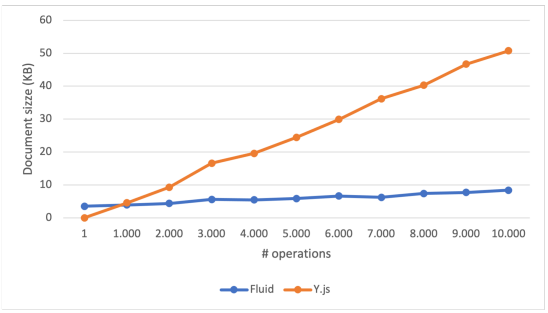
Figure 6.2: Insertions and deletions on 1 client with 1 additional read-only client.

Insert/remove N words (101 clients)

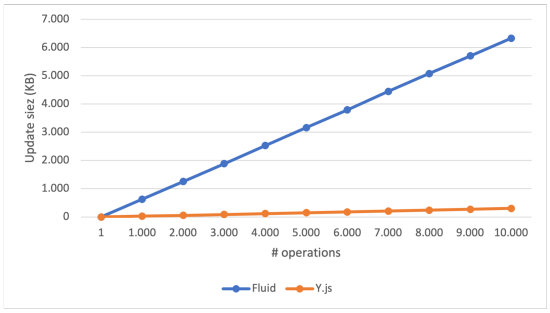
The previous scenario is repeated, but now 101 clients are connected (e.g. 1 teacher and 100 students). Again, only the teacher is conducting operations on the shared document (e.g. performing a demonstration for all students). As can be seen in Figure 6.3 Y.js once again outperforms Fluid.



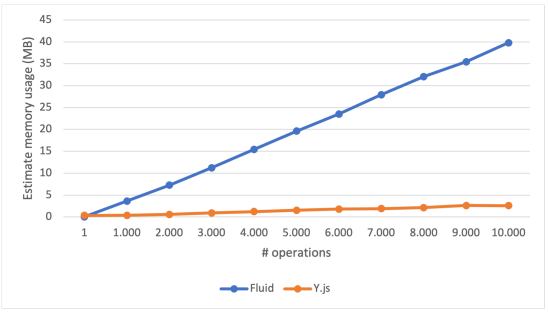
(a) Fluid has a significantly higher synchronisation time.



(b) The document size of Y.js is slightly bigger than Fluid



(c) The update size of Fluid operations is significantly bigger than Y.js



(d) Y.js has a much smaller memory usage than Fluid.

Figure 6.3: Insertions and deletions on 1 client with 100 additional 'reading' clients

Two clients simultaneously edit different cells of a notebook

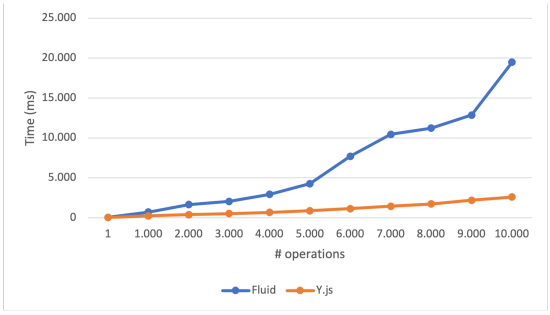
Consider two students working on the same assignment. Previous research [53] has shown that it is very unlikely that two users will concurrently edit the same part of a document. In collaborative notebooks, several collaboration strategies are used [42] but in all these strategies the users will try to avoid concurrent writing in the same cell. Figure 6.4 displays the results. These results are similar to the results from the first scenario. Y.js performs significantly better on all aspects except the document size.

6.2.5 Notes and issues

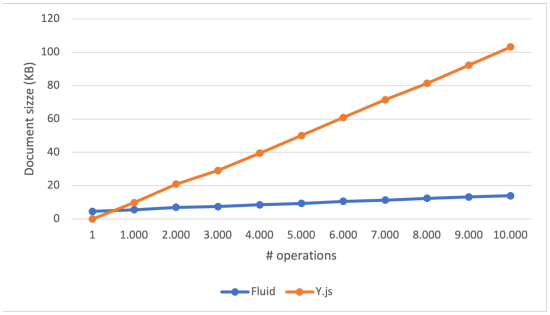
Real-world editing trace

In benchmark B4, all the operations (ops) of the editing trace are inserted into the document instantaneously. This is however not at all feasible for a real-world scenario as a human being cannot perform such an amount of operations that fast. This was also noticed when running the benchmark using the Fluid framework.

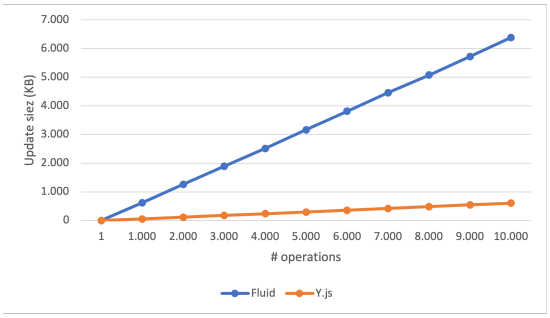
When performing this benchmark using Microsoft Fluid, it was still running after more than 4 days. While it is expected, based on the results of the other benchmarks, that Fluid requires more time than Y.js to process this editing trace, 4 days to synchronise around 250.000 operations is far outside acceptable boundaries. Taking into account that the Fluid Framework is still in its infancy, it is most likely a bug in the Fluid framework that causes some kind of deadlock when applying that many operations at once.



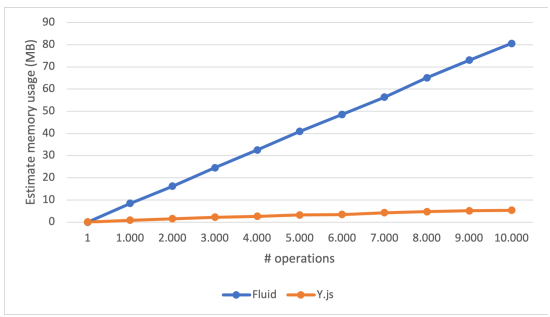
(a) Microsoft Fluid has a higher synchronisation time.



(b) The document size of Y.js is slightly bigger than Fluid.



(c) The update size of Fluid operations is significantly bigger than Y.js.



(d) Y.js has a much smaller estimated memory usage.

Figure 6.4: Two Clients simultaneously edit different parts of a document

Although applying all operations at once is not feasible at this time, an estimated result is obtained by allowing the framework to process the operations at a regular interval. Instead of performing all ops immediately, the editing trace was applied per 1.000 operations. Every thousand operations, the Fluid provider is allowed to first process all the previous 1.000 operations before it was served the following batch. This way, the deadlock was avoided and the result in the bottom of table A1 was obtained.

While this is not exactly the same as inserting all operations instantaneously, it is however considered to be much more representative for a real-world scenario in which a human being will never perform that many operations all at once without taking a break or at least pausing the necessary time for the underlying framework to process the operations.

Fluid Container size

The container size is based on so called summaries of the container. Fluid automatically summarises a container when a certain amount of ops has been processed or when the operation stream is idle for some time. A summary combines previous operations and removes ops that are not relevant anymore for the current state. This way, newly connecting clients can rapidly fetch an up-to-date version of the container without having to process all previous operations. Note that it takes an extra amount of time for a container to summarise its current state. The synchronisation time in previous results did not take this extra time into account. This time only indicates the time it takes for all clients to be in a consistent state, regardless of whether or not the clients' containers have already summarised the operations. Another important aspect is that Fluid only offers document storage using storage solutions such as Microsoft SharePoint and OneDrive. In contrast to Y.js, serialising a document and storing it just about anywhere

is thus not possible.

6.3 Usability

6.3.1 Study

As part of this master thesis a usability study has been conducted to obtain an initial idea on the ease-of-use of both frameworks. Participants of the study were asked to recreate a simple web application that makes use of real-time collaborative functionality. They were asked to do this once for each of the two tested frameworks. To provide some variety, two different example applications were created but each participant was asked to recreate only one of them. Afterwards, they were presented with a questionnaire to evaluate different aspects related to the usability of the particular framework. Hereafter follows a description of the example applications.

Chat example

In this example, the participant is asked to create a basic chat room application. The web page consists of three text boxes. One non-editable box that displays the chat and two editable fields to enter a username and a message. Finally a button is added to be able to send the message. Whenever a user sends a new message, all other clients should see the message appear in real-time. To achieve this, the participants were asked to make use of the RTC-functionalities provided by the given framework.

Matrix example

The second example consists of a web application that displays an editable 3x3 matrix and its determinant. For simplicity, the editable matrix is achieved by creating a 3 by 3 table in which each cell contains a HTML *input* element. Whenever one of the values in the matrix is edited, these changes should be visible to all other clients in real-time. Secondly, the determinant value should also immediately change accordingly. To achieve this the participants must use the functionalities provided by the given library.

6.3.2 Results

Eight users from different fields participated in this study. The different study fields or professions of the participants are summarised in Table 6.4. Out of these eight participants, four failed to complete both assignments. The other four were all able to complete the assignments using Y.js and three of those four also successfully completed the assignment using Microsoft Fluid. Y.js receives an average documentation score of 3.57 out of 5 while Fluid receives a 2.43 out of 5. The participants that successfully completed the assignment using Y.js spent on average 1 hour and 20 minutes on the assignment. For Fluid, the average time spent is 3 hours and 22 minutes.

Study Field / Profession	# Participants
Computer Science Engineering	3
Software QA Team Lead	1
Software Developer	3
Software Engineer	1
Data Scientist	1

Table 6.4: Participants

Part III

Prototype

Chapter 7

Introduction

This part describes the development of a prototype as a proof-of-concept application for RTC in Jupyter notebooks. Based upon the results from the evaluation part, the prototype is created using the Y.js framework.

7.1 JupyterLab Extensions

JupyterLab is developed in a modular way such that it consists of a compact core application which can easily be expanded with additional extensions. An extensions is a collection of plugins that belong together. JupyterLab supports several types of plugins such as Application plugins, Mime renderer plugins and Theme plugins. Mime renderer and theme plugins are not used in the development of the prototype and are not further discussed. A detailed description can be found in the JupyterLab documentation[1].

7.1.1 Application plugin

Listing 7.1 shows the elements which are used to construct an application plugin. Every plugin has a unique identifier which is normally the package name of the extensions together with a unique plugin name. The option *autostart* indicates whether the plugin is automatically activated or activated upon request by another plugin. A plugin can require services from other plugins through the *require* field. For every service required, its corresponding token is provided. A token is used to identify a certain service. The *optional* argument is again a list of service tokens, the difference being that these services are not necessarily required for the plugin but will be used when available. For example, the user's JupyterLab instance can be customised and not have the Launcher plugin activated. If in that case, the ILauncher would be passed in the *requires* argument, the plugin would fail to launch because that service is not available. A plugin can provide one or more services to other plugins and indicate this using the *provides* field by passing the service's corresponding token. The final argument is the *activation* function. This is where all of the plugins functionality resides. The arguments passed to this function are respectively the JupyterLab front-end application, the required services and lastly the optional services.

[1] https://jupyterlab.readthedocs.io/en/stable/extension/extension_dev.html

```
const plugin: JupyterFrontEndPlugin<MyToken> = {
  id: 'my-extension:plugin',
  autoStart: true,
  requires: [INotebookTracker],
  optional: [ILauncher],
  provides: MyToken,
  activate: (app: JupyterFrontEnd, tracker: INotebookTracker, launcher: ILauncher)
      => {...}
};
```

Listing 7.1: Javascript object representing an JupyterLab application plugin

7.2 JupyterHub

In the prototype setup, JupyterHub is used as entry point for the user. From the perspective of an educational use-case, this is the most obvious choice. JupyterHub provides authentication and the ability to provide user accounts to every student, researcher and teacher. These accounts can for example be linked to the user's institutional account. Whenever a user accesses JupyterHub a JupyterLab instance containing the RTC extension is spawned.

7.3 Code

The code for this prototype is available on GitHub[2].

[2]https://github.com/janmarien/jupyterlab-rtc-yjs

Chapter 8

Architecture

8.1 Server

Figure 8.1 displays the server architecture for the prototype. The server is running a regular JupyterHub instance at port 8000 along with the Y.js WebSocket server at port 1234. The JupyterHub instance is configured to spawn a JupyterLab instance for every user instead of the regular Jupyter Notebook interface. Upon authentication, the user is redirected to his JupyterLab instance. This JupyterLab instance contains the RTC extension that is developed as part of this thesis. Notebooks that are shared through this JupyterLab instance connect to the Y.js server to distribute and receive updates from other clients that are working on the same document. Apart from the Y.js WebSocket server, no other processes or extra configuration is required to be able to use the RTC extension.

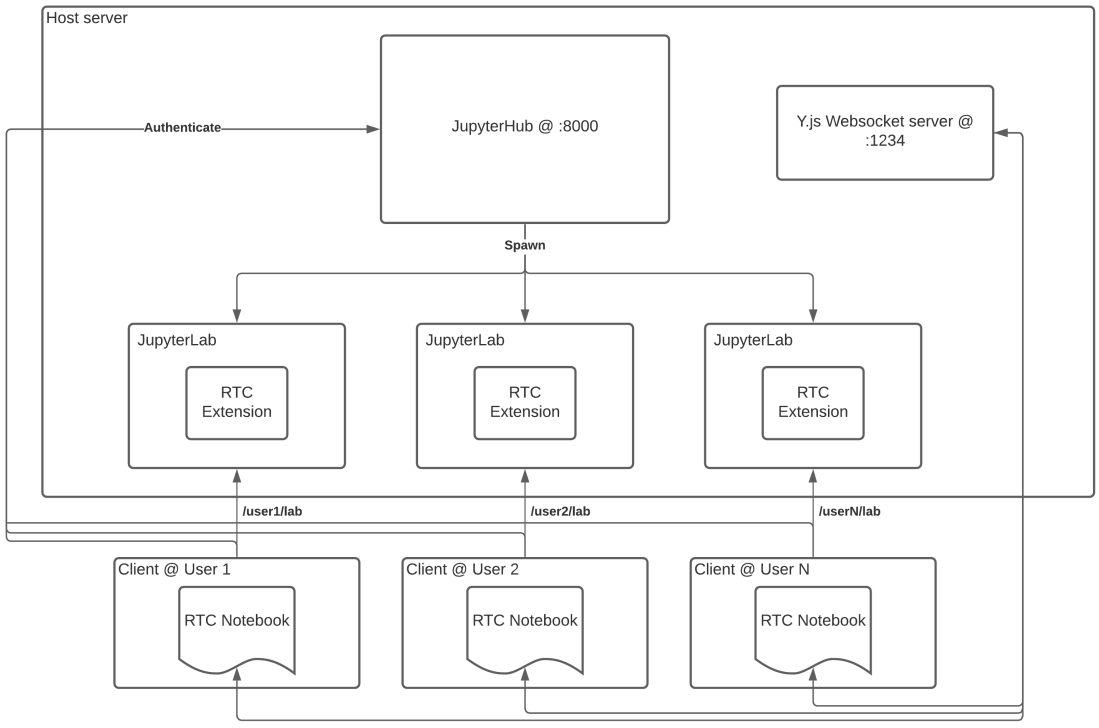


Figure 8.1: Architecture of the prototype setup

8.2 RTC Notebook

To synchronise our notebook with other clients, a shared data structure is required. For this structure, the Y.js CRDT framework is used. A notebook is represented by a Y.js shared document which supports several shared types such as text, arrays and maps. This document is kept in a separate structure called *RTCNotebook* which runs alongside the Jupyter notebook and handles all operations regarding synchronisation. Figure 8.2 shows a high-level representation on how these different structures interact with each other and how data is synchronised between two different clients. The Y.js doc is stored inside the *RTCNotebook* class which is bound to the loaded Jupyter notebook. The RTC structure listens to any changes or operations in the notebook and modifies the shared document accordingly. This Y.js document is connected to the Y.js WebSocket server along with all documents from other clients which are currently connected to the notebook. All those documents are synchronised by the WebSocket server and continuously send and receive updates. When the shared document receives an update, the RTC structure handles this update and reflects these changes onto the Jupyter notebook.

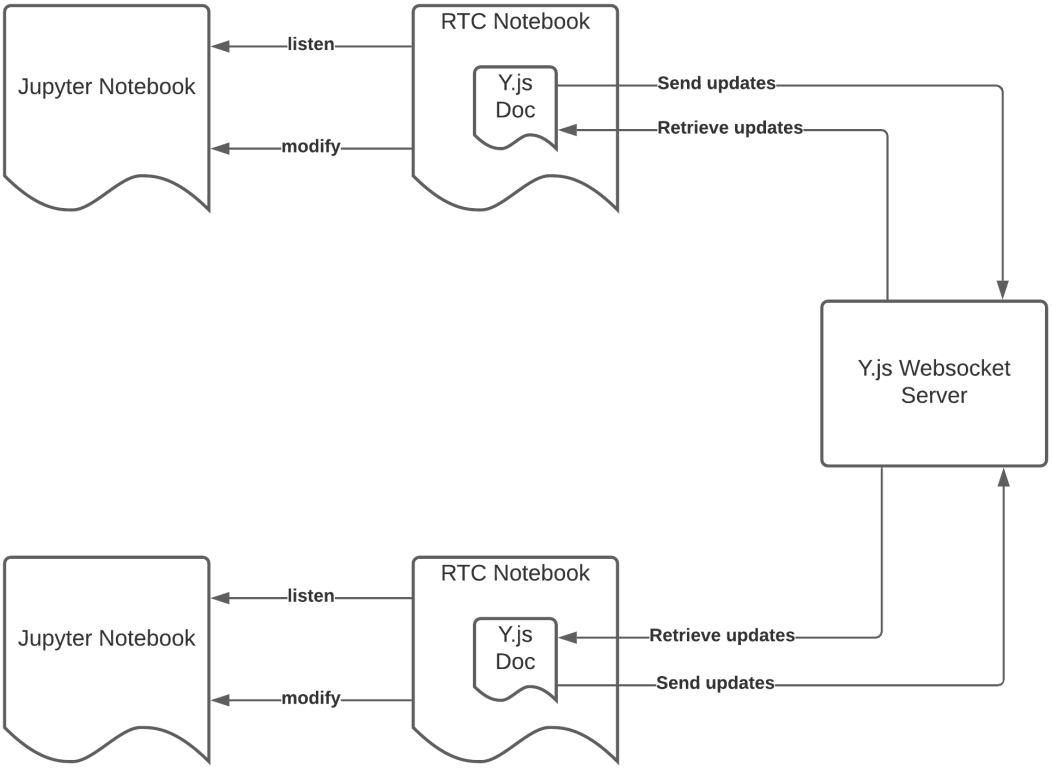


Figure 8.2: The *RTCNotebook*, which processes local and remote document updates, is kept alongside the original notebook and holds the shared document. Using this document, updates are distributed to all clients.

Chapter 9

Operations

9.1 Sharing a Notebook

As part of the RTC extension a share button is added to every notebook's toolbar as shown in Figure 9.1. To start sharing a notebook, the user can click this button and will then receive a unique id which his collaborators can use to access the shared notebook. Figure 9.2 illustrates the different operations when initialising a new shared notebook. When the user starts sharing his notebook, a new *rtc-id* will be generated and stored in the notebook's metadata. In addition, this *rtc-id* is also used to establish the WebSocket connection. Before showing this id to the user, the extension creates a new Y.js Doc and initialises all the required data for sharing. This data consists of three parts and is generated for every cell in the notebook.

- **rtc-id**
  To identify each cell, a unique *rtc-id* is generated for every cell and stored in the cell's metadata. This id is also used to store and retrieve the shared text and metadata. **Note:** At the time of development of the RTC extension, the notebook format already included an id for every cell. However this id changed every time a user saved the notebook. As this id is used to identify cells across shared notebooks, it can definitely not change throughout the lifetime of the notebook. Consequently, these id's could not be used by the RTC extension. JupyterLab was aware of this problem [1] and at the time of writing, notebook format 4.5 now includes unique cell id's.

- **Shared text**
  Using the cell's id, a shared text is created in the Y.Doc. The text is then initialised with the current content of the cell. Additionally, this shared text is bound to the cell such that whenever the cell's content changes, the remote text is changed accordingly and vice-versa. How this is accomplished is discussed in Section 9.3.

- **Shared cell metadata**
  Besides the cell's content, additional metadata is stored in the shared document. This metadata consists of the following data: the cell's *type*, it's *output*, the *execution count* and the *position* of the cell in the notebook. All this metadata will be used to synchronise operations such as cell addition/deletion, cell movement, cell execution and changing the cell's type. These operations are discussed in following sections.

[1] https://github.com/jupyterlab/jupyterlab/issues/9729

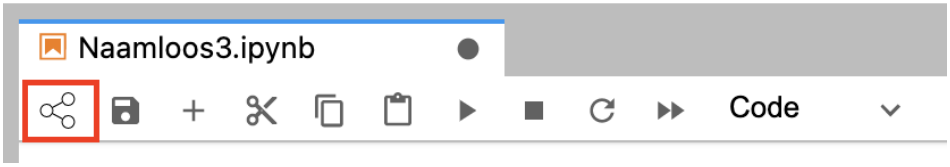


Figure 9.1: The RTC extension adds a simple button which allows the user to easily share his notebooks with collaborators

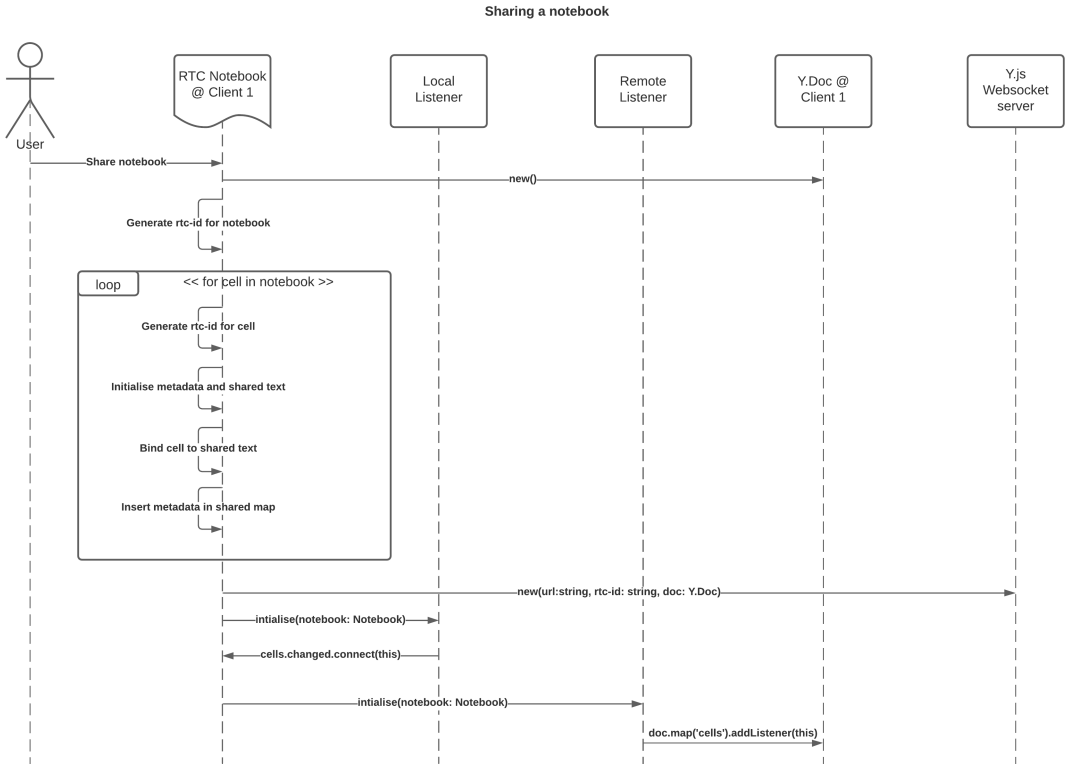


Figure 9.2: Sequence diagram showing the steps for sharing a notebook.

9.2 Connecting to a shared notebook

To connect to a shared notebook, the user receives the *rtc-id* for the shared notebook. Upon entering this id, the RTC extension creates a new notebook and synchronises this document with the shared notebook. A sequence diagram for this operation is shown in Figure 9.3. Upon creation of the new notebook, a new Y.doc is initialised and the connection to the WebSocket server is made using the *rtc-id* that was provided. Once the WebSocket is connected and indicates all data has been synchronised, the metadata map is loaded from the shared document. Each entry in the map is sorted according to its position such that all cells are inserted in the correct order. For each of these metadata entries a new cell is created based on its values. In addition, each cell is bound to the shared text that has the same id as the cell's *rtc-id*. This id is obtained as the key of its respective metadata entry. After inserting all the cells, the RTC extension enables the local and remote listeners to synchronise its state with other clients.

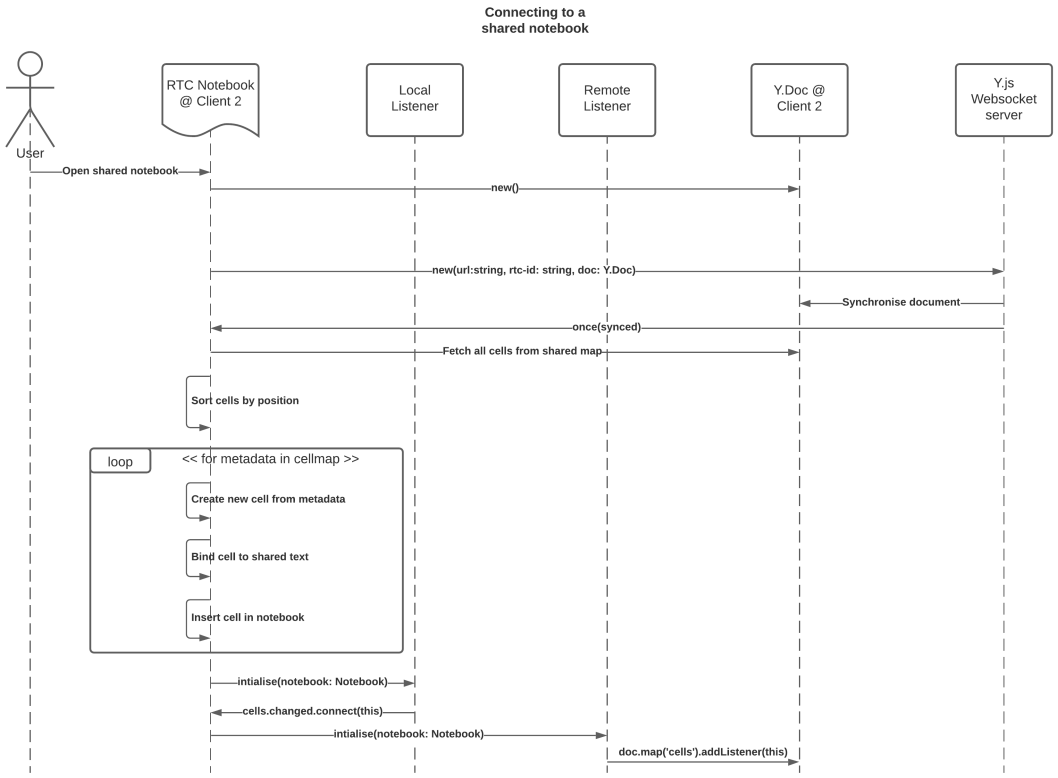


Figure 9.3: Sequence diagram illustrating the operations for connecting to a shared notebook.

9.3 Cell editing

Whenever a cell's input is modified, this input should be synchronised across all clients. To do so, Y.js' CodeMirror binding[2] is used. Jupyter uses the CodeMirror Editor[3] as underlying editor for notebook cells and thus Kevin Jahns' binding can be used to synchronise the input of every cell in the notebook. Each of these bindings is stored in a map so they can later be correctly destroyed. This is required for example when closing the notebook. When closing a notebook, all cells will be disposed and prior to this disposal their input will be cleared. If the binding is still activate at that time, all remote clients will end up with empty cells.

9.4 Adding and removing cells

Figure 9.4 shows the sequence diagram for a cell insertion. Based on this diagram two events are described: an addition and a deletion of a cell. Both operations follow a similar sequence of actions.

9.4.1 Addition

Whenever a user inserts a cell at a certain position, an event is fired by the notebook model. The RTC extension listens to these events using a local listener. The local listener will handle the addition of a cell by generating a unique *rtc-id* for that cell and initialising all its metadata such as position, execution count, type and outputs. It then sets the cell's input in the shared text that has the same id as the

[2]https://github.com/yjs/y-codemirror
[3]https://codemirror.net/

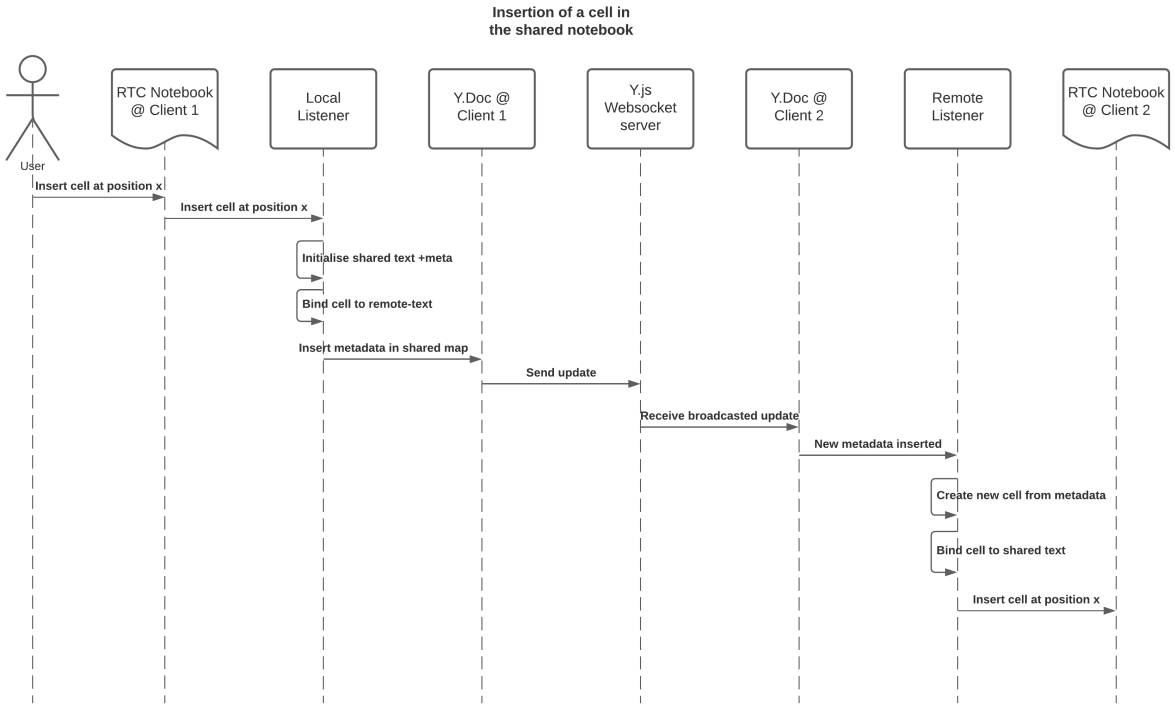


Figure 9.4: Sequence diagram showing the steps for inserting a cell in a shared notebook.

cell's *rtc-id* and bind this cell to this text using Y.js' CodeMirror binding. Finally the cell's metadata is added to the shared cell map. All those changes on the shared document are automatically forwarded to the WebSocket server. This server consequently distributes the changes to all other clients. These clients have a remote listener which is responsible for handling incoming changes. This listener receives an update which indicates that an entry is added to the cell map and acts accordingly. It creates a new cell locally, set its id and metadata based on the metadata that already resides in the cell map and insert this cell in the notebook. However, this notebook also has a local listener enabled like every other client that is connected. To avoid ending up in an endless loop, this local listener checks whether the added cell already has an *rtc-id* assigned. If so, the listener knows that this cell was added due to a remote update and will take no further action.

9.4.2 Deletion

The deletion of a cell is quite similar. Whenever the user deletes a cell, the local listener destroys the cell's text binding, remove the cell from the notebook, remove the cell metadata from the shared map and adjust the positions for all cells that were below the deleted cell in the notebook. The removal of the cell from the shared map again triggers an update which is received by all other clients. These clients destroy the cell's text binding and delete the cell from their notebook. Once again, this triggers the local listener at that notebook. To avoid an endless loop it suffices to simply check whether the cell's binding still exists. If not, the cell is already deleted due to a remote update and no further action is required.

9.5 Move cells

Figure 9.5 illustrates a user moving a cell to another position in the notebook. Upon this action from the user, the local listener is notified that the cell is moved from position x to position y. It subsequently updates the cell's position and in addition also updates the position of every cell between positions $x+1$ and y as these cells move along (either up or down) with the originally moved cell. The remote listener at every other connected client will receive several updates indicating that certain cells' positions have changed, starting with a position update for the originally moved cell.

This results in all remote clients moving this cell at first, before handling any other position update. Accordingly, these cells are already at their correct position and all subsequent position updates (from the cells that were originally located at positions $x+1$ through y) can be discarded.

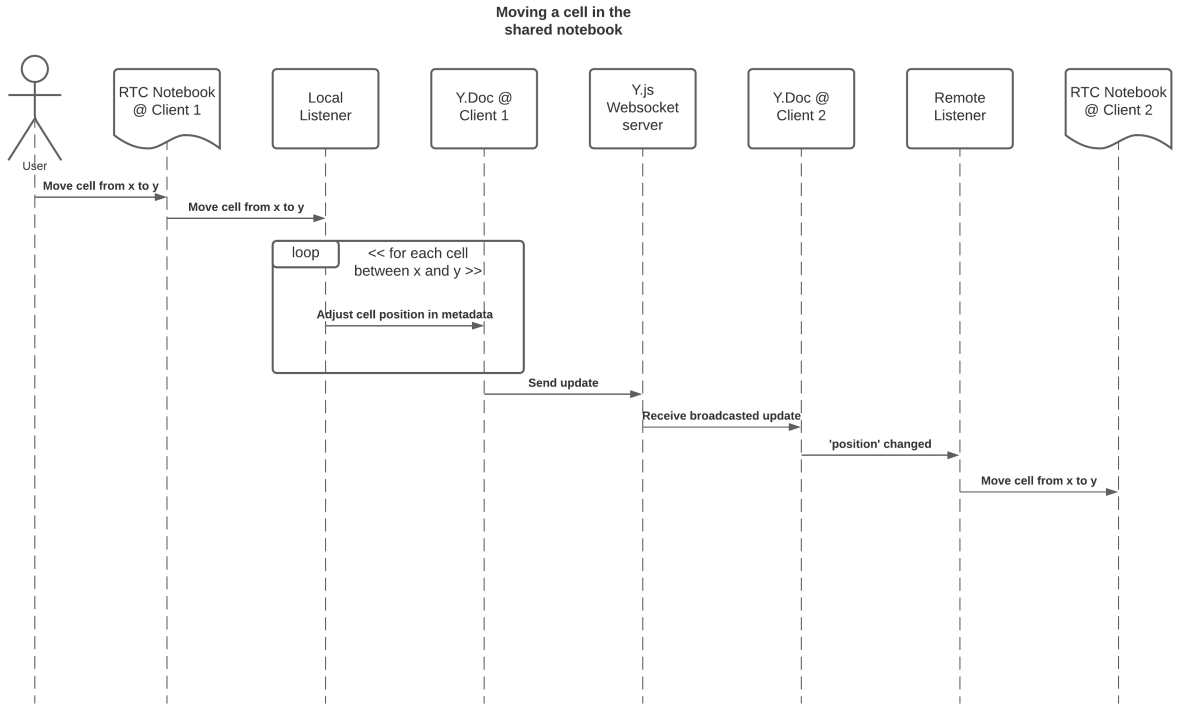


Figure 9.5: Sequence diagram showing the steps for moving a cell in a shared notebook.

9.6 Executing cells

In this prototype, cells are only executed by one client. This client is called the host and is initially selected as the client that starts sharing the notebook. For the execution of cells, two different cases are distinguished. Figure 9.6 displays the scenario where the user is not currently hosting the shared notebook. Upon execution of a cell, the local listener is notified that this cell has been executed (this happens instantaneously as a non-hosting user has no kernel linked with this notebook and thus no real execution takes place at this user's JupyterLab instance, see 10.1 for more details). The listener then checks whether the current user is hosting the notebook or not. If not, as in this scenario, the execution counter of the cell's metadata is increased. The remote listeners of all other clients receive an update indicating that this cell's execution counter has increased. Only the client that is hosting the notebook

responds to this update by executing the cell in his notebook and updating the cell's metadata with the output it received from this execution. All other clients will receive this update and set the cell's output accordingly in their notebooks.

Note that upon executing a cell, the extension checks whether the hosting client is still active. If this is not the case, the client that is currently requesting an execution becomes the host and notifies all other clients of this change.

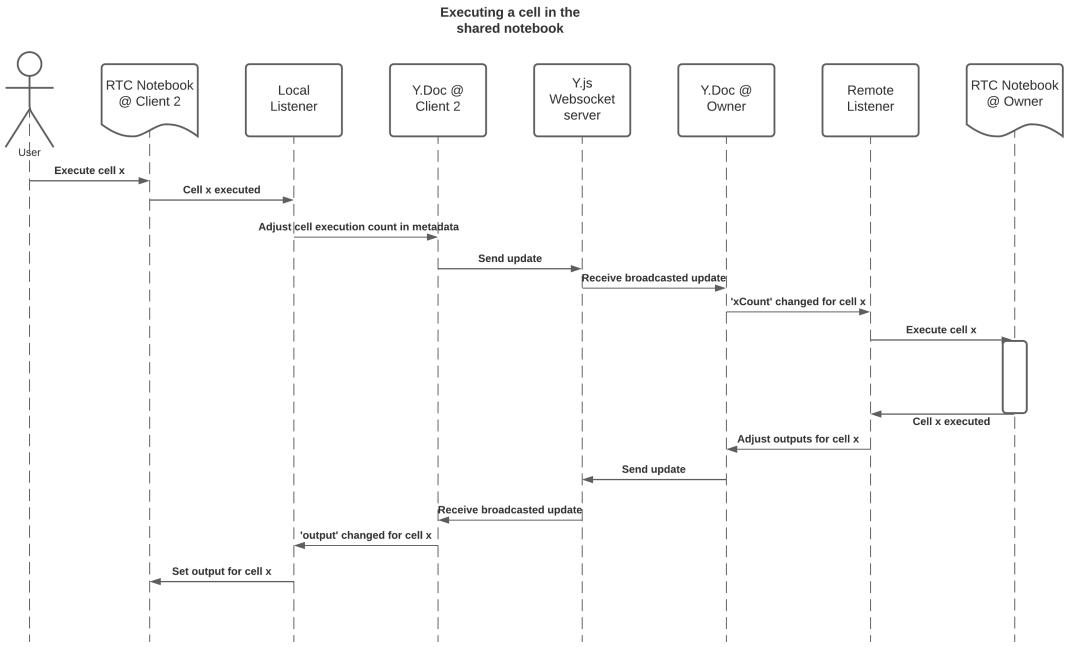


Figure 9.6: Sequence diagram showing the steps for executing a cell in a shared notebook.

In the other case, where the current user is hosting the notebook, the cell will be executed and upon completion the output will be stored in the cell's shared metadata. The other clients will receive this update an set the cell's output accordingly.

9.7 Change cell type

To change the type of a cell a user selects the new type for the cell from a dropdown list. Whenever this occurs, the local listener is notified that a certain cell is *set* which means that this cell is replaced by a new cell. The listener then verifies if the type of the new cell is different from the type of the old cell. If so, the binding for the old cell is destroyed and the new cell is re-bound to the shared text. Finally, the listener sets the *type* entry in the shared metadata to the new cell's type.

At the remote clients, this triggers the remote listener indicating that the type has changed for a certain cell. This cell's model is then serialised and a new model with the new type is created from this serialised cell. This causes all other cell data such as its content and metadata to persist across the type change. Successively, the old cell is replaced by this new cell and the cell is re-bound to the shared text.

The above flow is displayed in the sequence diagram in Figure 9.7.

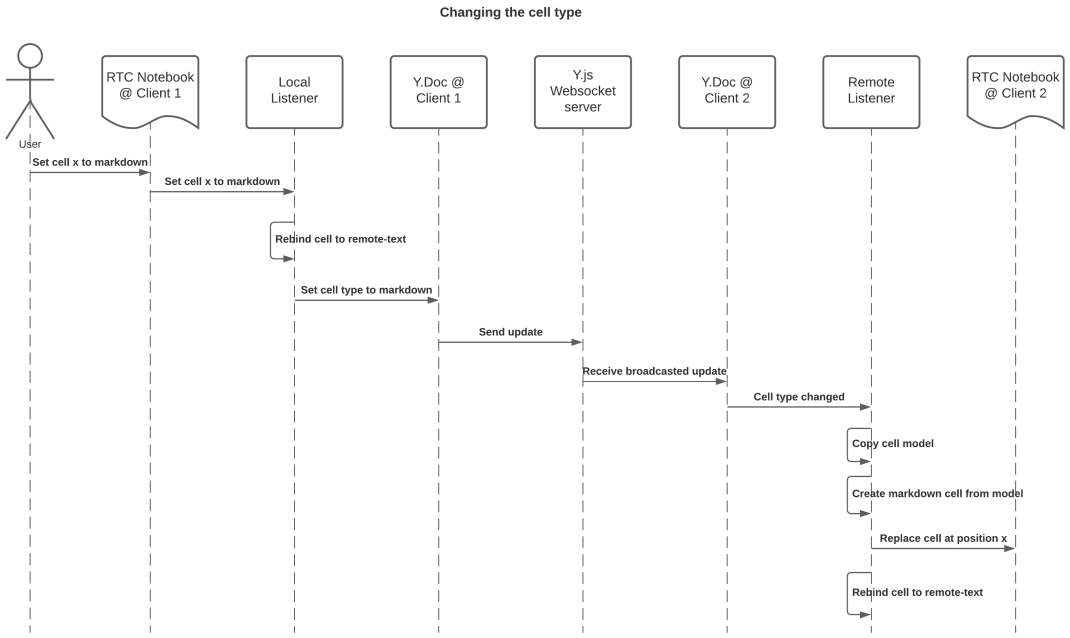


Figure 9.7: Sequence diagram showing the steps for changing a cell's type in a shared notebook.

9.8 Saving the notebook

Upon closing a Jupyter notebook, either the document is simply closed or a dialog is shown if the notebook has unsaved changes. The RTC extension overwrites both close- and save handlers with two purposes:

- **Make sure all bindings are destroyed.**
  Upon closing a notebook all widgets, such as cells and the notebook itself, are disposed. The disposal of a cell clears the cell's content before destroying it. As mentioned in section 9.3, if at that moment the cell binding is still active, the 'clearing' of the cell's content propagates to all remote clients which is not the intention of the user that is only closing his notebook. To avoid such issues, all listeners and text-bindings are destroyed before closing the notebook.

- **Save the shared structure to the notebook file.**
  Whenever a save is triggered the shared Y.js document is encoded and stored in the notebook's metadata. These encoded bytes can later be used to restore the document's shared state and to provide features such as offline editing. How this can be achieved is more thoroughly discussed in section 10.4.

Chapter 10

Future work

10.1 Kernel

To execute code, Jupyter uses so called kernels. Kernels are background processes which are bound to a specific programming language and run the code that is executed by the user. Whenever a user executes a code cell in the notebook, the cell's input is passed to the kernel and the kernel executes this input as code. Upon completion of execution, the result is published by the kernel and displayed in the notebook. Currently execution is implemented such that only the client that is hosting the notebook has an associated kernel and execute the code. All other clients then obtain an exact copy of the output from the host. From an educational perspective, this provides an easy way for teachers to debug a student's notebook and kernel state.

However, other possibilities such as separate kernels with shared execution or just separate kernels with local execution exist and whether or not to synchronise execution state heavily depends on the use case.

10.2 Integration

The Y.js WebSocket server is running alongside the JupyterHub server on the same host. However, to simplify the architecture and setup complexity it would be beneficial to integrate the WebSocket server within the JupyterHub or JupyterLab ecosystem. However as the Jupyter server side is written in Python this would require either some kind of binding between the WebSocket running on the Jupyter server and the Y.js server. Another option is to rewrite the Y.js server in python. However as this server uses components from the Y.js JavaScript library this is a difficult task. Furthermore a seamless switch between a shared notebook and regular notebook should be made possible.

10.3 Security

Currently, no security measurements have been taken. This means that everyone who has access to the JupyterHub server can access any document being shared on this server. Even worse, the Y.js server needs to be available externally as well for the extension to be able to function correctly. This means that anyone with the RTC extension and an *rtc-id* can access a shared document from outside the organisation by very little modification to the extension's code. This is of course not feasible. Simply keeping the shared

document id secret and only share it with people that are permitted to view the document theoretically suffices. However, in practice this is not a decent solution. Ideally, one would like to have some kind of permission system with which a user that is sharing the notebook can indicate permissions such as who has access to this notebook, can this user write to it and can they execute code.

10.4 Offline Editing

The prototype supports the loading of a local notebook file which is linked to a shared notebook. Upon opening this file the RTC extension synchronises this notebook with any changes being made by other clients. However when opening a notebook in a regular way, meaning that the notebook is not connected to the WebSocket, changes are not applied to the shared document. It is therefor not possible to synchronise the changes at a later time. To enable such functionality, a notebook that was once shared (thus contains an *rtc-id*) or that was made by connecting to a shared notebook, should always be linked to a Y.Doc. By doing this, all changes are reflected onto this shared document which is then stored inside the notebook file. This means that whenever a network connection is available, the shared Y.Doc can easily be connected to the WebSocket and all changes are broadcast to other clients and vice-versa. In short, this means that the RTC extension should be enabled at all time and active for all shared notebooks, regardless of the way they are opened.

Part IV

Conclusion

In this thesis, research is conducted into the possibilities of bringing real-time collaboration to Jupyter notebooks. The literature study shows that RTC is an active, well researched technology and many solutions are available for developers to embed real-time collaborative functions in their applications. To aid students and teachers in organising remote lab sessions using Jupyter notebooks, a proof-of-concept prototype that enables RTC in computational notebooks is developed. To do so, a candidate library is selected from two existing RTC frameworks. Y.js, which has proven itself to be the best performing JavaScript CRDT library of this moment, and Microsoft Fluid, which was only recently released to the public and shows great potential. As part of this thesis, both performance and usability tests are conducted on these libraries. To measure performance of the Fluid Framework an existing benchmark for Y.js was adapted to include Fluid and results for both libraries were compared. In addition, a study was conducted to get an indication of the usability of these frameworks. Based on both these studies, it is concluded that the Y.js library currently makes the best candidate to implement RTC in Jupyter. Although Microsoft Fluid is a decent framework and shows great potential, the current version is not deemed stable enough and continuous changes makes this library less fit for production-ready implementations.

Based on the above conclusion, a proof-of-concept extension for JupyterLab was build using Y.js. This prototype demonstrates the possibilities of RTC in Jupyter notebooks and proves that the current state-of-the-art technology on RTC is up for the task. Currently, the JupyterLab community is working on the development of fully integrated RTC functionality in their product through the use of the Y.js framework. After a long year of planning and several discussions in the community on how to implement RTC, it seems that the actual development recently kicked off and is making good progress. It is thus expected that RTC is coming to JupyterLab sooner rather than later. The work and research on this specific topic seems complete. However this thesis opens the road for other, related e-learning problems. Further research into how one can lower the threshold for teachers and students to organise remote assignments is one of the possibilities. For example tools which provide an easy way for deploying notebooks and providing students with a develop-ready environment without having to distribute all the required data manually (which can be a lot when working with e.g. big data) could really improve remote learning assignments.

Bibliography

[1] T. I. D. Team. Ipython documentation. [Online]. Available: https://ipython.readthedocs.io/en/7.22.0/whatsnew/version0.13.html

[2] P. Parente, "Estimate of public jupyter notebooks on github," 2014.

[3] Microsoft. Visual studio live share. [Online]. Available: https://visualstudio.microsoft.com/services/live-share/

[4] C. Sun and C. Ellis, "Operational transformation in real-time group editors: Issues, algorithms, and achievements," in *Proceedings of the 1998 ACM Conference on Computer Supported Cooperative Work*, ser. CSCW '98. New York, NY, USA: Association for Computing Machinery, 1998, p. 59–68. [Online]. Available: https://doi.org/10.1145/289444.289469

[5] S. Chengzheng. Operational transformation frequently asked questions and answers. [Online]. Available: https://www3.ntu.edu.sg/scse/staff/czsun/projects/otfaq/

[6] N. Fraser, "Differential synchronization," in *DocEng'09, Proceedings of the 2009 ACM Symposium on Document Engineering*, 2 Penn Plaza, Suite 701, New York, New York 10121-0701, 2009, pp. 13–20. [Online]. Available: http://neil.fraser.name/writing/sync/eng047-fraser.pdf

[7] Microsoft. Microsoft fluid framework - architecture. [Online]. Available: https://fluidframework.com/docs/concepts/architecture/

[8] J. Singer, "Notes on notebooks: Is jupyter the bringer of jollity?" in *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, ser. Onward! 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 180–186. [Online]. Available: https://doi.org/10.1145/3426428.3426924

[9] F. Perez and B. E. Granger, "Project jupyter: Computational narratives as the engine of collaborative data science," 2015.

[10] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, p. 558–565, Jul. 1978. [Online]. Available: https://doi.org/10.1145/359545.359563

[11] C. J. Fidge, "Timestamps in message-passing systems that preserve the partial ordering," *Proceedings of the 11th Australian Computer Science Conference*, vol. 10, no. 1, p. 56–66, 1988. [Online]. Available: http://sky.scitech.qut.edu.au/ fidgec/Publications/fidge88a.pdf

[12] F. Mattern, "Virtual time and global states of distributed systems," in *Parallel and Distributed Algorithms*. North-Holland, 1989, pp. 215–226.

[13] F. B. Schmuck, "The use of efficient broadcast protocols in asynchronous distributed systems," Ph.D. dissertation, USA, 1988, aAI8900827.

[14] B. Liskov and R. Ladin, "Highly available distributed services and fault-tolerant distributed garbage collection," in *Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '86. New York, NY, USA: Association for Computing Machinery, 1986, p. 29–39. [Online]. Available: https://doi.org/10.1145/10590.10593

[15] C. Sun, Y. Yang, Y. Zhang, and D. Chen, "A consistency model and supporting schemes for real-time cooperative editing systems," 1996.

[16] C. A. Ellis and S. J. Gibbs, "Concurrency control in groupware systems," *SIGMOD Rec.*, vol. 18, no. 2, p. 399–407, Jun. 1989. [Online]. Available: https://doi.org/10.1145/66926.66963

[17] 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 ACM Symposium on User Interface and Software Technology*, ser. UIST '95. New York, NY, USA: Association for Computing Machinery, 1995, p. 111–120. [Online]. Available: https://doi.org/10.1145/215585.215706

[18] M. Shapiro and N. Preguiça, "Designing a commutative replicated data type," INRIA, Research Report RR-6320, 2007. [Online]. Available: https://hal.inria.fr/inria-00177693

[19] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-free Replicated Data Types," in *SSS 2011 - 13th International Symposium Stabilization, Safety, and Security of Distributed Systems*, ser. Lecture Notes in Computer Science, X. Défago, F. Petit, and V. Villain, Eds., vol. 6976. Grenoble, France: Springer, Oct. 2011, pp. 386–400. [Online]. Available: https://hal.inria.fr/hal-00932836

[20] G. Oster, P. Urso, P. Molli, and A. Imine, "Real time group editors without Operational transformation," INRIA, Research Report RR-5580, 2005. [Online]. Available: https://hal.inria.fr/inria-00071240

[21] E. Karayel and E. Gonzalez, "Strong eventual consistency of the collaborative editing framework woot," 03 2020.

[22] N. Preguica, J. M. Marques, M. Shapiro, and M. Letia, "A commutative replicated data type for cooperative editing," in *Proceedings of the 2009 29th IEEE International Conference on Distributed Computing Systems*, ser. ICDCS '09. USA: IEEE Computer Society, 2009, p. 395–403. [Online]. Available: https://doi.org/10.1109/ICDCS.2009.20

[23] S. Weiss, P. Urso, and P. Molli, "Logoot: A Scalable Optimistic Replication Algorithm for Collaborative Editing on P2P Networks," in *29th IEEE International Conference on Distributed Computing Systems - ICDCS 2009*, ser. 2009 29th IEEE International Conference on Distributed Computing Systems. Montreal, Canada: IEEE, Jun. 2009, pp. 404–412. [Online]. Available: https://hal.inria.fr/inria-00432368

[24] K. Martin, B. Annette, and S. Marc. Conflict-free replicated data types. [Online]. Available: https://crdt.tech/

[25] N. Fraser, "Differential synchronization," *DocEng'09 - Proceedings of the 2009 ACM Symposium on Document Engineering*, pp. 13–20, 01 2009.

[26] Micrososft. (2020) Fluid framework. [Online]. Available: https://fluidframework.com/

[27] Microsoft 365 Team. (2020, Sep) Fluid framework is now open source. [Online]. Available: https://developer.microsoft.com/en-us/office/blogs/fluid-framework-is-now-open-source/

[28] B. Posey. (2021, Jan) Why microsoft's fluid framework may be the next big thing. [Online]. Available: https://redmondmag.com/articles/2021/01/06/fluid-framework-next-big-thing.aspx

[29] Microsoft. (2020, Sep) Microsoft fluid framework. [Online]. Available: https://www.microsoft.com/en-us/videoplayer/embed/RE4wEb1

[30] ——. (2020) Fluid framework - frequently asked questions. [Online]. Available: https://fluidframework.com/start/faq/

[31] P. Nicolaescu, K. Jahns, M. Derntl, and R. Klamma, "Near real-time peer-to-peer shared editing on extensible data types," 11 2016, pp. 39–49.

[32] K. Jahns. (2020, Dec) Are crdts suitable for shared editing? [Online]. Available: https://blog.kevinjahns.de/are-crdts-suitable-for-shared-editing/

[33] M. Haverbeke. Prosemirror. [Online]. Available: https://prosemirror.net/

[34] ——. Codemirror. [Online]. Available: https://codemirror.net/

[35] M. Kleppmann and A. R. Beresford, "A conflict-free replicated json datatype," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 10, p. 2733–2746, Oct 2017. [Online]. Available: http://dx.doi.org/10.1109/TPDS.2017.2697382

[36] M. Kleppmann and A. R. Beresford, "Automerge: Real-time data sync between edge devices," 2018. [Online]. Available: https://www.mobiuk.org/abstract/S4-P5-Kleppmann-Automerge.pdf

[37] M. Kleppmann, A. Wiggins, P. van Hardenberg, and M. McGranaghan, "Local-first software: You own your data, in spite of the cloud," in *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, ser. Onward! 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 154–178. [Online]. Available: https://doi.org/10.1145/3359591.3359737

[38] P. S. Almeida, A. Shoker, and C. Baquero, "Delta state replicated data types," *CoRR*, vol. abs/1603.01529, 2016. [Online]. Available: http://arxiv.org/abs/1603.01529

[39] N. Smith and J. Gentle. Sharedb. [Online]. Available: https://share.github.io/sharedb/

[40] D. Wang, A. Mah, and S. Lassen. (2010, Jul) Google wave operational transformation. [Online]. Available: https://svn.apache.org/repos/asf/incubator/wave/whitepapers/operational-transform/operational-transform.html

[41] A. Y. Wang, Z. Wu, C. Brooks, and S. Oney, "Callisto: Capturing the "why" by connecting conversations with computational narratives," in *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, ser. CHI '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1–13. [Online]. Available: https://doi.org/10.1145/3313831.3376740

[42] A. Y. Wang, A. Mittal, C. Brooks, and S. Oney, "How data scientists use computational notebooks for real-time collaboration," *Proc. ACM Hum.-Comput. Interact.*, vol. 3, no. CSCW, Nov. 2019. [Online]. Available: https://doi.org/10.1145/3359141

[43] SageMath Inc. Cocalc - collaborative calculation and data science. [Online]. Available: https://cocalc.com/

[44] Apache Software Foundation. Zeppelin. [Online]. Available: https://zeppelin.apache.org/

[45] DeepNote. Deepnote. [Online]. Available: https://deepnote.com/

[46] The Etherpad Foundation. Etherpad. [Online]. Available: https://etherpad.org/

[47] T. Baumann. Firepad - an pen source collaborative code and text editor. [Online]. Available: https://firepad.io/

[48] I. Appjet. (2018, Dec) Etherpad and easysync technical manual. [Online]. Available: https://github.com/ether/etherpad-lite/blob/develop/doc/easysync/easysync-full-description.pdf

[49] Microsoft. Visual studio live share. [Online]. Available: https://visualstudio.microsoft.com/services/live-share

[50] GitHub. Teletype for atom. [Online]. Available: https://teletype.atom.io/

[51] JetBrains. Code with me - the ultimate collaborative development by jetbrains. [Online]. Available: https://www.jetbrains.com/code-with-me/

[52] CodeCollab. Codecollab. [Online]. Available: https://codecollab.io/

[53] G. D'Angelo, A. Di Iorio, and S. Zacchiroli, "Spacetime characterization of real-time collaborative editing," *Proc. ACM Hum.-Comput. Interact.*, vol. 2, no. CSCW, Nov. 2018. [Online]. Available: https://doi.org/10.1145/3274310

Appendix

N = 6000	**Y.js**	**Fluid**	**Fluid (Tinylicious)**
Version	13.5.2	0.36.0	0.36
Bundle size	77013 bytes	1219674 bytes	1219674 bytes
Bundle size (gzipped)	22509 bytes	335508 bytes	335508 bytes
[B1.1] Append N characters (time)	263 ms	3305 ms	2007 ms
[B1.1] Append N characters (avgUpdateSize)	27 bytes	599 bytes	574 bytes
[B1.1] Append N characters (docSize)	6031 bytes	11099 bytes	10991 bytes
[B1.1] Append N characters (memUsed)	0 B	65.9 MB	51.3 MB
[B1.1] Append N characters (parseTime)	7 ms	468 ms	573 ms
[B1.2] Insert string of length N (time)	6 ms	164 ms	13 ms
[B1.2] Insert string of length N (avgUpdateSize)	6031 bytes	6629 bytes	6564 bytes
[B1.2] Insert string of length N (docSize)	6031 bytes	11084 bytes	10979 bytes
[B1.2] Insert string of length N (memUsed)	0 B	0 B	186.8 KB
[B1.2] Insert string of length N (parseTime)	14 ms	20 ms	56 ms
[B1.3] Prepend N characters (time)	241 ms	3270 ms	1946 ms
[B1.3] Prepend N characters (avgUpdateSize)	27 bytes	586 bytes	571 bytes
[B1.3] Prepend N characters (docSize)	6041 bytes	11099 bytes	10991 bytes
[B1.3] Prepend N characters (memUsed)	3.7 MB	62 MB	48.5 MB
[B1.3] Prepend N characters (parseTime)	27 ms	449 ms	587 ms
[B1.4] Insert N characters at random positions (time)	269 ms	3496 ms	2084 ms
[B1.4] Insert N characters at random positions (avgUpdateSize)	29 bytes	607 bytes	574 bytes
[B1.4] Insert N characters at random positions (docSize)	29554 bytes	11099 bytes	10991 bytes
[B1.4] Insert N characters at random positions (memUsed)	0 B	51.2 MB	49.1 MB
[B1.4] Insert N characters at random positions (parseTime)	37 ms	468 ms	536 ms
[B1.5] Insert N words at random positions (time)	340 ms	5228 ms	3550 ms
[B1.5] Insert N words at random positions (avgUpdateSize)	36 bytes	611 bytes	611 bytes
[B1.5] Insert N words at random positions (docSize)	87924 bytes	41288 bytes	41180 bytes
[B1.5] Insert N words at random positions (memUsed)	0 B	64.8 MB	63.7 MB
[B1.5] Insert N words at random positions (parseTime)	58 ms	667 ms	725 ms
[B1.6] Insert string, then delete it (time)	8 ms	163 ms	24 ms
[B1.6] Insert string, then delete it (avgUpdateSize)	6053 bytes	7253 bytes	7135 bytes
[B1.6] Insert string, then delete it (docSize)	38 bytes	5071 bytes	4966 bytes
[B1.6] Insert string, then delete it (memUsed)	0 B	0 B	0B
[B1.6] Insert string, then delete it (parseTime)	17 ms	17 ms	47 ms
[B1.7] Insert/Delete strings at random positions (time)	325 ms	6675 ms	5436 ms
[B1.7] Insert/Delete strings at random positions (avgUpdateSize)	31 bytes	647 bytes	645 bytes
[B1.7] Insert/Delete strings at random positions (docSize)	28377 bytes	7605 bytes	7497 bytes
[B1.7] Insert/Delete strings at random positions (memUsed)	4.3 MB	77.6 MB	65.2 MB
[B1.7] Insert/Delete strings at random positions (parseTime)	35 ms	843 ms	931 ms
[B1.8] Append N numbers (time)	287 ms	3470 ms	2316 ms
[B1.8] Append N numbers (avgUpdateSize)	32 bytes	605 bytes	594 bytes
[B1.8] Append N numbers (docSize)	35636 bytes	67960 bytes	67852 bytes
[B1.8] Append N numbers (memUsed)	0 B	53.9 MB	52.6 MB
[B1.8] Append N numbers (parseTime)	16 ms	460 ms	606 ms
[B1.9] Insert Array of N numbers (time)	15 ms	173 ms	40 ms
[B1.9] Insert Array of N numbers (avgUpdateSize)	35659 bytes	63610 bytes	63548 bytes
[B1.9] Insert Array of N numbers (docSize)	35659 bytes	68099 bytes	67994 bytes
[B1.9] Insert Array of N numbers (memUsed)	0 B	0 B	452.4 KB
[B1.9] Insert Array of N numbers (parseTime)	17 ms	23 ms	61 ms

[B1.10] Prepend N numbers (time)	232 ms	3325 ms	2256 ms
[B1.10] Prepend N numbers (avgUpdateSize)	32 bytes	604 bytes	591 bytes
[B1.10] Prepend N numbers (docSize)	35667 bytes	68027 bytes	67919 bytes
[B1.10] Prepend N numbers (memUsed)	6.7 MB	65.3 MB	52.8 MB
[B1.10] Prepend N numbers (parseTime)	27 ms	445 ms	572 ms
[B1.11] Insert N numbers at random positions (time)	265 ms	3599 ms	2215 ms
[B1.11] Insert N numbers at random positions (avgUpdateSize)	34 bytes	615 bytes	593 bytes
[B1.11] Insert N numbers at random positions (docSize)	59139 bytes	67985 bytes	67877 bytes
[B1.11] Insert N numbers at random positions (memUsed)	0 B	52.9 MB	51.9 MB
[B1.11] Insert N numbers at random positions (parseTime)	36 ms	495 ms	605 ms
[B2.1] Concurrently insert string of length N at index 0 (time)	5 ms	2 ms	29 ms
[B2.1] Concurrently insert string of length N at index 0 (updateSize)	12058 bytes	13181 bytes	13125 bytes
[B2.1] Concurrently insert string of length N at index 0 (docSize)	12149 bytes	17193 bytes	17088 bytes
[B2.1] Concurrently insert string of length N at index 0 (memUsed)	0 B	0 B	0 B
[B2.1] Concurrently insert string of length N at index 0 (parseTime)	14 ms	16 ms	58 ms
[B2.2] Concurrently insert N characters at random positions (time)	131 ms	7098 ms	2374 ms
[B2.2] Concurrently insert N characters at random positions (updateSize)	66250 bytes	7203803 bytes	6883904 bytes
[B2.2] Concurrently insert N characters at random positions (docSize)	66346 bytes	17204 bytes	17099 bytes
[B2.2] Concurrently insert N characters at random positions (memUsed)	7.9 MB	123.5 MB	98.5 MB
[B2.2] Concurrently insert N characters at random positions (parseTime)	43 ms	910 ms	1004 ms
[B2.3] Concurrently insert N words at random positions (time)	220 ms	7298 ms	2376 ms
[B2.3] Concurrently insert N words at random positions (updateSize)	177626 bytes	7335937 bytes	6948149 bytes
[B2.3] Concurrently insert N words at random positions (docSize)	177775 bytes	76846 bytes	76741 bytes
[B2.3] Concurrently insert N words at random positions (memUsed)	16.9 MB	135.5 MB	114.2 MB
[B2.3] Concurrently insert N words at random positions (parseTime)	132 ms	982 ms	971 ms
[B2.4] Concurrently insert & delete (time)	447 ms	23990 ms	5920 ms
[B2.4] Concurrently insert & delete (updateSize)	278576 bytes	14481649 bytes	13936410 bytes
[B2.4] Concurrently insert & delete (docSize)	278720 bytes	82328 bytes	82223 bytes
[B2.4] Concurrently insert & delete (memUsed)	29.1 MB	249.6 MB	225.5 MB
[B2.4] Concurrently insert & delete (parseTime)	255 ms	2315 ms	2502 ms
[B3.1] 250 clients concurrently set number in Map (time)	29 ms	699 ms	1041 ms
[B3.1] 250 clients concurrently set number in Map (updateSize)	7927 bytes	4071445 bytes	150547 bytes
[B3.1] 250 clients concurrently set number in Map (docSize)	5249 bytes	4582 bytes	4475 bytes
[B3.1] 250 clients concurrently set number in Map (memUsed)	939.8 kB	131.7 MB	202.8 MB
[B3.1] 250 clients concurrently set number in Map (parseTime)	228 ms	305 ms	425 ms
[B3.2] 250 clients concurrently set Object in Map (time)	42 ms	760 ms	1287 ms
[B3.2] 250 clients concurrently set Object in Map (updateSize)	15380 bytes	4078418 bytes	157542 bytes
[B3.2] 250 clients concurrently set Object in Map (docSize)	6768 bytes	4618 bytes	4616 bytes
[B3.2] 250 clients concurrently set Object in Map (memUsed)	0 B	130.6 MB	200.5 MB
[B3.2] 250 clients concurrently set Object in Map (parseTime)	225 ms	299 ms	414 ms
[B3.3] 250 clients concurrently set String in Map (time)	41 ms	794 ms	1453 ms
[B3.3] 250 clients concurrently set String in Map (updateSize)	167990 bytes	4231288 bytes	310375 bytes
[B3.3] 250 clients concurrently set String in Map (docSize)	5743 bytes	4835 bytes	4833 bytes
[B3.3] 250 clients concurrently set String in Map (memUsed)	0 B	131.1 MB	239.1 MB
[B3.3] 250 clients concurrently set String in Map (parseTime)	228 ms	302 ms	424 ms
[B3.4] 250 clients concurrently insert text in Array (time)	36 ms	23839 ms	5662 ms
[B3.4] 250 clients concurrently insert text in Array (updateSize)	8383 bytes	4066825 bytes	145996 bytes
[B3.4] 250 clients concurrently insert text in Array (docSize)	4158 bytes	7022 bytes	6915 bytes
[B3.4] 250 clients concurrently insert text in Array (memUsed)	0 B	552.1 MB	414.6 MB
[B3.4] 250 clients concurrently insert text in Array (parseTime)	223 ms	479 ms	524 ms
[B4] Apply real-world editing dataset (time)	4837 ms	3573627 ms	37086 ms
[B4] Apply real-world editing dataset (updateSize)	29 bytes	586 bytes	588 bytes
[B4] Apply real-world editing dataset (docSize)	159929 bytes	127041 bytes	127041 bytes
[B4] Apply real-world editing dataset (memUsed)	2.8 MB	764.9 MB	355.4 MB
[B4] Apply real-world editing dataset (parseTime)	74 ms	32022 ms	41669 ms

Table A1: Benchmark results for Y.js, Microsoft Fluid and Microsoft Fluid with Tinylicious

FACULTY OF ENGINEERING AND ARCHITECTURE

Real-Time Collaboration in Jupyter Notebooks

Jan Mariën
Student number: 01507966

Supervisors: Prof. dr. ir. Filip De Turck, Prof. dr. Bruno Volckaert
Counsellors: ing. Merlijn Sebrechts, Sander Borny

Master's dissertation submitted in order to obtain the academic degree of
Master of Science in Computer Science Engineering

Academic year 2020-2021

GHENT UNIVERSITY