CRDTs for Fonto

by

Quentin Lee
Martin Li
Cas van Rijn
Wang Hao Wang

to obtain the bachelor degree of Computer Science and Engineering
at the Delft University of Technology

TU Delft Coach: Bart Gerritsen
Bachelor Project Coordinator: Huijuan Wang

Client: Bert Willems (Fonto)
Advisors: Stef Busking and Martin Middel (Fonto)

This report is confidential and cannot be made public until July 1, 2020.

TU Delft

Summary

During this project a research has been done for Fonto whether CRDTs(Conflict-free Replicated Data Type) can be used to enable collaborative editing in block based documents such as XML. The approach of this research was to make a prototype as a proof of concept, to show if block-based operations such as inserting, deleting, moving, merging and splitting blocks can be supported with a CRDT implementation. Compared to normal text editing, block text editing can lead to more types of conflicts since it support more types of operations.

CRDTs are used to be able to have multiple replicas of a data structure which can be independently updated and changed whilst guaranteeing that eventually all replicas converge to the same state. CRDTs are currently being used for counters, timers and other types of data structures. There are also CRDTs for simple text files, only supporting inserting and deleting characters.

To enable collaborative editing in block based documents, firstly, a CRDT implementation was chosen based upon the research done for different types of CRDTs. After Logoot was chosen as the most extendable and suitable CRDT implementation for block-based editing, an already existing implementation was pulled from GitHub as starting point. This implementation already supported the insert and delete operations for characters, but did not yet support block operations.

The chosen Logoot implementation was extended with block operations. To validate the robustness of the designed solutions for these operations, tests were written using the Mocha testing framework and Chai assertion library. For different types of scenarios, tests were written and checked whether all replicas eventually converged to the same state. The created CRDT was also connected to a text editor, which was extended with block functionality to enable manual testing and debugging.

The operations insert, delete, move and splitting of blocks were successfully implemented. All replicas converged when using these operations in different test cases. No bugs were found when running these test cases, even when replicas worked offline for multiple operations. The merge operation is the most complex operation of all implemented block operations. The merge operation still has situations in which replicas diverge or even content is lost. As of now, there are still two open issues with merge, the first is that it is possible to have complex circular merges resulting in loss of content. The second issue is when splitting and merging multiple times in the same block offline, as this results into diverging states of replicas.

The research prototype shows that it is possible to support block operations except for the merge operations. Strengths of the created prototype are that it is easily extendable with different types of blocks and nested blocks, which are both prevalent in XML structures. The biggest weakness is that some of the operations are inefficient, resulting in slowdowns for larger documents or big operations such as copy pasting large amount of text.

Recommendations for Fonto and/or other further research are to investigate whether it is possible to fix the issues with merge and ensure all operations work in combination with each other. It is also advised to see if it is possible to lower the complexity of the operations, resulting in better performance. It is also recommended to investigate other CRDT structures to see whether their design could lead to other insights regarding complexity and the merge operation.

Preface

This research topic comes from the desire of Fonto to research the possibilities to enable users to collaboratively work in the same XML document. We want to thank Fonto for giving us the possibility to dive deeper into the world of collaborative editing and for giving us the opportunity to research CRDTs, which is a field that is rather new and therefore gave us a lot of freedom. This has been a very exciting project for all of us where we were able to put our knowledge learned at the university to use in a practical environment. We want to thank our supervisors from Fonto, Stef Busking, Martin Middel and Bert Willems for their guidance and effort during the research and Remko Zuiderwijk for helping us during our time in the office. They were always available and happy to help us and/or give their input on challenging issues. We also want to thank Bart Gerritsen, our supervisor from the TU Delft for ensuring that the educational demands from the TU Delft were met. We thoroughly enjoyed this project and our stay at Fonto.

Quentin Lee
Martin Li
Cas van Rijn
Wang Hao Wang
Delft, June 2020

Contents

1 Introduction

1.1. About the project

The Bachelor End Project (TI3806) is the last course in the bachelor program for Computer Science and Engineering at the Delft University of Technology. During this project, all knowledge that was acquired throughout the years will have to be applied. The topic of the project is about Conflict-Free Replicated Data Types (CRDT). In short CRDTs are a data structure which allows itself to be replicated, where each replica has the exact same state, even after one replica applies changes. This data structure will be the backbone for a real-time concurrent editing feature for the client.

1.2. Fonto

The client for this project is Fonto. Fonto is a company that started in 2013 and their product is an online structured content editor. Some clients of Fonto are: SDL, Huawei and Spotify. Fonto aims to make editing XML as easy as editing a Word document. The Fonto editor provides a rich text, What You See Is What You Mean (WYSIWYM) editing experience which can be configured to fit any XML schema.

1.3. Problem definition

Currently, the ability for clients to work together on one document is very limited. The Fonto editor can load multiple documents at the same time and it is possible for multiple users to connect to the same editor and work on distinct documents. When someone starts working in a document, the document will be locked for every other user until it gets unlocked again by the user who first got the lock. So only one author can work on any part of a document at a time. Fonto would like to lift this limitation in the future and enable true concurrent editing, as seen in Google Docs. The difference in complexity between Google Docs and Fonto lies in the support of many extended features that the Fonto editor has, e.g. support for arbitrary XML schemata and the ability for Fonto's partners to write any custom mutation to manipulate the XML model. CRDTs could be used to add this functionality to the Fonto editor.

1.4. CRDT

A conflict-free replicated data type (CRDT) is a data structure which can be replicated across multiple machines in a network, where the replicas can be updated independently and concurrently without coordination between the replicas, and where it is always mathematically possible to resolve inconsistencies between the data structures.

The CRDT concept was formally defined in 2011 by Marc Shapiro, Nuno Preguiça, Carlos Baquero and Marek Zawirski. Development of CRDTs was initially motivated by collaborative text editing and mobile computing, but CRDTs have also been used in online chat systems, online gambling, and in online games.

In this project, CRDTs will be used for real-time collaborative editing. Although CRDTs are fairly new concepts, there are already a few known CRDT implementations. The main goal of this project is to conduct a research about CRDTs and to create a research prototype of a CRDT that fits Fonto's needs. First, research was conducted about CRDTs to gain general knowledge about the problem. After that, an in-depth problem analysis was done and requirements were formed. Then based on the analysis and the requirements, a proof of concept prototype was made to show that it is possible to support block operations with a CRDT.

1.5. Development process

For this project the Scrum methodology was used for the development process. At the end of each one-week sprint, a planning was made with issues to work on for the next sprint. There was no dedicated Scrum master, prioritisation of issues was done together. The product owner was the client of the project, but a lot of freedom was given to the development team on how the final product would look like.

The development team worked with a pull-based development model. The code is hosted on GitHub, where one 'master' branch contains the production version of the product. All new functionalities should be merged with the master branch through pull-requests which should be reviewed by other developers. This ensures high code quality and a lower chance of bugs occurring in the master branch.

To improve the development process, Travis CI has been used as a continuous integration (CI) tool. The CI runs all tests and checks for code style errors to ensure all code is written in the same style. Finally, Codecov has been added to this CI, so one can ensure that new functionality will also be tested without dropping the code coverage.

2

Research report

2.1. Introduction

In this research report, research will be done about real-time collaborative editing. Fonto currently has a product, which is an editor for structured documents. Fonto does not have a single universal editor. Clients are free to use their own XML schema and add their own custom operations. XML schemas are the set of rules, which the XML structure has to respect. However, one limitation is that it does not support real-time collaborative editing. Currently, writers have to lock files to ensure that no one else is editing the file. This research will take a look at different algorithms which can be used to implement real-time collaborative editing and what algorithm would fit best for the current editor used by Fonto.

Creating a collaborative editor nowadays requires quite some time and effort to implement as these editors need to support all kinds of different operations, ranging from removing characters to undoing previous changes. Designing and implementing such system may take up to two years and therefore would not be suitable for this project. In order to fit this into the given time frame of ten weeks, research is conducted in the first two weeks to find key points of already defined systems. The most commonly used systems for collaborative editing are operational transform (OT) and conflict-free replicated data type (CRDT). As the title of this graduation project implicates, Fonto suggests to look further into CRDTs. However, the possibility to use other approaches should still be considered as it might be a better and more fitting approach. This research report will mainly compare all kinds of possibilities and will serve as support for the decisions made for Fonto.

This research will tackle three research questions:

1. Why CRDT over other collaborative editing algorithms?
2. What types of CRDTs are there?
3. Which CRDTs are the best fit for Fontos purpose?

The first research question will discuss why CRDTs should be used instead of other collaborative editing algorithms such as OT. The second research question will go over different types of CRDTs and their use case, advantages and disadvantages. Finally, the last research question will discuss the different CRDTs and decide what CRDTs fit best for Fonto's purpose.

2.2. CRDTs

2.2.1. What are CRDTs?

A conflict-free replicated data type (CRDT) is a data structure where replication is the main aspect. The data structure behind a CRDT can be replicated across multiple devices in a network, where each replica can be updated independently and concurrently. A CRDT should always mathematically be possible to resolve inconsistencies and different replicas should eventually converge to the same state. CRDTs are the answer to the problem of synchronising data in distributed environments [26].

2.2.2. What does conflict-free mean?

Conflict-free has a very broad description, but in distributed systems it means that for a data structure, that it does not need exclusive write access and it is able to detect concurrent updates and perform deterministic, automatic conflict resolution. This means that the output is always able to be determined upfront, based on the metadata contained within the data structure itself [26]. For a real-time collaborative CRDT to be considered correct, it has to respect the CCI criteria [28]:

- Causality: All operations are ordered by a precedence relation
- Convergence: The system converges if all replicas are identical when the system is idle.
- Intention: The expected effect of an operation should be observed on all replicas.

For CRDTs, the core structures are sets, registers, and counters, but from these structures complex structures like trees, maps, graphs, and even XML can be composed.

2.2.3. What does replication mean?

In essence, replication means the act copying or reproducing something. So for ”replication” in the context of CRDTs and real-time collaborative editing, it is about copying documents. This means that several peers can edit a document from different places. Each peer hosts its own replica of the collaborative document. The replica contains elements visible to the end user as well as metadata required to manage eventual consistency [5].

There are different replication types. There is a difference between pessimistic, optimistic replication, active and passive replication, and operation-based and state-based replication [7].

- Pessimistic replication: In pessimistic replication, the system simulates that there is a unique copy of the object despite all clients working on different replicas, but because every replica has the exact same content, it creates the illusion of having one single document. This method requires synchronisation among replicas to guarantee that no stale data is retrieved.
- Optimistic replication: Optimistic solutions allow replicas to diverge. This means that clients can read/write different values, for the same object, if they contact different replicas. When using such level of consistency the application must be aware of this fact.
- Active replication: In active replication (or synchronous), all replicas are contacted inside a transaction to get their values updated. This replication mechanism can be implemented using a peer-to-peer approach, where all replicas have the same responsibilities and can be accessed by any client. Alternatively, a primary replica may have the responsibility of coordinating all other replicas.
- Passive replication: Passive replication (or asynchronous) model assumes a replica will propagate the updates to the other replicas outside the context of a transaction. As usual, replicas must receive all the updates.
- Operation-based versus State-based replication: There are two fundamental methods to propagate updates among replicas. In state-based replication, updates contain the full object state (or in optimised versions, a delta of the state). In operation-based, the updates contain the operations that modify the object and must be executed in all replicas. The size of an object is typically larger than the size of an operation. Transmitting the whole state of an object can introduce a

large overhead in message size. On the other hand, if the number of operations is high it can be better to transmit the whole state instead of all operations. Operation-based replication is used by commutative CRDTs and state-based replication is used by convergent CRDTs.

2.2.4. What are the use cases of CRDTs?

CRDTs can be used in situations that have to deal with multi-master scenarios, where multiple replicas can update an object, either at different times or on the exact same moment. There are CRDTs for counters, sets, trees and other types of objects. An example would be a rating system where users can give their opinion about a product. When a user gives a product a rating, the rating will be shared with all other replicas and the user receives all updates from the other replicas so that all the replicas converge to the same state, and all users sees the same rating for the product [26]. Some commercial implementations of CRDT are:

- Amazon uses CRDTs to keep their order cart in sync [23]
- League of Legends uses CRDTs for their in game chat system [19]
- TomTom uses CRDTs to manage navigation data and user actions and sync them with a server or other devices [11]

One of the biggest use cases of multi-master scenario is collaborative text-editors in which multiple people can edit one document at the same time. As of this moment a lot of research is done to see if CRDTs are a viable option to make commercial collaborative text-editors, but there are no commercial implementations as of yet.

2.3. CRDTs versus other collaborative editing algorithms

There are a lot of algorithms which enable collaborative editing; Woot, Logoot, key-locking, Jupiter, and SOCT 3/4 are some examples. Out of all the collaborative editing algorithms most fall into one of two categories; Operational Transformation (OT) or CRDT. During this project a CRDT will be implemented but it is important to elaborate on the differences between CRDT and OT, the other dominant form of implementing collaborative editing.

2.3.1. Operational Transformation (OT)

OT is a technique originally invented to support collaborative editing in text editors, while maintaining consistency across all users. Originally it only supported concurrent insertion and deletion of characters in text documents. After a few years just supporting deletion and insertion of characters was not enough when collaborative working on a document due to the rise of word processors. Word processors support more options than just adding and removing text, such as styling text, the use of images, tables, lists and many more features. For this to be made possible OT was improved with the update and undo operations alongside other features [25].

OT was proposed in 1989, and has been in development ever since. Multiple forms of OT were proposed of which some were proven wrong, because OT had too many edge cases. As of right now, the only versions of OT that had not been proven to be wrong are Jupiter, SOCT 3/4 and TTF. These can be categorised into two forms; server-based OTs and OT with a so called ”transform property 2”, this means that for every three concurrent modifications, no matter in which order they are executed, the result state is always the same. OTs with transform property two turned out to be so complex, that there are very few data structures that have a working implementation [9].

OT has been in development for more than twenty years now and supports a wide range of operations such as insert, delete, update and undo. It also supports many different applications such as HTML/XML and other tree-structured document editing, office productivity tools, 3D digital media design tools and many more [24].

As of right now, all commercial real-time collaborative editors use OT with a central server to handle multiple operations of different users and ensure everyone ends up in the same state. Some example of collaborative editors are Google Docs and Etherpad.

In OT, whenever a user executes an action, it is broken down into one or more operations. Every operation is processed in four steps [9, 17]:

- execute locally
- broadcast to other sites
- reception by other sites
- execution on other sites

These four steps can be categorized in two components [17]

- Integration, which takes care of reception, diffusion (breaking down) and execution of operations
- Transformation, merging of concurrent modification in order to execute them in serial order

Most practical OTs use a central server to decide an order of operation execution and to ensure that all sites get the same document state (The transformation step). By using a central server, the difficulty of transform property two gets eliminated, since operations are executed in the same order on every site [9].

2.3.2. Comparison between OT and CRDT

The difference between OT and CRDT relies mostly on the implementation of the chosen algorithm. An OT implementation with a central server is chosen since this is a proven concept and is currently

being used commercially.

Currently OT is the main algorithm used to implement collaborative editing. OT guarantees eventual consistency and gives a good user experience, both in speed as well as in usage. It is also very light on the client side, so it does not require a lot of resources from the users computer. The main disadvantage for OT, is the complexity of the transformation function. For every new operation that is added to the function, the function becomes quadratically larger. For N operations the functions requires N x N conflict resolutions, since for every combination of operations, a conflict resolution is needed. In Fonto's editor, clients can add an infinite amount of operations which could cause the transformation function to be infinitely large. One of the other disadvantages of OT is that it requires a central server, this can be a privacy threat since most central servers do not only store the operation, but also some personal information for history or other purposes. A central server is also a cost which needs to be accounted for. Also, most OTs do not scale well in cloud and peer-to-peer environments with dynamic groups where a lot of users leave and join quickly. The document users work on is also only saved on the server, so the users need a connection with the network to be able to open and work in the document. [2, 3, 9].

CRDT tries to tackle the downsides of OT. It is a decentralised algorithm that scales well with a lot of users and peer-to-peer environments. Since it does not require a central server, its implementation also has the potential to cost less and is less of a threat to privacy since no data is saved in a centralised point. With CRDT, it is also possible to work without an active connection. With CRDTs the users can work offline, make changes and eventually all replicas will synchronise when the connection is active again, while OT needs an active connection with the server. CRDTs can also potentially outperform OTs in both time as well as space-complexity [2]. OT is also at the limit of its potential, while CRDTs are relatively new and have a lot of room for improvements [9].

However, CRDTs also have downsides. There is a reason why they are not widely used yet. CRDTs only support two main types of data as of now, namely; plain text and arbitrary tree structured documents such as JSON and XML, while OT supports a wide range of data types. CRDTs also do not have an implementation which supports all types of operations, most CRDT implementations only support deletion and insertion. There are CRDTs that support update operations [2] and research is being done into supporting the undo operation [31]. As of this moment there is no CRDT implementation that supports all these types of operations.

CRDTs also have an inferior user experience compared to OTs. CRDTs have a harder time capturing the user intent due to the way how it breaks down operations in incredibly small pieces, such that the user intent is easily lost. OT is more complex, but it is better at capturing the users intention when applying operations. Currently available CRDTs can also give users a poor experience when applying operation such as splitting text or updating elements [9].

All in all, CRDTs currently provide a decentralised solution to collaborative editing, gives better offline support and potentially can outperform OT. CRDTs are cheaper than OT to implement and have a lot of room for improvement. OT supports a wider range of operations and data types, gives the user a better experience and is better at capturing the user intent, but has reached the limits of its potential.

2.3.3. Conclusion

OTs currently support more functionality, but are more complex and expensive than CRDTs. From the comparison between CRDT and OT, it can be concluded that CRDTs are a better fit for Fonto's purpose. They require a flexible solution which allows clients to add their own operations. In their case, clients would have to define the transformation function for the custom operations themselves, which should guarantee consistency. Fonto does not want to require clients to specify these functions themselves, since as mentioned before, they could become complex and very time consuming. Fonto also prefers a good support for offline working, so that clients can work on their documents while traveling and having limited access to internet. CRDTs have more potential to be more flexible and allow clients to add their own operations by breaking them down into smaller operations which can be merged. Fonto's editor only uses XML-format, therefore there is no need for a wider support of data types. CRDT can also be

faster than OT.

2.4. Commutative vs Convergent CRDT

CRDTs have two approaches, either commutative (operation-based) or convergent (state-based). This section is dedicated to explaining the differences between the two approaches and what the benefits and possible drawbacks are for each approach [21].

2.4.1. Commutative CRDT

The general idea of commutative CRDTs is that it only sends the operations done by every replica. So if one CRDT inserts a string at a certain position, this CRDT broadcasts the string and the position of this string to all other CRDTs. Each CRDT has its own local state and updates its state based on local operations and operations broadcasted to the CRDT.

A challenge that has to be solved when using commutative CRDTs, is that operations can arrive at other CRDTs in a different order [4]. To achieve that all CRDTs are in the same state, one should ensure that CRDTs can decide a total orders on all operations. This can be achieved for example by using clocks and timestamps when broadcasting operations. There are different algorithms for clocks available and each comes with its disadvantages. There are several different clocks which can be used [6]:

- Logical clocks: The causality relation is captured when using logical clocks. So there will be captured which event happened before the other event instead of sending a timestamp. A disadvantage here is, that all nodes need to receive all communication to decide the happen before relationship of all operations. One example implementation of a logical clock is vector clocks. Here every CRDT saves a list of timestamps of all other CRDTs.

- Physical time: Here, the clocks are synchronised using the Network Time Protocol [6]. One disadvantage when using this would be that there exists a chance that updates might occur at the same time and that CRDTs can therefore not decide on an order.

- TrueTime: TrueTime makes use of GPS clocks and atomic clocks to decide time. The disadvantage here is that to make use of this type of clock, the system needs to have an atomic clock and in the use case of the client, this would not be feasible.

- Hybrid Logical clocks: Hybrid Logical clocks is the type of clock that combines logical and physical clocks together [6]. Hybrid Logical Clocks guarantees that causal information is captured and CRDTs are not bounded on how fast they send new updates.

A disadvantage of using commutative CRDTs is that each operation will be broadcasted exactly once (efficient synchronization of state-based CRDTs). Therefore, it needs to be ensured that every message can be received by the client. Undelivered messages can lead to different states in multiple CRDTs and this is not the expected behavior in CRDTs. Another disadvantage is that it might be too complex to update the CRDT when many operations are sent [8]. When many operations are sent, it might become difficult for the CRDT to find the correct solution while making sure all CRDTs will find the same solution

However, commutative CRDTs have as advantage, that they do not send a lot of data on every update and that it will be easier to track history, because to track history of the CRDT, the CRDT only needs to save the operations. So commutative CRDTs will be able to perform updates and track history at a low cost.

2.4.2. Convergent CRDT

The general idea of convergent CRDTs is that all replicas occasionally send its own state to some other replica. Every update will then (in)directly be reached by every replica, where they can update their state to match with all the other replicas. The most important part of the convergent CRDT is the merge method, which essentially takes two corresponding replicas of the same logical entity, resolving any conflicts, and produce an updated state as an output [26]. The merge method must conform to three

properties: commutativity, associativity, and idempotency. The properties commutativity and associativity essentially points out that the order of the merge operations does not matter, since a state can be achieved through multiple different ways. The property idempotency states that it is not needed to care about potential duplicates.

For tracking updated objects, a vector clock or dotted version vectors can be used, using logical time rather than chronological time [8]. For distributed systems, a physical clock approach would not be suited since synchronizing the clocks is a very hard task to do and would require an additional server with synchronization algorithms [14]. Therefore, vector clocks would be a very suitable option. However, using vector clocks, the entire vector needs to be sent for every message, which means that synchronizing the vector clocks would be pretty expensive depending on the amount of processes [10]. There are several methods for reducing such overhead like the Singhal-Kshemkalyani's differential technique, which drastically reduces the space complexity from O(n^2) to O(n) [12].

The downside of convergent CRDTs is that state-based CRDTs can introduce a large overhead due to the size of the object [7]. However, this can be optimized by introducing deltas instead of sending the whole state [8].

2.4.3. Commutative CRDT versus Convergent CRDT

Whether to use commutative CRDT or convergent CRDT depends on the type of application and desired features. Both types have its advantages and limitations which both should be considered before making the decision which approach would be the most suited for that particular application. Commutative CRDTs send smaller messages, which gives less overhead and would make it easier to track history. However, the commutative CRDTs have to decide on a total order between all received operations.

Convergent CRDTs on the other hand would not consider the order of the messages important since the merge method must have the three properties: commutativity, associativity, and idempotency. These properties would make it harder to implement the CRDT. Aside from the hard-to-implement merge method, convergent CRDTs introduces a large overhead since the messages need to contain the entire state, without optimisations such as delta CRDTs. The downside of commutative CRDTs is that the operations are only broadcasted once, which means that if the communication channel is not reliable, the operation might not be received by the client properly, which might result in diverged replicas.

Finally, when adding new operations to the CRDT, it would be easier to implement this for commutative CRDTs instead of convergent CRDTs, since for commutative CRDTs, only an operation should be added to the CRDT, while for convergent CRDTs, the merge method should be changed to support the new operation while maintaining the idempotency property. Whether to use commutative CRDTs or convergent CRDTs for this project depends on what CRDTs are already available online and what features would be necessary for the use case of this project.

2.5. What known CRDTs are available?

This section will discuss different types of CRDTs. Use cases, advantages and disadvantages will be mentioned and finally there will be a conclusion on what CRDT will fit best for the use case of this project.

2.5.1. Tree-based CRDTs vs Set-based CRDTs

When discussing CRDTs, they are mainly defined as either set CRDTs or tree CRDTs. Set consists of elements at least contain a unique ID. This ID might be a tag or position ID. In tree-based CRDTs, every element is a node. There are one or more root nodes and every node has zero or more children. When using trees, operations may not only have effect on the element itself, but also on the children of the element. Therefore a removal of an element will also lead to a removal of the child, while something like this will not happen in set-based CRDTs.

2.5.2. G-Counter

Simple types of CRDTs are with counters and registers. A counter is a replicated integer supporting operations such as increment, decrement and value to query it. The G-counter, also known as the state-based increment-only counter, does not make use of the decrement operation. The idea of the G-counter is to make use of vectors of integers to track, where each replica has its own designated entry. The sum of all the integers in the vector would then be the value and the merge operation would then return the maximum of each entry. An important point which should be considered when using G-Counters, is that the payload is assumed to never overflow. This kind of CRDT would be useful for peer to peer applications which counts certain attributes, such as likes [20].

2.5.3. PN-Counter

The PN-Counter CRDT is a commutative CRDT which is practically the same as the G-Counter, but with an additional feature to decrement the counter. This can be achieved by using two G-Counters, one for increments, another one for decrements. With this additional vector, the value should not be the sum of both vectors, but the difference between the two G-Counters. The merge step merges both vectors into one. Such CRDT would be used in applications such as Skype for counting the numbers of logged in users. Due to asynchrony, the count may diverge temporarily from its true value, but will eventually converge and be exact [20].

2.5.4. Non-Negative-Counter

A non-negative counter is a counter that can only be positive. For instance, this can be used to count the remaining credits of an avatar in World of Warcraft. It is quite difficult for this counter to preserve the CRDT properties. If the current value is one, and two different users decrement the value at the exact same time, the value converges to minus one, which should not be possible in a non-negative counter. As counters in general are not really convenient for real-time collaborative editing, there is no real use case of non-negative counters for this graduation project [20].

2.5.5. Growth Only Set

A G-set is a CRDT where only elements can be added. One disadvantage of this CRDT, is that it cannot remove elements. On the other hand, code will be less complex, so if it does not have to deal with remove operations, a growth only set would be useful [20].

2.5.6. 2P-Set

A 2P-Set CRDT is a CRDT with the same functionality as a G-set CRDT, but with the remove functionality added [20]. The remove functionality is often implemented using tombstones. Using tombstones, instead of removing the element from the set, the CRDT will mark the element as invisible.

2.5.7. U-Set

U-Set is the simplified version of 2P-set under two assumptions. If elements are unique, a removed element will never be added again. Furthermore, if a downstream precondition ensures that $add(e)$ is delivered before $remove(e)$, there is no need to record removed elements, and the removed-set is

redundant [20].

2.5.8. Last-Writer-Wins-Element-Set (LWW)

The LWW CRDT is based on timestamps. All operations and elements in the CRDT get a timestamp assigned and all operations will be executed in the order of the timestamps [20]. When trying to update an element, the timestamp of the operation will be compared with the timestamp of the element. Only if the timestamp of the operation is more recent than the timestamp of the element itself, the operation will be executed.

So if there are two operations which will be executed sequentially, then after the first operation is executed, the timestamp of the element will be more recent than the timestamp of the second operation and therefore the second operation will not be executed.

2.5.9. Observed Remove Set

In an Observed Remove set, instead of giving elements unique timestamps or position identifiers, each element will get a unique tag. This tag is now used to add or remove elements and since each tag is unique, no conflicts should occur [1]. However, one constraint is that operations should be executed in a causal order at every CRDT.

2.5.10. PN-Set

The PN-Set algorithm is used for collaborative editing sets. Every element gets its own counter, initially zero. When Adding an element, its associated counter is incremented and when removing an element its counter is decremented. When an elements counter is positive, it is part of the set, when its counter is negative, it is not. [20, 29].

2.5.11. Sequence CRDTs

Sequence CRDTs are sequences, lists or ordered sets CRDTs. These CRDTs can be used to build a collaborative text-editor. Sequence CRDTs have a lot of different implementations such as treedoc, RGA, WOOT, Logoot, LSEQ and many more.

All these algorithms use a different type of implementation, but the general idea is the same. Each element gets a unique position in the document and this position is saved somewhere, when adding an element, it gains a unique position and is added to the data structure in which the positions are saved. When deleting an object, it is either deleted and the data structure is reorganized, or the position is kept in the data structure and the element is marked as deleted and/or invisible to keep the structure intact. Updating an element is not supported by all sequence CRDTs, most only support deletion and insertion of elements [2].

WOOT for example, uses a linear structure in which every element gets a unique id. When removing an element, the element is marked as deleted. It is not removed to keep the linear structure intact.

All these algorithms have the same characteristic in that they all guarantee that eventually every replica will end up in the same state. The difference of all the different type of sequence algorithms is mostly in flexibility, some versions only support deletion and insertion while others also support the update operations of elements. There is also a difference in time-complexity for these algorithms. The same holds true for space-complexity[2, 5].

2.6. Best CRDTs for our use case

Out of all the above mentioned CRDTs, sequence CRDTs would be the most suitable choice for our use case. All CRDTs based on counters are designed for the use case of likes or views for example. They are useful when a certain object receives multiple updates in a short amount of time. When dealing with text-editors, one can deal with a lot of objects, each character can be seen as a single object, and the text document is the a sequence or set of objects. The issue with set CRDTs is that they do not account for order. In a text document, there is a certain structure that the document must adhere to. Set CRDTs only check whether an element should be in the set or not and do not have any complexity that guarantees the order of the set, which is required to guarantee in a text document.

Sequence CRDTs have many different type of implementations of which all potentially could fit our use case. In the next chapter all the versions of sequence CRDTs will be analysed and evaluated. Both theoretical models as well as practical implementations of the different types will be researched. Based on this in-depth research on sequence CRDTs a type of implementation will be chosen and possibly extended with extra functionality to supplement extra functionality required for our use case.

2.7. Conclusion Research Report

In this research report, the following research questions have been discussed:

1. Why CRDT over other collaborative editing algorithms?
2. What types of CRDTs are there?
3. Which CRDTs are the best fit for Fonto's purpose?

Regarding the first research question, it can be concluded that for Fonto's use case, CRDTs are indeed the most suitable option. Collaborative editing currently knows two approaches, namely CRDTs and OTs. Although currently OTs support more functionalities, CRDTs are far more flexible, which would fit Fonto's current system better than OTs. OT's transformation function also grows quadratically for every new operation added. Since Fonto allows infinitely many operation from many different clients, it would become too complex for the editor.

When discussing different types of CRDTs, it can be concluded that there are currently two known types of CRDTs, namely commutative and convergent CRDTs. Commutative CRDTs are operation-based and convergent CRDTs are state-based. The difference is that commutative CRDTs send operations to other replicas while convergent CRDTs send their state to other replicas. When talking about the inner structure of a CRDT, sets and trees can be used for both types. These structures can be used to implement different CRDTs such as G-Set, 2P-Set and sequence CRDTs.

Out of all different CRDTs mentioned in this report, it should be decided which CRDT is the best fit for Fonto's purpose. First, it should be decided whether a commutative or convergent CRDT should be used. It can be concluded that commutative CRDTs would fit best for Fonto's purpose since commutative CRDTs offer easier to update the current state and it would be easier to add different operations to the CRDT in comparison with convergent CRDTs. Convergent CRDTs also require more overhead for each operation in comparison with commutative CRDTs. Sequence CRDTs would fit the best for this project, since Fonto uses a XML-based editor and sequence CRDTs are the best to use for collaborative editing.

3

Requirements and problem analysis

3.1. Requirements

The goal of this project is to find a solution for Fonto to add support for real-time collaborative editing in their block-based editor. Based upon research done in chapter 2 and the project description, requirements have been formulated. These requirements were formulated with the idea that if these requirements are met, Fonto would have the answer to their question whether it is possible to use CRDTs to add real-time collaborative editing to their editor. The requirements are split up in mandatory and optional requirements. The mandatory requirements form the basics, which show that the CRDT is able to be support block-based text editing, while the optional requirements are extra functionalities to work on if there is time left.

Mandatory

1. Insert text
   1.1. Text can be inserted into the document
2. Delete text
   2.1. Text can be deleted from a document
3. Move text on block level
   3.1. Block-level text (e.g. a paragraph) can be moved in the document
4. Splitting text
   4.1. Block-level text (e.g. a paragraph) can be split into multiple blocks
5. Merge text
   5.1. Multiple block-level texts (paragraphs) can be merged into one block
6. Convergent
   6.1. Every replica needs to converge to the same state after all operations have been processed
7. Offline support
   7.1. Operations can be saved locally and sent through the network to other replicas later while still converging
8. Extendable and modifiable
   8.1. It should be easy to add new operations or functionality to the CRDT

8.2. It should be easy to modify operations of the CRDT

8.3. It should be easy to test whether such modifications still ensure convergence

9. The CRDT supports a minimum of two concurrent editors for research purposes

10. The CRDT can be given an initial state

Optional

11. XML Data Model

11.1. Extend the CRDT to implement other aspects of the XML data model, such as attributes

12. Optimisations

12.1. Automatically reorder operations into a sequence that minimizes the number of times the "current author" switches

12.2. Insert or delete large amounts of text

13. XML Schema

13.1. Investigate approaches for dealing with restrictions placed on the document

14. User Intent

14.1. When a user is typing in a block and the block is moved, the user cursor will be moved to the new position of the block

14.2. When two authors write at the same time in the same position, the edits should be inserted as two separate blocks of text instead of having both edits mixed within each other

14.3. When text is inserted or removed, the cursor of other authors should stay at the same position relative to the surrounding text, not in terms of a number of characters

Two other well-known operations within editors, are undo and redo. These two operations are however out of scope for this project due to the complexity of these operations.

3.2. Problem Analysis

Designing and developing a CRDT from scratch would either take too much time or would have many imperfections at the end of this project. As suggested by our client and (former) coach, looking into already existing CRDTs and using the CRDT as a base to extend the CRDT to meet the requirements formed by the client is the go-to method. From the research report, it was quite obvious that sequence CRDTs will be used for this project. Currently many forms of sequence CRDTs are known, such as Logoot, WOOT, and RGA. Looking into the repositories on GitHub, a selection of already-implemented CRDTs are chosen to be looked further into. These types of CRDTs and its implementation will be discussed in the following sections.

3.2.1. Types of sequence CRDTs

As mentioned in the research report, there exists many types of CRDTs nowadays. For this problem, it would be ideal to find a CRDT with the purpose of editing text. The CRDTs that are found on GitHub and interesting enough are of the following types: Logoot, WOOT, RGA, YATA, and TreeDoc.

Logoot is essentially an operation-based CRDT (commutative). This means that the operations done by different replicas should somehow have a total order such that the order of operations are the same for every replica. The Logoot model is composed by so-called 'lines', which consists out of a position id and the content (line). The order of the operations should be decided by the values of the id and the site in which the operation is performed on [27]. Out of the ordinary, the implementations found for this model contains an additional Leeds Sleep Evaluation Questionnaire (LSEQ) algorithm and a tree structure, for faster character lookup. The LSEQ algorithm is used for allocating nodes at the most efficient place in the tree [13]. This implementation however is character-based instead of line-based.

WOOT is a fairly easy approach. Each operation directly sends the orderings since this is known, unlike Logoot. Each insert between two letters will include this in the broadcasting message, which makes it easier for each replica to find the correct spot. When one or more of these letters are deleted, it will be marked as invisible (tombstones), so finding the correct spot will still be fairly easy. When multiple users insert text at the same location, a topological order is adhered such that all the replicas will converge to the same state [16].

RGA, also known as Replicated Growable Array, is essentially a sequence, implemented with a linked list. An element has an identifier, which corresponds to a timestamp, which is assumed to be unique. When conflicts happen, two users insert at the same position, the operation with a higher timestamp occurs to the left of the operation with the lower timestamp. RGA makes use of tombstones upon deletion, in order to accommodate a concurrent add operation [22].

YATA uses graphs, lists, and objects for collaborative editing. The core idea of YATA is enforcing a total order on the shared data types. An insertion contains a left and right value, such that it knows where to insert the newly added character. When conflicts occur upon insertion, the insertion with the smaller creator id will be inserted first. An extra constraint is that no crossing connections are allowed between conflicting insertions. The left and right value of the operations should not cross each other. The two permitted operations are shown in figure 3.1 [15].

Figure 3.1: No line crossing

TreeDoc, as the name implies, uses a binary tree to use paths in an efficient way. The characters which comes before a certain character will be placed on the left with a 0, whilst the characters which comes after will be placed on the right with a 1. The binary tree identification structure is insufficient for concurrent edits. When there are two insertions at the same position, internal mini-nodes will be created. The node containing the mini-nodes is called the major node. To keep the tree balanced, an *explode and flatten* method can be used, to keep the path as efficient as possible [18].

3.2.2. Time and Space Analysis

In this subsection there will be a time and space analysis of the following implementations: Woot, Logoot, YATA and RGA. No information on time and space complexity has been found for TreeDoc, therefore it is excluded in this section. For a user to comfortably notice and respond to changes, the time it takes for remote and local operations to be processed is 50ms [2]. Therefore a CRDT should be able to process an operation in a certain amount of time. For space, it is desired to limit the amount of space a CRDT uses. For every operation the CRDT saves a piece of metadata to be able to communicate to every replica what operations to execute so that all replicas converge. Depending on the implementation both space and time complexity can greatly vary. The time and space analysis can help to substantiate a choice for a CRDT.

All CRDTs have different types of implementation, therefore space usage and execution time is dependent on different variables, here is a list of variables important for space and time for different algorithms. [2, 15]

- $H =$ The number of operations that has affected the document, affects all CRDTs when comparing worst cases
- $c =$ The amount of operations that are executed at the same time can impact RGA
- $n =$ The size of the document, has an influence on RGA and Logoot (non deleted characters)
- $N =$ The total number of inserted characters, both non-deleted as well as deleted characters (tombstones) impacts RGA, WOOT, and YATA, since these CRDTs use tombstones
- $k =$ The average size of the identifier is of importance for the space usage of Logoot, which gives every character a unique identifier
- $t = N - n$, The amount of tombstones
- $d = (t + c)/n$ The average number of elements found between successive document elements (tombstones included), important for WOOT implementations.

Time complexity

For time complexity, the worst and average case have been researched. For the worst case scenarios, all CRDTs are reliant on the number of operations that have affected a document. In average case scenarios, the complexity is more reliant on the implementation of the CRDT and how it scales with the type of executed operations.

CRDT	Local Insert	Local Delete	Remote Insert	Remote Delete
WOOT	$O(H^3)$	$O(H)$	$O(H^3)$	$O(H)$
WOOTO	$O(H^2)$	$O(H)$	$O(H^2)$	$O(H)$
Logoot	$O(H)$	$O(1)$	$O(H * log(H))$	$O(H * log(H))$
RGA	$O(H)$	$O(H)$	$O(H)$	$O(log(H))$
YATA	$O(log(H))$	$O(log(H))$	$O(H^2)$	$O(log(H))$

Table 3.1: Worst case time complexity of certain CRDT implementations [15]

However, due to the versatile use of collaborative real-time editing, worst case scenario almost never represent the actual achieved complexity. A more precise representation, which also is better at showing the bottlenecks for most CRDTs, is represented by the average time complexity. There is no information available for the average time complexity of YATA, therefore it is not in the table.

CRDT	Local Insert	Local Delete	Remote Insert	Remote Delete
WOOT	$O(N * d^2)$	$O(N)$	$O(N * d^2)$	$O(N)$
WOOTO	$O(N * d^2)$	$O(N)$	$O(N + d^2)$	$O(H)$
Logoot	$O(k)$	$O(1)$	$O(k * log(n))$	$O(k * log(n))$
RGA	$O(N)$	$O(N)$	$O(1 + c/n)$	$O(1)$

Table 3.2: Average case time complexity of certain CRDT implementations [2]

From these tables it can derived that the time complexity of both WOOT implementations are bad in both the worst as well as the average case. WOOT is reliant on the amount of inserted characters and the distance between successive document elements, so the more deletion the user does, the worse WOOT is. Logoot has decent worst case time complexity and its average time complexity is reliant on its identifier size. So if it is possible to keep the identifier small, Logoot is a good candidate. RGA worst-case is decent and it is mostly reliant on the amount of inserted characters (including deleted). YATA has a decent worst case time complexity, together with Logoot and RGA. Since YATA also uses tombstones it can be concluded that it is also mostly reliant on the amount of inserted characters (including tombstones).

Space complexity
For space complexity the most important part is what the CRDT needs to save. If a CRDT uses tombstones, then every tombstone requires a certain amount of space, even if its character is not in the document anymore. Therefore the space complexity of all WOOT implementations, YATA, and RGA are $O(N)$ on average and in worst case $O(H)$. Logoot does not require tombstones, but requires some extra metadata per character. So the space complexity of Logoot is $O(n * k)$ and in worst case (H^2).

Algorithm	Worst Space Complexity	Average Space Complexity
WOOT-WOOTO	$O(H)$	$O(N)$
Logoot	$O(H^2)$	$O(n * k)$
RGA	$O(H)$	$O(N)$
YATA	$O(H)$	$O(N)$

Table 3.3: Worst and average case space complexity of certain CRDT implementations [2]

It is possible to implement a garbage collector to remove tombstones. A possible solution for a garbage collector would be to use a vector clock to verify whether all users are up-to-date with previous operations. This would imply that when a delete operation has been received by all replicas, the tombstone can be removed to improve the space complexity of tombstone based CRDTs. It is however, hard to determine whether all replicas have received a remove operation and thus hard to implement [15].

Overview time and space complexity
To conclude this section, WOOT and all its implementations have bad worst and average time complexities. Logoot, RGA and YATA have good worst case time complexities, where each CRDT excels in certain operations. No results were found for YATA on average time complexities, but since it also uses tombstones it will most likely be close to RGA.

For space complexity, Logoots worst case is $O(H^2)$, which is worse than the others, but since Logoot does not use tombstones, it scales better when a lot of deletions are executed. Therefore it can be better than WOOT, RGA and YATA when comparing average space complexities, but then it depends on the size of the identifier used by Logoot.

3.2.3. Degree of implementation difficulty
When searching for different already-implemented CRDTs, implementations of WOOT, Logoot, YATA and RGA have been found. When deciding whether a CRDT is suitable to use as a starting point, these are the requirements used as criteria:

- Is the CRDT easy modifiable and extendable? (8)
- Does the CRDT always converge? (6)
- Is the CRDT operation-based?
- Can the CRDT insert text? (1)
- Can the CRDT delete text? (2)

The implementation of YATA is already very complete regarding functionality. Regarding the requirements, this CRDT supports insertions and deletion and the provided editor has decent user intent. This CRDT also converged on all different manual tests performed when testing this CRDT. This CRDT however, has a very large code base and looking at the code, it would be very difficult and very time consuming to extend and modify this code base. Since the code base is not easy extendable and modifiable. This implementation would not be very useful for this project, although it has all other criteria.

The implementation of RGA worked decent and was simple. However, it is a state-based CRDT and therefore not useful for this project.

The implementation of Treedoc did not work properly and contained several bugs. Since the implementation contained bugs, it would not be suitable to use for this project since solving these bugs may take more time than actually implementing the desired functionalities.

The implementation of WOOT was decent, but WOOT has a lot of overhead and uses a lot of memory. The developer of this WOOT implementation also developed a Logoot implementation and the developer stated that Logoot has a much better memory performance in addition to some other small improvements. Both implementations are very bare boned and therefore easy to extend and modify. Logoot is preferred over WOOT, because of the lower memory overhead. The Logoot implementation has more functionalities that are required in comparison to the WOOT implementation. Logoot already has the insert and delete functionality and is, as mentioned before, easy to extend and modify. Finally, When manually tested, no bugs has been found and the CRDT always converged.

3.2.4. Chosen CRDT

Based on the found implementations, it had to be decided which implementation would be used to build on for the proof of concept prototype. The implementation that is going to be used is the Logoot implementation of Thomas Mullen (https://github.com/t-mullen/logoot-crdt). This Logoot implementation passed all criteria as mentioned in the previous section. This CRDT supports insert and delete functionality and is easy to modify and extend. Additionally, the implementation is character-based which is beneficial for this project, since it is easier to insert nodes in between characters. If it is line-based and an insertion is performed somewhere in the line, the line needs to be split, and an additional identifier needs to be created. Finally, when manual testing the implementation of Logoot, no situation has been found where the CRDT did not converge. The following mandatory requirements are still missing in the CRDT:

- Move text on block level (3)
- Merge text on block level (5)
- Split text on block level (4)
- Offline support while still converging (7)

In the following sections, it will be discussed how these requirements will be implemented.

4

Design

In this chapter, the design of the CRDT will be described. The content of this chapter will elaborate on the already existing Logoot implementation and the changes implemented in its design. Next, the design of the features required by Fonto will be explained, and how the Logoot instance is extended with extra functionality. Since this report is a proof of concept, chapter 4 and chapter 5 will discuss which concepts of CRDTs were used to achieve the requirements and how these have been implemented in the research prototype. Chapter 6 will describe how it has been proved that the described concept indeed meets the requirements.

4.1. Basic Logoot implementation

The chosen Logoot implementation uses a tree structure to support the insertion and deletion of text. Initialising a Logoot instance creates a basic tree consisting out of one root node with two child nodes, which serves as a begin and end node of the document. The child nodes of the starting root node always have the Id 0 and 256. An example of such tree is shown in figure 4.1. Originally, every node has an identifier, which consists out of an Id, site and a clock. The Id is an integer that is mainly used for the position. The site is a string used to identify which user performed the operation. The clock is an integer which represents the count of operations performed by that particular user. All three components are used for determining the placement for the node. So when the Ids are the same, the site is the next value to compare with. Similarly, when the sites are identical, the clock will be compared. Since the clock is the operation count, if all three are the same, it means that it is the same operation. However, to simplify this in this report, when an Id is mentioned, the Id of the identifier of that particular node is meant.

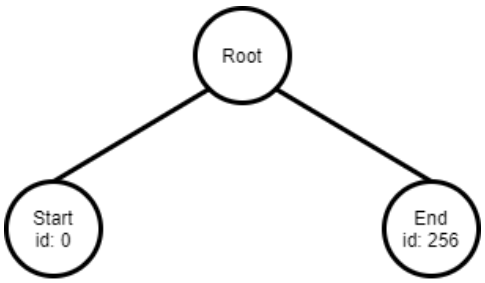


Figure 4.1: Basic tree

4.1.1. Insert and delete nodes

Every insert operation must have an index, such that the CRDT knows where to insert the value. Based on the index (depth-first search), a position is generated using the identifiers of the neighbour nodes. The neighbour nodes are the current node at the given index and the node at the given $index + 1$. The index will firstly be checked whether it is out of bounds. When this is the case, the index of the

last node will be used, such that the insertion will be performed at the end of the block. Once the two neighbour nodes have been found, the CRDT will generate a random integer for the Id, which is between the two Ids of the neighbour nodes. Cases can occur when there exists no integer between the two Ids of the neighbour nodes. In these cases, the node would be inserted into the tree as a child of the left neighbour node and would have a randomised Id (figure 4.2). When a node is inserted in the CRDT from one client, it emits a message containing the path (sequence of identifiers) and the content of the node. In the case of figure 4.2, the users emits the path [255, 58] (simplified to only Ids) to the other CRDTs in the network. The receiving client receives this message and builds on the exact same location a node with the same content, if there exists no node on that path. When the exact same location already contains a node, where the identifiers are exactly the same, the insert operation will be aborted. Since the clock is the operation count, an operation can only be aborted if that operation is already executed. This means that the Id, site and clock are exactly the same as the incoming operation.

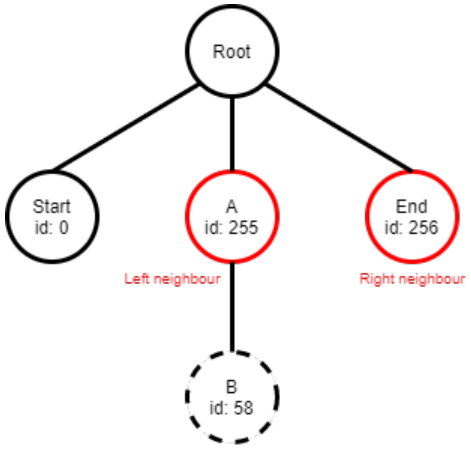


Figure 4.2: Inserting B after A with no integer existing between the identifiers of the neighbour nodes

Similar to the insert operation, the delete operation must have an index. The index will lead to a node in depth-first manner, which should be removed from the tree. However, when this node contains child nodes, the node is set to empty with an 'empty' attribute set to true and this node will now basically function as a tombstone. When the node does not have children, it can directly be deleted from the tree. The CRDT will recursively go over the parent nodes of this node to check if the parent can be deleted from the tree as well. If removing the node from the tree results in a tombstone parent node not having any children anymore, this parent node will also be removed from the tree. Just like the insert operation, a message is sent containing the path of the deleted node, such that the receiving client can delete the exact same node from the tree.

As noticed, both the insert and the delete operations will send messages to other CRDTs with their respective path of the inserted or deleted node. This means that all the CRDTs will contain the exact same tree structure with the same content since all nodes will be inserted or deleted on the same path. Because all CRDTs contain the same tree structure, convergence is guaranteed.

4.2. Approaches for block operations

For the block operations, two approaches had been thought of. One approach (approach one) would insert special block characters at the start and end of a group of characters to define a block. The order of these blocks would then be managed by a new CRDT, specially designed for handling the order of the blocks. The other approach (approach two) would introduce 'block' nodes. The idea is to make every block node have its own Logoot instance, such that inserting characters in the block would be easy. Considering both approaches, the decision is mainly based on how the block operations would be integrated. In approach one, moving and splitting blocks would be easy to implement. Merging blocks on the other hand, would require many special adjustments to the Logoot since the blocks might have been moved before. Aside from implementing these operations, an additional CRDT should be

created to manage the order of the blocks for approach one. For approach two, the block operations would be relatively easy to implement. The main issue with approach two is that splitting a block would need to keep the tree intact, since each insertion emits a path to the other clients, the path would be lost and therefore not executed properly. Based on these criteria and considering the given time span, approach two had been chosen. Creating a new CRDT for managing the blocks would consume too much time. Even when the new CRDT is completed, the time left for adjusting the Logoot to function properly for the split functionality would be very limited. Aside from time management, approach two would have better scalability for Fonto's purpose. For example the XML tags and attributes for certain texts, a block node can be created with the XML attributes in the class itself.

4.3. Detailed design

Approach two starts with a Logoot instance, which will be referred to as the main Logoot. This Logoot has the same basic tree structure as figure 4.1. This Logoot instance serves as the communication channel for the all other replicas which are connected. All messages would be handled with this Logoot instance. Since the document would consist out of blocks with text, only block nodes are inserted in this Logoot. The block nodes are inserted in exactly the same fashion as the basic Logoot instance for characters, but rather than characters, block nodes are inserted. Each block node, additionally, contains a special blockId and its own Logoot instance, where the blockId is independent of the main Logoot structure. The blockId is a unique randomised string of length five, which allows the Logoot to search on on blockId instead of using the identifier. The idea behind the Logoot instance in the block node is to insert text into blocks easily, since the code for Logoot already exists and had been tested for correctness when inserting and deleting text. Since each block of text should be in a block, upon emitting the message, the blockId can be included, which allows the text to be inserted in the Logoot instance of that particular block. Similar to the original Logoot, when a user inserts text, an emit message is sent with the path and the blockId. The blockId will then be used to search the corresponding block and the character node would then be inserted into the Logoot of that particular block node.

Block operations

As for the block operations (moving, merging, and splitting), new approaches had to be thought of in order to guarantee convergence, since CRDTs should also support operations which are executed offline. This drastically boosts the complexity of the code, since the order of the operations can differ per replica. Every operation should be able to converge when order of execution is different. When no delay (offline operations) is taken into account, these block operations could easily be implemented just by using the insert and delete operations. As moving blocks could be performed by inserting a new block, deleting the original block, and changing the blockId of the newly created block into the original blockId. Merging blocks A and B could be performed by taking the value of block B and inserting it at the end of block A and deleting block B afterwards. Splitting block A could then be performed by creating a new block B and insert part of A into B whilst deleting that part in A. These methods do not work for offline operations, since the order in which the operations are performed needs to be the same for some operations. For example, insert operations are performed before someone split a block, but that someone worked offline, so the order is flipped for this user. This means that at one replica the insert operations are split correctly, while the other replica inserts the insertion in the wrong block (figure 4.3). Since these approaches do not support offline operations, a new approach had to be thought of to support these operations.

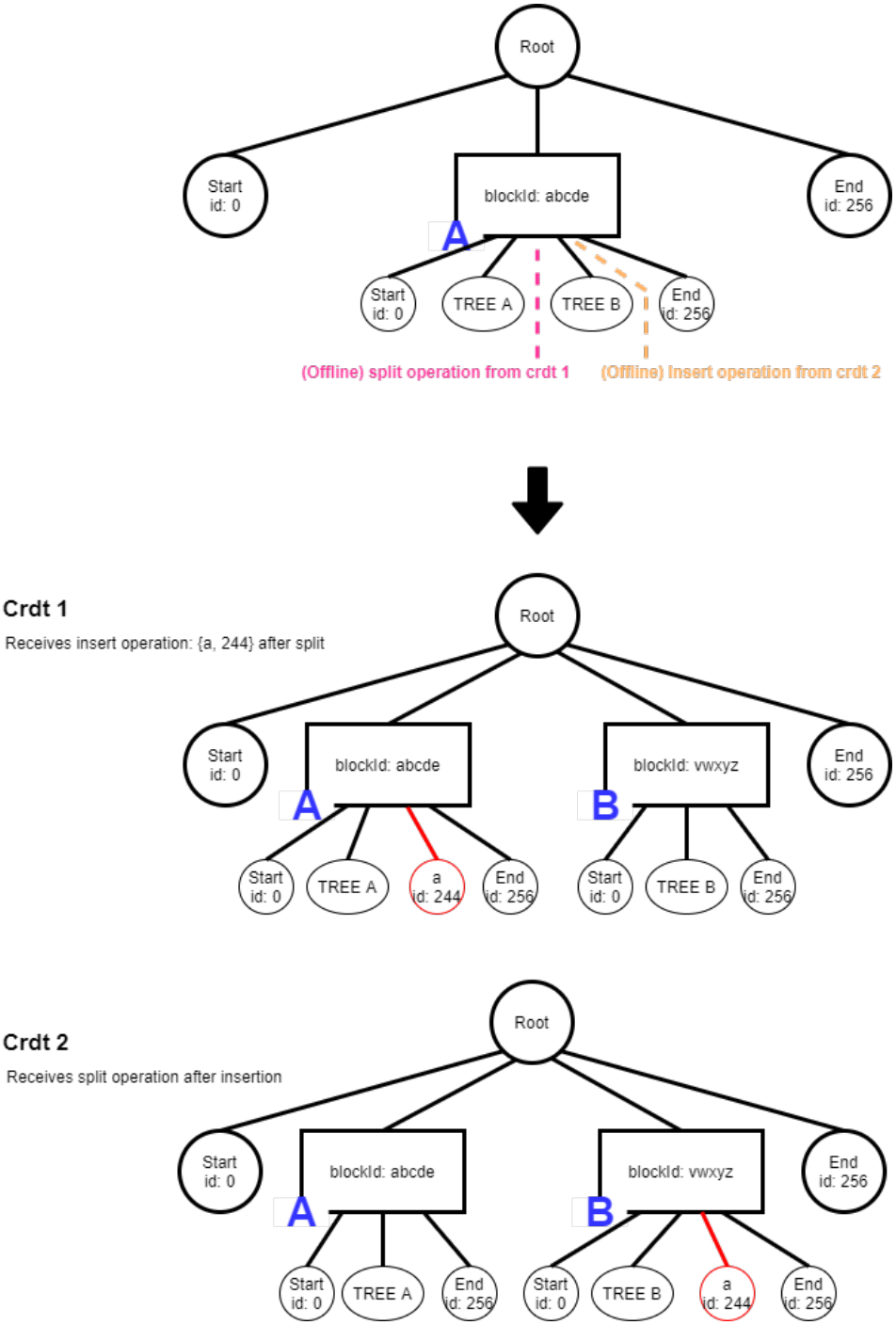


Figure 4.3: Two CRDTs diverge when both CRDTs are working offline using the split and insert operation

Move block operation

The approach for moving a block is as following. Moving block A requires an index, which is used for determining the position. When the move operation is called, it generates a new block B at that particular position. Block B has a different blockId compared to block A. When block B is generated, the Logoot of block A is transferred to block B, such that all the content is moved safely. Afterwards, block A is deleted and block B will take over the blockId of block A (figure 4.4). Since the inner Logoot stays the same, the CRDTs converges even after offline insert or delete operations since all insert and delete operations require the blockId. This means that when the offline operations arrives, the block would first be searched and the operation would then be performed on the found block.

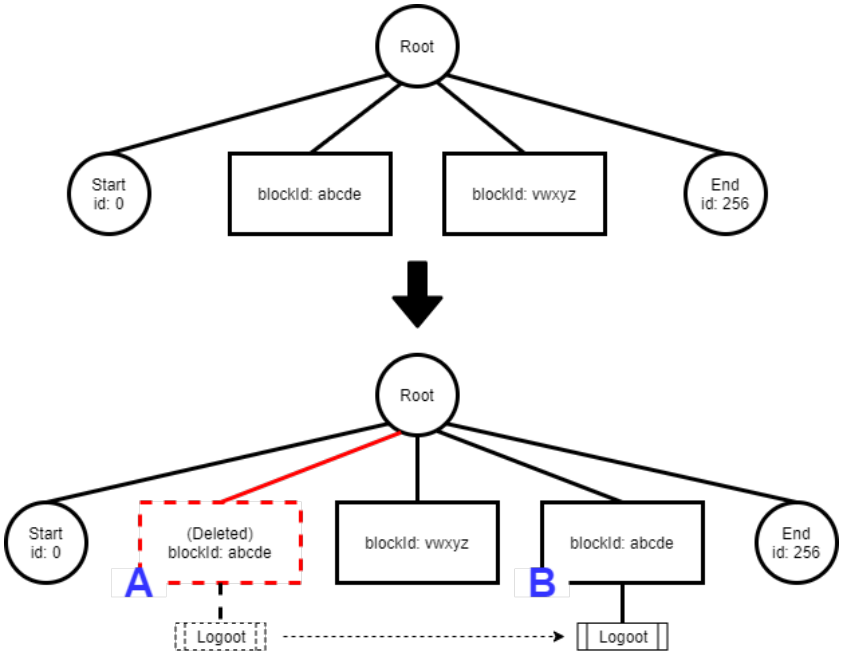


Figure 4.4: Moving block with blockId: abcde after block with blockId: vwxyz

Merge blocks operation

For the merge blocks operation, three approaches had been thought of. The first approach works for certain cases, which do not include two or more CRDTs that are merging the same blocks in different order. The second approach has fixed this issue, but has complications in combination with the split operation. In order to make it easier to function with the split operation, a third approach had been thought of. All these approaches will be elaborated in this sub-section and a more detailed description and flaws of each approach will be given in the next chapter.

The essence of the first approach for the merge operation is to set the status of the merged block to 'merged' and move its tree to block A. The CRDT adds two 'additional' nodes to the root node of the tree in block A and appends the original tree structure of A and B to these two nodes. To make this more clear, an example will be given. In this example, block A and block B will be merged. Block A will generate two additional nodes, attached to the root of block A and moves the original content to one of the two newly appended nodes as children. The other additional node will be used to transfer the tree structure of block B. Since the tree structure stays the same, for block A and block B, offline insertions before the merge operations can be redirected by appending the corresponding additional node to the path of the operation (figure 4.5). However, this approach could lead to some complex states, where two offline replicas merge three blocks, both in different orders. The aforementioned approach could not solve this problem since the additional nodes are permanent, which results in different states (diverges). Thus, a different approach had been thought of, which uses reference nodes.

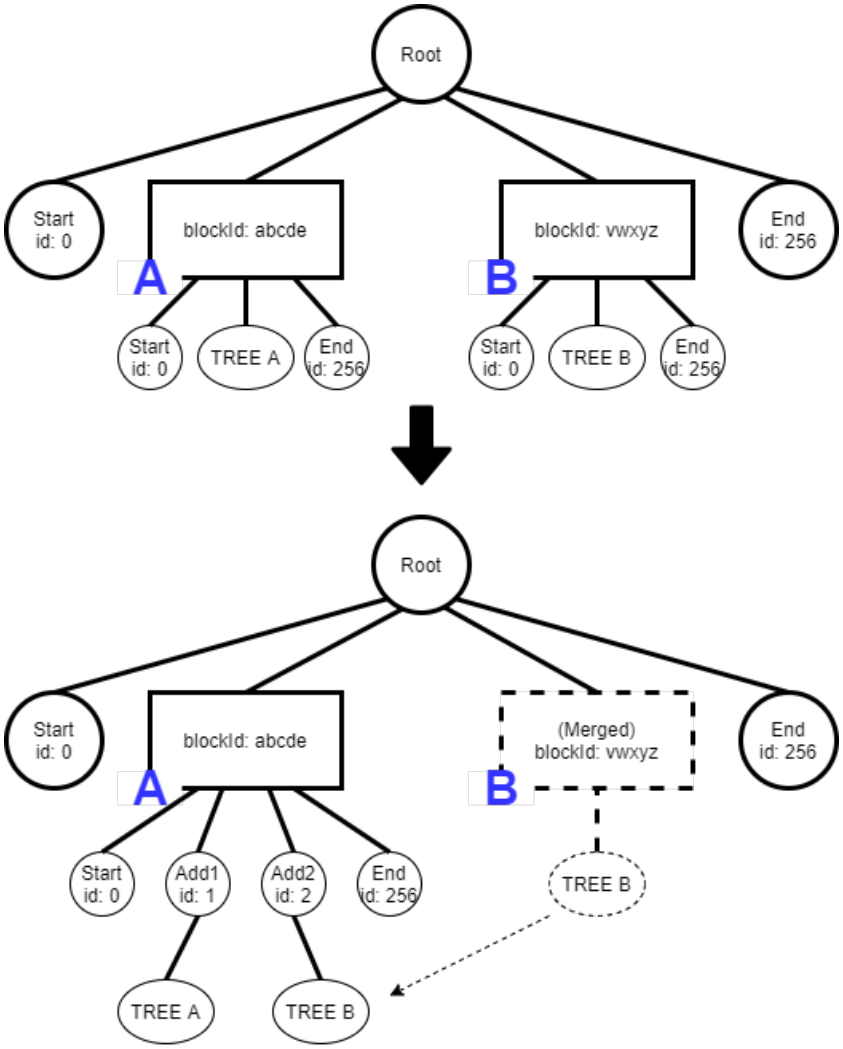


Figure 4.5: Merging block B into block A with first approach using sub-nodes

The second approach used for merging blocks is as following. Merging blocks A and B only inserts a 'MergeNode' at the end of block A. The status of block B will be set to merged, which practically makes it invisible for the user. The MergeNode has a reference to block B. This approach is shown in figure 4.6. This method does not actually merge the two blocks, but rather simulate the block as if they are one block. For this approach to work, the length of block A needs to be taken into account. Since the content is not in block A, but rather simulated, the maximum length of the index (which is used for inserting characters) of that block is incorrect, because it does not take the content of block B into account. This means that the usage of the index should change for this problem to be solved.

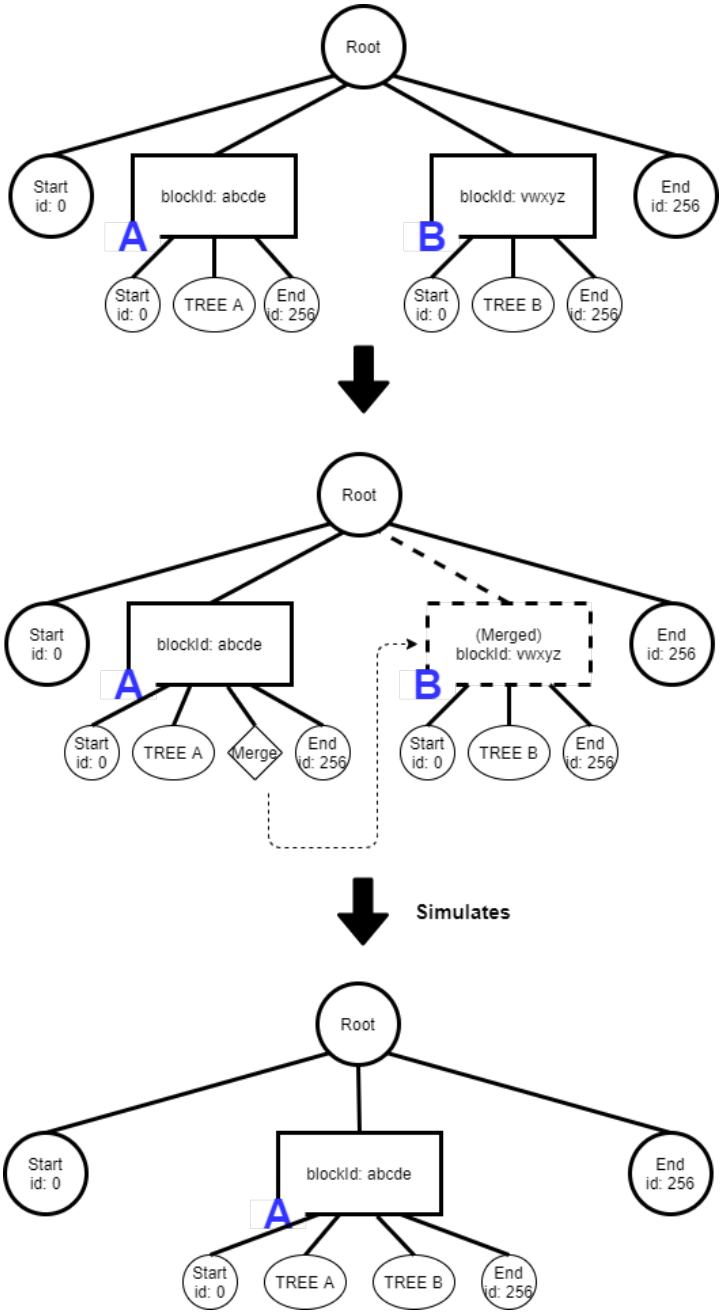


Figure 4.6: Merging block B into block A final approach using references

For the final approach, which is very similar to the second approach, the end node of block A will be changed into a merge node. So when block A and block B are merged, block A will change its end node to a merge node, which contains a reference to block B. This guarantees the state of the block where the merge node will be put at the end of the block. Complications about the second approach was that when a split operation is performed in block B, the content of block B and block A after the merge node should be moved whilst keeping the tree intact (figure 4.7). By guaranteeing that the merge node is at the end of the block, the content after the split node in block A does not exist. This makes the split easier by just splitting in the merged block B. An example of a valid state of a merged block is given in figure 4.8.

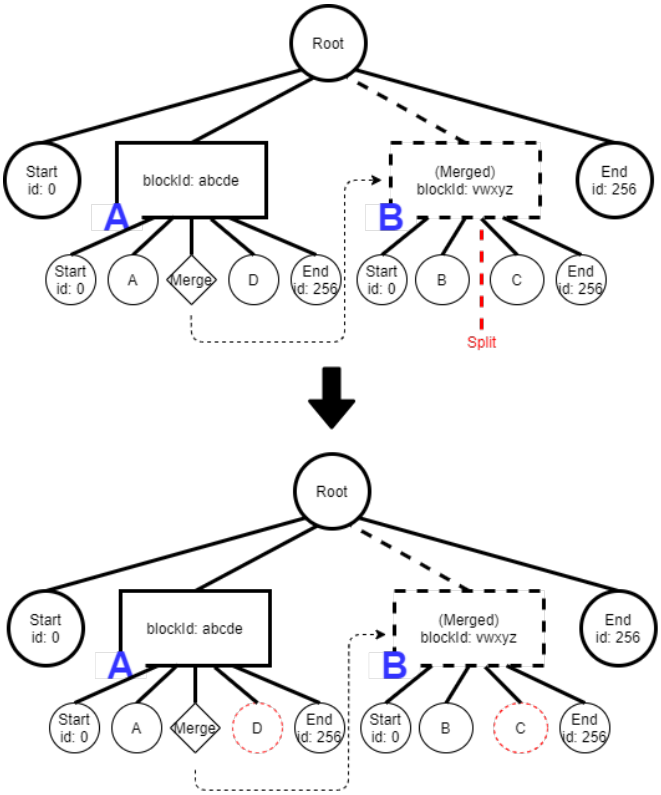


Figure 4.7: Splitting in a merged block with content 'ABCD' on B and C, will need to move node C in block B and node B in block A to a new block

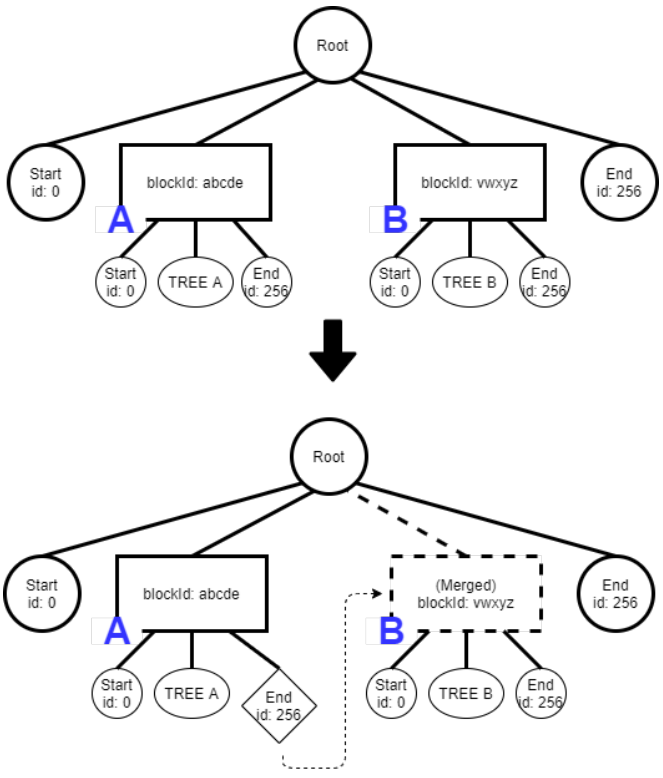


Figure 4.8: A valid state of the blocks when block A and block B are merged

Split block operation

The approach for splitting a block is as following. Splitting a block requires an index, such that the CRDT can determine where to split the block. The block (A) will insert a 'SplitNode' according to the index, just like the normal insertion, which represents the split of the block. A block B, with a new blockId, will be created right next to block A and copies the exact same tree structure of block A into block B. For block A, the blockId will stay the same and all nodes on the right side of the SplitNode will be removed, whilst for block B, all nodes on the left side, including the SplitNode, will be removed (figure 4.9). When offline operations happen at the same time as the split operation, it might happen to be an operation which should be performed on the right side of the SplitNode in block A. In these cases, the operation should be delegated to block B. Since every operation is sent individually, it can easily be checked whether it should be delegated since the first-left node should be the SplitNode in these cases. A more detailed implementation will be described in the next chapter.

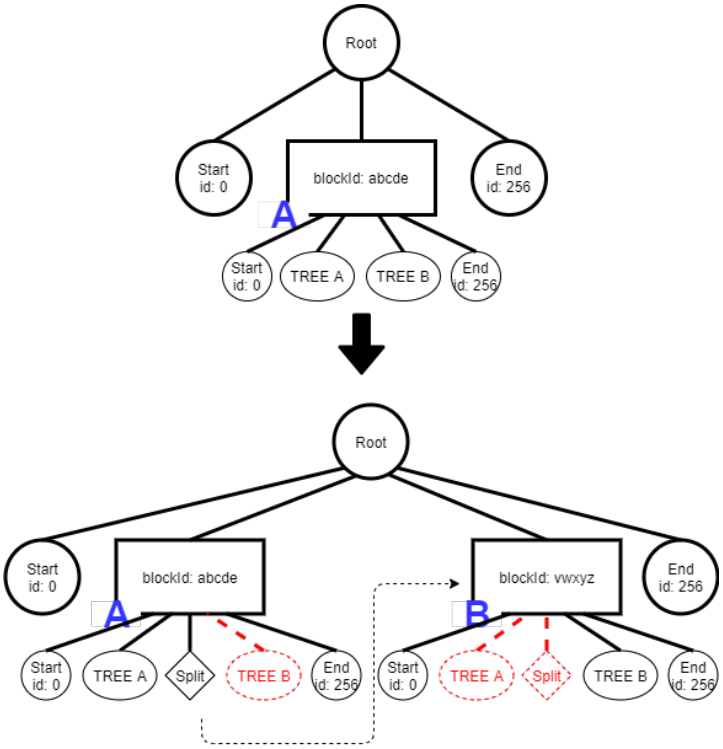


Figure 4.9: Splitting block A using references

In this chapter, the design was introduced of how the operations would be implemented. This should give a global view of what the CRDT does and which methods were applied to achieve this. In the upcoming chapter, a more detailed description about how these methods were implemented and the issues that arose during the implementation will be given.

5 Implementation

The previous chapter gave a more high-level explanation of the concepts that were used. This chapter will discuss the implementation of these concepts and will therefore give a more detailed explanation about the concepts. This chapter will also discuss the issues that arose during the implementation and how it has been verified that this implementation is correct.

5.1. Validating the basic implementation

Before writing the code for new functionalities such as inserting, moving and splitting blocks, the functionality of inserting and deleting plain text had to be validated beforehand to be sure that it actually works as expected. Tests were written for the following functionalities:

1. Insert text
2. Delete text
3. Set CRDT initial state
4. Replace text
5. Offline support
6. Network delays

When writing tests, the Mocha test framework and Chai assertion library have been used. Since the most important property of CRDTs is convergence, every test case has at least an assertion which tests if the state of different CRDTs are still equal after several operations. If it is clear what the expected value of the CRDT should be, an assertion to test the result value to the expected value has been added as well. It turned out that when testing network delays and offline support, the result after performing the same set of operations can differ when executed multiple times. For example when two CRDTs insert one character offline, the final order of these two characters will be based on the identifier and since the Id of the identifier is randomly generated, the order between these two characters might differ when executed multiple times. Therefore, it could not be tested that the value is as expected, but it can be tested that the states of the two CRDTs should always be the same. When asserting on states, it can be concluded that the state when testing delays and offline support, is always the same.

5.2. Implementing block functionality

As discussed in the previous chapter, the block functionality will be implemented using different types of nodes in the tree for characters and blocks. A BlockNode and CharacterNode class were created which extends from the already existing Node class. The BlockNode serves as a node to represent blocks. A BlockNode has an extra blockId field, which makes it easier to find specific blocks, and therefore the Logoot field. The Logoot field in the BlockNode represents the content of the BlockNode. The CharacterNode represents a character. It has an extra value field, which represents the character

of the CharacterNode. For the merge and split operations, additional MergeNode and SplitNode were created, which both extend from Node as well. The MergeNode and the SplitNode both contain a referenceId, which should refer to a blockId. The MergeNode is being used to simulate that the block is actually at that position, whilst the SplitNode serves as a check for whether insertions and deletions should be delegated. A class diagram for these classes are shown in figure 5.1.

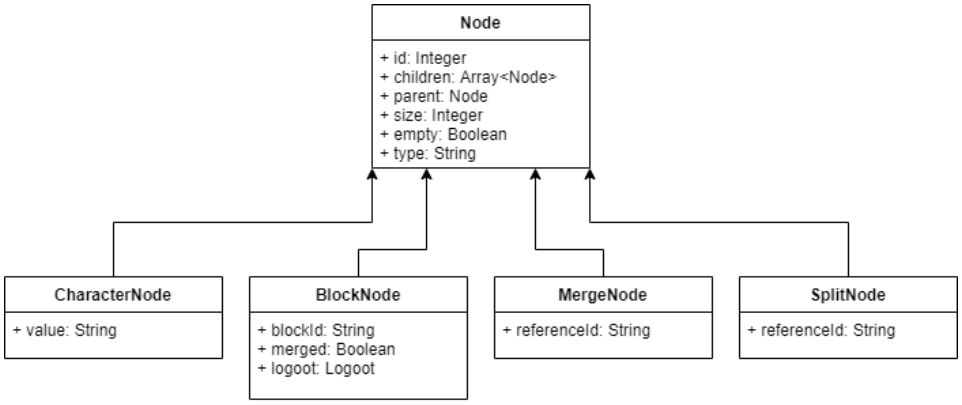


Figure 5.1: Class diagram for the node classes

According to the requirements, the BlockNodes should support the following operations:

1. Inserting blocks
2. Deleting blocks
3. Moving blocks
4. Splitting blocks
5. Merging blocks
6. Inserting content in blocks
7. Removing content from blocks

5.2.1. Insert and delete nodes in the CRDT

Before discussing the implementation of the operations described above, one should first know how to actually insert and delete nodes from the CRDT, since this method would be used in almost all of the above mentioned operations. The method to insert and delete nodes, is described in section 4.1.1.

5.2.2. Insert and delete blocks

For inserting blocks, the insert function for nodes can be reused, which is described in section 4.1.1. However, instead of creating a CharacterNode, a BlockNode must be created. This is done by creating a new method, specially developed for this sole purpose of creating blocks. This method creates the BlockNode and assigns a randomised blockId of length five. Just like the insertion for CharacterNodes, an emit message must be sent when inserting a BlockNode. This emit message contains the path to the BlockNode and the corresponding blockId. When the other replicas receive this emit message, it will traverse through the path and construct a BlockNode at the exact same position. The blockId which was sent will be assigned to this newly created BlockNode.

For deleting blocks on the other hand, it does not use the index, unlike the above mentioned method. Rather than using an index, the blockId is being used, since this is more compatible with offline operations. Cases exists when one tries to delete a block, but it is moved by someone else who worked offline. Since the previous delete operation was based on the path, the deletion would not delete the actual block since this has an other path. Therefore, it was decided to use the blockId instead of indices. The delete operation makes use of the searchBlock method for finding the block. When the

specific block is found, the block will be deleted in the same manner as the previous mentioned delete method. One difference with the previous mentioned method is that deleting a block will always result in a tombstone, even if the BlockNode does not have any children. This has been decided, because the CRDT does not know if any offline split or merge operations have been performed on this block on other CRDTs. Always using tombstones for BlockNodes will allow CRDTs themselves to decide what content should be kept and what content should be removed. The emit message sent upon deleting blocks contains the blockId. The other replicas which receive this message will search for the block with this blockId through the searchBlock method and deletes it in the same manner.

5.2.3. searchBlock

Searching for blocks is necessary for these operations: deleting blocks, moving blocks, splitting blocks, merging blocks, and inserting and deleting characters in blocks. The searchBlock function has one argument, the blockId. The search algorithm that is being used is the breadth-first search (BFS) algorithm. This means that it starts scanning every block after the root node from left to right on the same level, before it goes to the next level until it finds the desired block. BFS has a time complexity of $O(|V|+|E|)$. BFS is not the most efficient algorithm to search for a block, but BFS is the least complex to implement. In chapter 7, a more in-depth discussion about this searchBlock algorithm and other alternative algorithms will be given.

5.2.4. Insert and delete content in blocks

Inserting and removing content from blocks is implemented using the searchBlock function. First, the CRDT searches for the block. On the Logoot of the block, the CRDT can simply call the insert and delete methods described in section 4.1.1. After an insertion, the local CRDT sends the blockId and the path within the block to the other replicas along with the value. The other replicas search for that block and insert the node in that block at the given path. After a deletion, the CRDT sends the blockId and the path of the deleted node to other replicas. These replicas search for the block and delete the node at the given path as described in section 4.1.1.

5.2.5. Moving blocks

As explained in chapter 4, moving blocks has been implemented by locally deleting the block one wants to move and inserting a new block with the same blockId at the given index. To actually move the content of the block to the new block, the CRDT can simply transfer the Logoot of the old block to the newly inserted block. Since deleting a node does not necessarily remove the node from the tree, the CRDT also has to remove the blockId when deleting the block from the node, so one can ensure that there will not be duplicate blockIds in the CRDT. To send the move operation to other replicas, the message consists out of the blockId of the moved block and the path of the newly inserted block. The receiving CRDT can then delete the block, when inserting a new block at the given path while transferring the Logoot from the deleted block to the inserted block.

One problem occurred when two offline CRDTs move the same block to a different index. When both CRDTs go online, they will move the block to the index of the other CRDT, which results in diverging states. To solve this problem, timestamps were introduced. For every insert of a block, a timestamp will be saved. When receiving a move block operation from a different CRDT, this operation will only be executed if the timestamp is higher than the current timestamp of the block. This approach is based on the Last-Writer-Wins approach and will resolve the previous mentioned issue, since all CRDTs will have the move operation of the most recent timestamp.

5.2.6. Merge blocks

As explained in chapter 4, three different approaches have been designed and tested. In this chapter, the implementation of all three implementations will be elaborated upon. It will be explained how these approaches have been implemented and the challenges of all these approaches, as well as the reason why some approaches had been replaced. In this chapter the terms base block and merge block are used. The base block is the first block of the merge and the merge block is the block that is being merged into the base block.

First Approach

The first approach uses two additional nodes. These additional neighbouring nodes will be created under the root node and the Logoot structure of base block will be copied to the left additional node and the Logoot structure of the merged block will be copied to the right additional node. As shown in figure 5.2. The purpose of these two additional nodes, is that the path of the Logoot structure of the merged block does not change. Delayed or offline insertions or deletions in the merged block will be propagated to the right additional node.

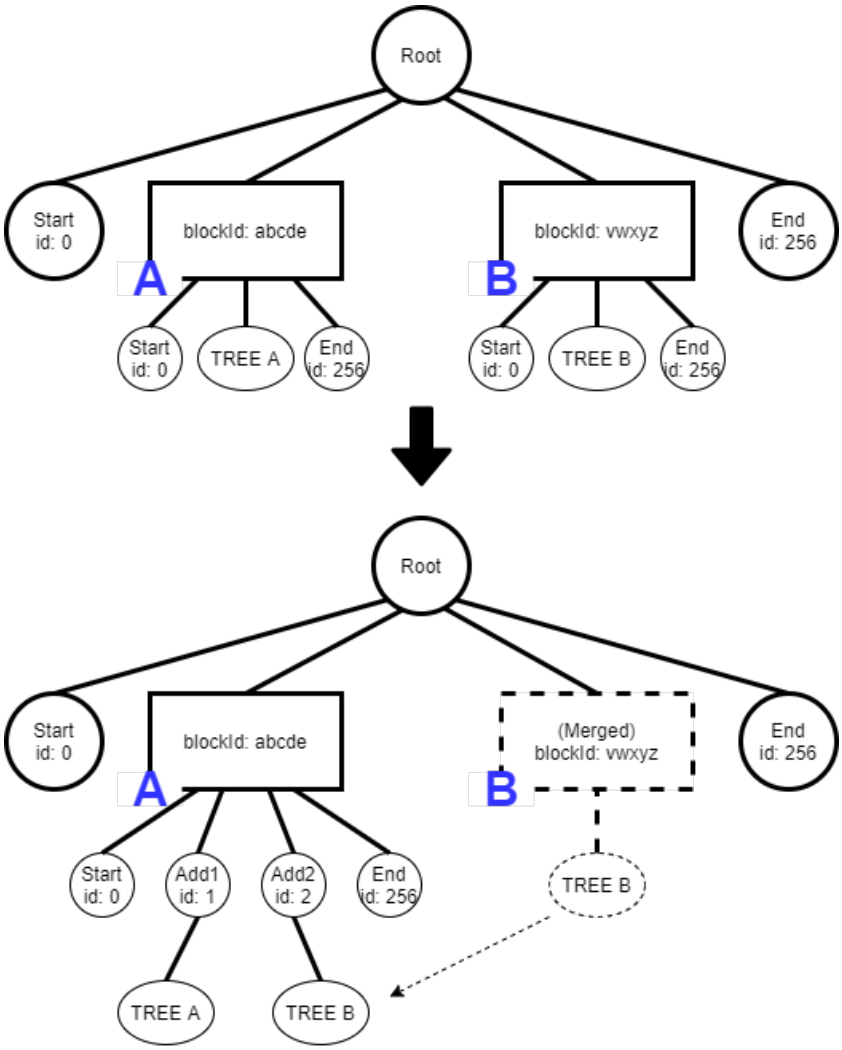


Figure 5.2: Merging block B into block A with first approach using additional nodes

In figure 5.2 there is a so-called 'MergedBlock'. During a merge, the block that is being merged into another block is replaced with a MergedBlock to indicate that it has been merged into another block. All the nodes that are part of that blocks Logoot are removed and inserted as children of the right additional node, keeping the exact same tree structure (node Add2 in figure 5.2). When the CRDT receives an operation for the block that has been merged, the MergedBlock will propagate the operation to the base block where the additional nodes are (block A in figure 5.2). When propagating the operation to the base block, the MergedBlock adds the identifier of the right additional block to the path in the operation. Since the right additional block has exactly the same tree structure as the MergedBlock, the operation will be performed on the correct position in the tree. In figure 5.2 for example, if the CRDT receives a delayed insertion in block B on path [3,4], the MergedBlock B will propagate this insertion to block A and add id 2 to the path. So the MergedBlock B will propagate an insertion of path [3,4] to block A with path [2,3,4].

However, this approach does not work very efficient with delayed and offline merge operations. When looking at the scenario described in figure 5.3, when two offline CRDTs merge different blocks with block A, then after both CRDTs go online, they have to merge the other block with block A and with this approach. Achieving a convergent state would not be an easy task. This would require using timestamps and removing and merging the blocks again in the order of the timestamps. However, removing sub-trees and inserting these trees again at a different position is very complex to do while keeping the tree intact. Therefore, it has been decided to not move trees when merging blocks, but create references, such that it would be more efficient and easier to solve scenarios as described in figure 5.3. A more detailed explanation of using references will be described in approach two and three of merging blocks.

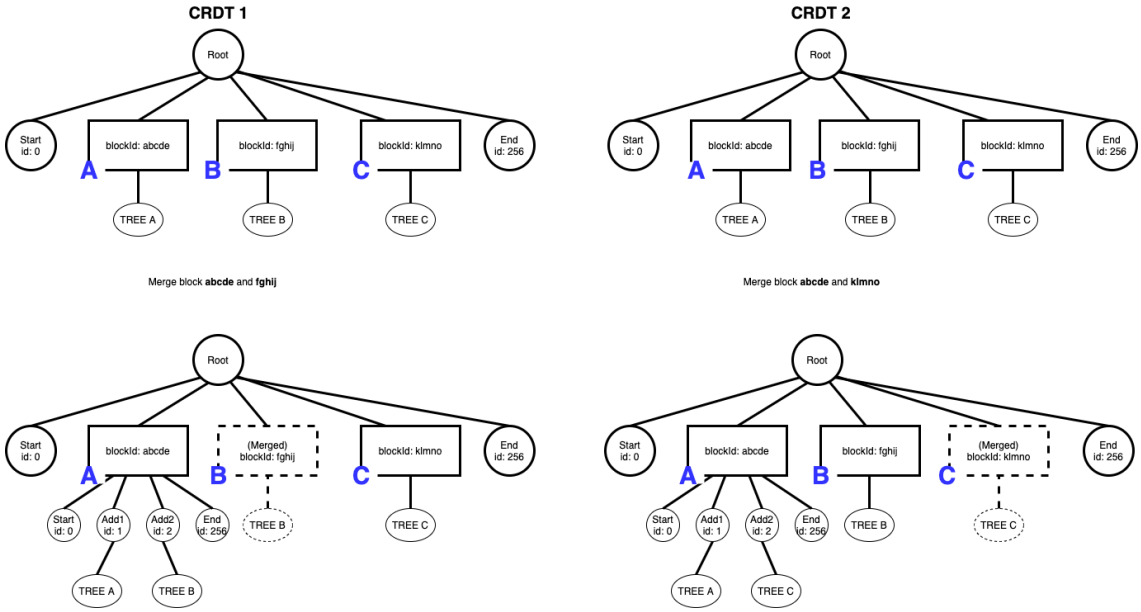


Figure 5.3: CRDT with block A, B and C and one CRDT merging block A and B and the other CRDT merging block A and C

Second Approach

The second approach uses the so-called 'MergeNode' in a block to indicate that a block has been merged into that block. The merged block will then be marked as merged, so when reading the content from the CRDT, the merged block will only be accessed through the reference node (MergeNode). So for example see figure 5.4 below, where two blocks are merged and will output the word 'Fonto'.

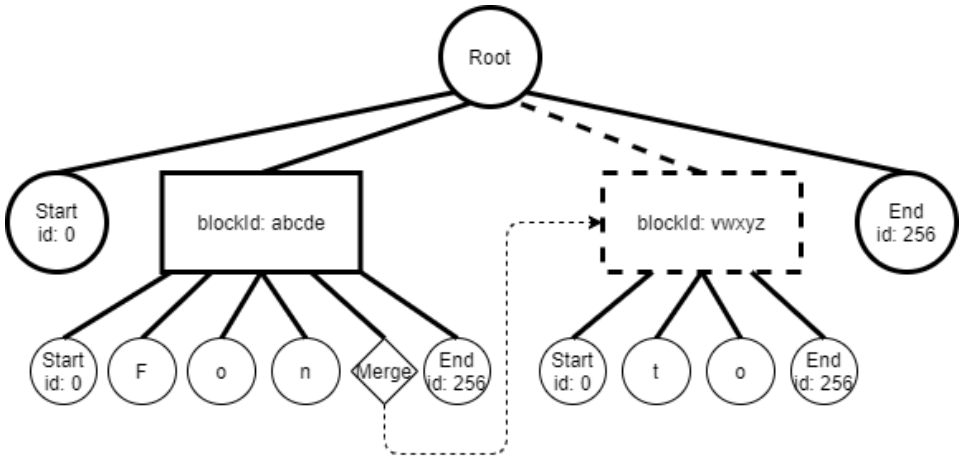


Figure 5.4: A tree containing 2 merged blocks

The tree is read in depth-first order and will only print CharacterNodes which are not set to empty. When a MergeNode is read, the content of the merged block will be read instead of the MergeNode. When a merged block is read through the depth-first search (DFS), it will not be read since the merged block can only be read through a MergeNode. In figure 5.4 the merged block is only read through the MergeNode. The word which will be outputted would be 'Fonto'.

One problem that occurred was that two offline CRDTs can merge the same two blocks before going online. When both CRDTs go online, this will result in a duplicate merge, since both CRDTs will also execute the received merge. This has been solved by adding a timestamp to the merged block and using the Last-Writer-Wins approach. When a block is already merged and it receives another merge operation, the merge overwrites the current merge if the timestamp of the received merge operation is higher than the timestamp of the current merge. Overwriting a current merge is done by deleting the current MergeNode and inserting a new MergeNode. This way, all CRDTs will only merge the block with the most recent timestamp.

To support insertion of characters with the merge functionality, it should be checked whether the inserted CharacterNode should be placed inside the base or merged block. As explained in 4.1.1, to insert text, the CRDT searches the block in which the CharacterNode needs to be inserted. The position for the node to be inserted would be generated by the Logoot of that particular block. To generate this position, the left and right neighbour of the new node are looked for. Based on the ids of these neighbours, a new CharacterNode is inserted between them. When inserting in a block, which was merged before, four cases had to be considered:

1. Insert between two CharacterNodes in the base block
2. Insert between two CharacterNodes in the merged block
3. Insert between the base and MergeNode
4. Insert after the MergeNode

Insert between two CharacterNodes in the base block
Finding the index for an insertion in the base block is the same as a normal insert. A previous and next CharacterNode are found and the new CharacterNode is inserted between these two nodes.

Insert between two CharacterNodes in the merged block
When the given index for an insertion falls in the merged block, then the insertion must be propagated to the merged block with an adjusted index. To achieve this, the Logoot of the base block is traveled in depth-first order and a counter keeps track at which node the DFS currently is. When a MergeNode is discovered and $(index - counter) < size\_of\_merged\_block$, then the node must be inserted in the merged block. Then the insert is propagated to the merged block with a new index $(index - counter + 1)$. The $(+1)$ is needed to make sure that the index is correct. Insertions between the StartNode and the first CharacterNode of the merged block should not happen, since that is the same as inserting between the last CharacterNode and the MergeNode in the base block, therefore the index needs to be corrected with $+1$.

Insert between the base and merged block (end of the first block)
When inserting characters between the base and the merged block the situation arises that there are two places the node can be placed to obtain the exact same result. The new CharacterNode can either be inserted between the CharacterNode just before the MergeNode and the MergeNode itself or it can be inserted in the merged block between the StartNode and the first CharacterNode of the merged block. The decision had been made to insert the new CharacterNode between the CharacterNode before the MergeNode and the MergeNode to limit the amount of operations in the merged block, since that block does not really 'exist' anymore.

Insert after the merged block
When inserting after the merged block, there are two possible solutions. Placing it at the end of the merged block or placing the insertion after the MergeNode in the base block. To limit the amount of

operations required in the merged block, the decision had been made to insert in the base block after the MergeNode.

When deleting a CharacterNode in a block that has been merged there are two cases to consider:

1. Delete CharacterNode in the base block
2. Delete CharacterNode in merged block

When deleting a CharacterNode, the Logoot travels through the tree in DFS order, while keeping track of a counter. When the counter is equal to the index of the delete operation the CharacterNode can be deleted. When a MergeNode is discovered, a check is done to see whether the deletion is in the merged block or not with $(index - counter < size\_of\_merged\_block)$. If $(index - counter < size\_of\_merged\_block)$ is true, the deletion is in the merged block and the operation will be propagated to that block. If $(index - counter < size\_of\_merged\_block)$ is false then there is text after the merged block and the deletion is not in the merged block, so the deletion stays in the base block with a corrected index, $index - size\_of\_merged\_block$, since all nodes in the merged block can be skipped.

Moving a merged block has no influence on merge, when the base block is moved, all contents of the Logoot instance is moved with it, including the MergeNode. When the merged block is accidentally moved, for example when someone is working offline, it does not matter, since the merged block is only accessed through the MergeNode, thus its position in the tree does not matter for the other operations.

Issue with approach two
Splitting blocks in combination with this approach had some issues as explained in chapter 4. The reason for this is that it is not possible to guarantee that the merged block is also the last content, since characters added at the end of the block would be inserted in the first block. An idea to propagate all insertion on the edges of the merged block into the merged block would not work, since then replicas would diverge because of different order of execution. Therefore the split implementations would not work since split takes the content in a block from the split index to the end of the block and places that content in a new block and removing it from the split block. So if the split would happen in a merged block, then if there was content written after the merge and at the end of the merged block it would still be in the first block but it should have been added to the block created with split. See figure 5.5 for details.

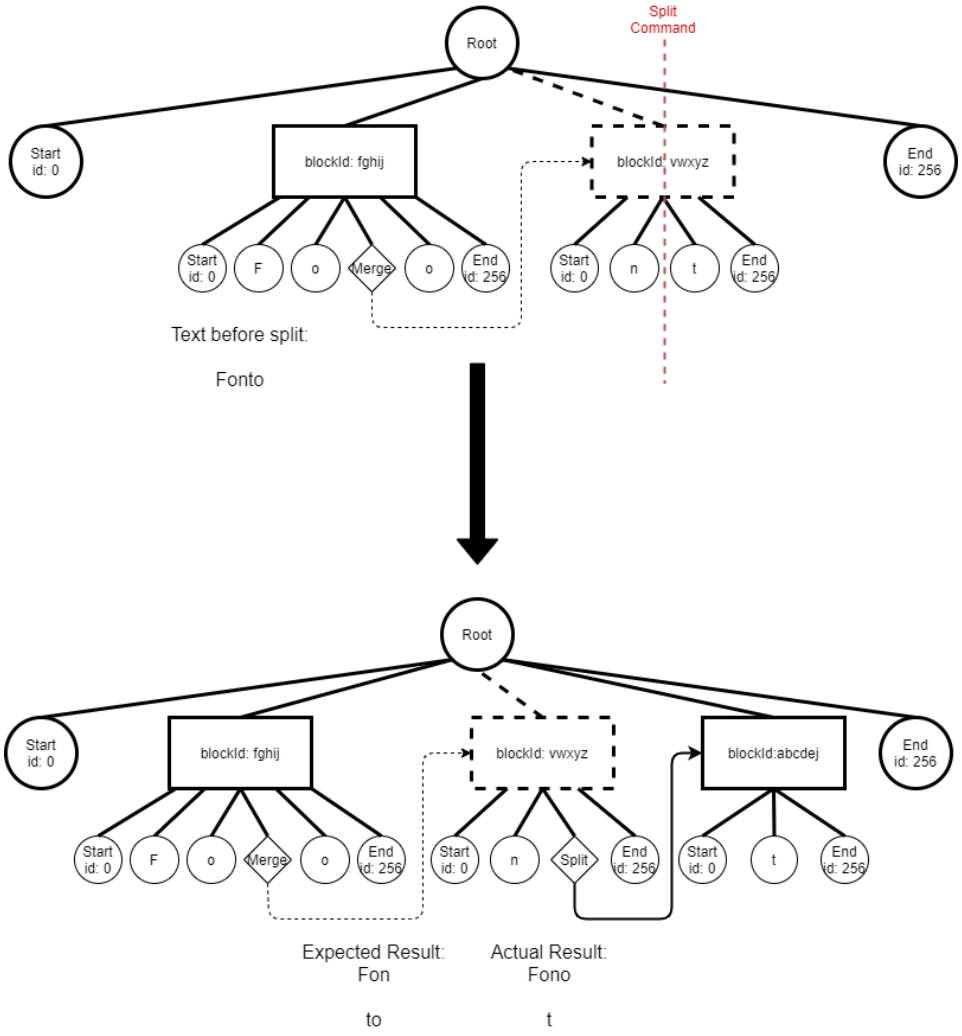


Figure 5.5: Merge and split operation conflict

Approach two also had issues when multiple replicas do the same merges offline, since this approach had no way to detect identical merges, which have been merged in different order. It was possible to gain duplicate references. This would result in text being displayed multiple times. As example, if there is a CRDT consisting out of three blocks. Replica one merges the block in the order: block 1 with block 2 and then block 1 with block 3. Replica two merges the block in the order: block 2 with block 3 and block 1 with block 2. Both should result in the same outcome (all two blocks merged into one). But since the CRDT cannot detect that this sequences of commands are exactly the same, duplicate references will appear in the CRDT. See figure 5.6 for details.

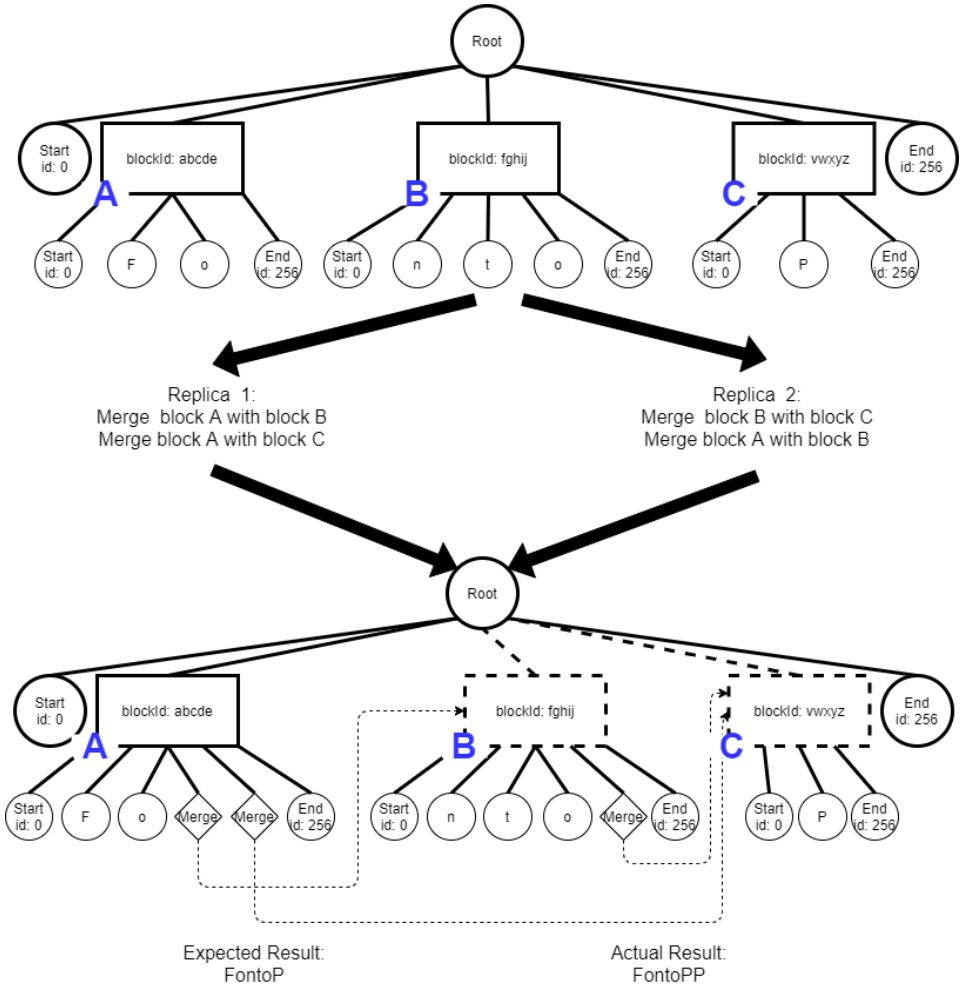


Figure 5.6: Multiple Offline Merges Conflict

Due to these and other bugs/flaws in the second approach, a third approach has been designed to tackle mostly these two problems.

Third Approach

The third approach is based on the second approach, but with adjustments to tackle the issues discovered in the second approach. To make split and merge compatible with each other, when two blocks are being merged, the end node of the base block is being converted into a MergeNode. A merged block has also been expanded with a timestamp, to check when two blocks have been merged. This timestamp will be used for keeping track of the order of merges. Since the MergeNode is now at the end of a block it is no longer needed to check if a merge was in a merged block or not. Insertions and deletions done outside of the base block can be automatically propgated to the merged block. Each block can only have one MergeNode since each block has only one end node. When multiple blocks are being merged into one block, a daisy chain has been formed instead of adding multiple MergeNodes in one block, as you can see in figure 5.7.

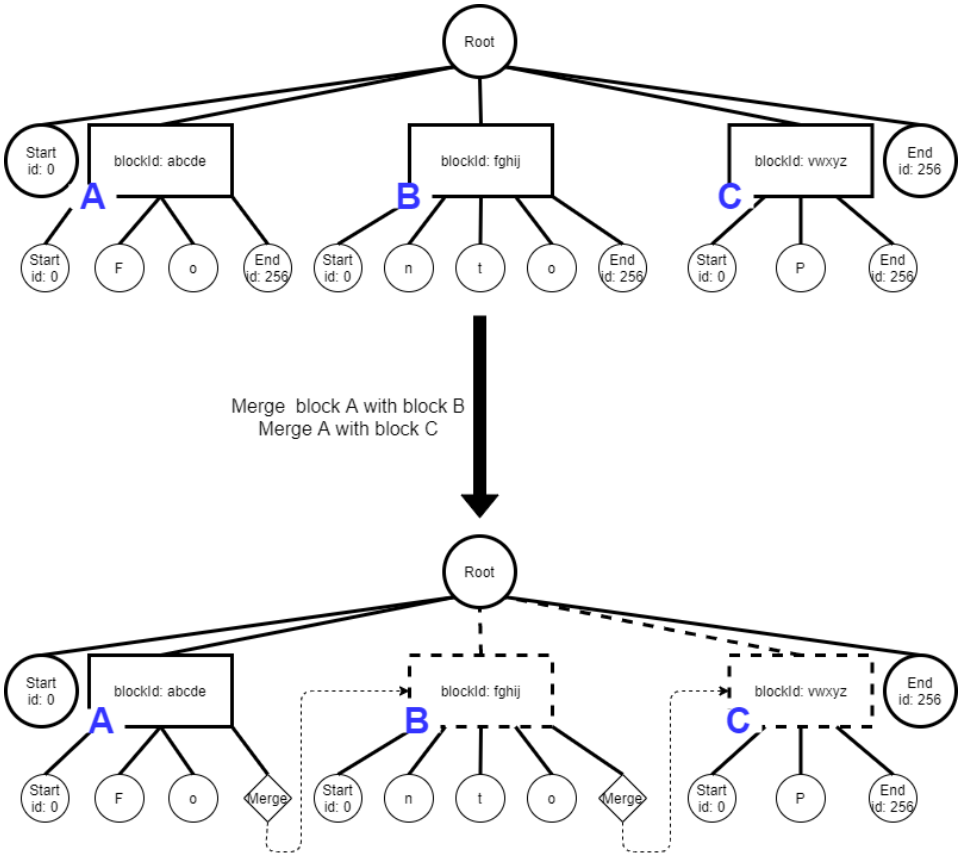


Figure 5.7: Merge third approach daisy chain

Since now it is impossible to have nodes in a block after the MergeNode, splitting will not result in unexpected behaviour, as in approach two.

To fix the issue of duplicate references, a new method has been created when merging two blocks. When a merge command is received, the CRDT rebuilds the entire block to see if there have been multiple merges which attain the same result. To clarify this method it will be explained with an example where replica one has the state shown below in figure 5.8 and receives a merge command from replica two, which has been working offline.

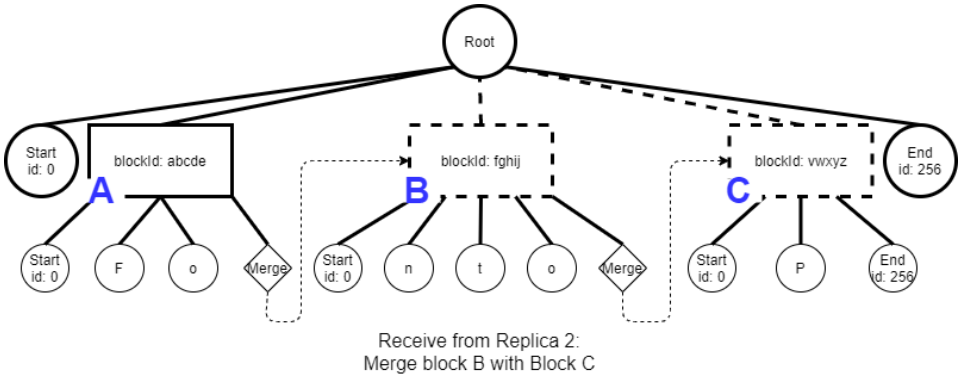


Figure 5.8: State of replica one

When the merge from replica two is received by replica one, one list is created, this list contains all merges in the base block, the merge command and all merges in the merged block. First, the list

containing all the merges in the base block is created, figure 5.9. This list contains objects existing out of; from block, to block and timestamp.

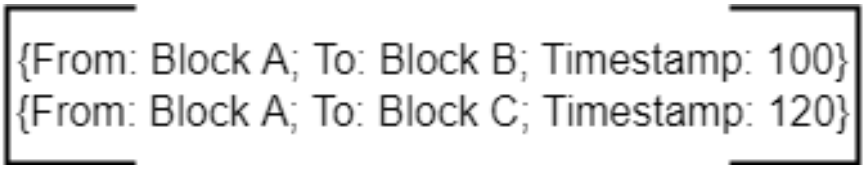


Figure 5.9: List created from the base block

Then, the received command will be converted to an object, which could be added to the list, figure 5.10.

Figure 5.10: Object created from the merge received from Replica two

First, it is checked if object in the list from the base block have the the same ’to’ location as the received merge. If so, then the block is already merged into this block and it gets ignored, so in the example the merge command can be ignored, since there already is a merge from block A to block C, and the merge command wants to merge block C into it, which is already merged.

If there is no matching ’to’ location in the list from the base block then the blocks can be merged. Another list is created from the merged (to block), containing all the merges in this block, these lists are then appended to each other and ordered based on timestamp.

To complete the operation, all merges in the list are removed from the CRDT and re-inserted, based on timestamp. This is needed to ensure that all replicas have the same order of merges, which is needed to ensure convergence. See figure 5.11 for an entire oversight of a merge.

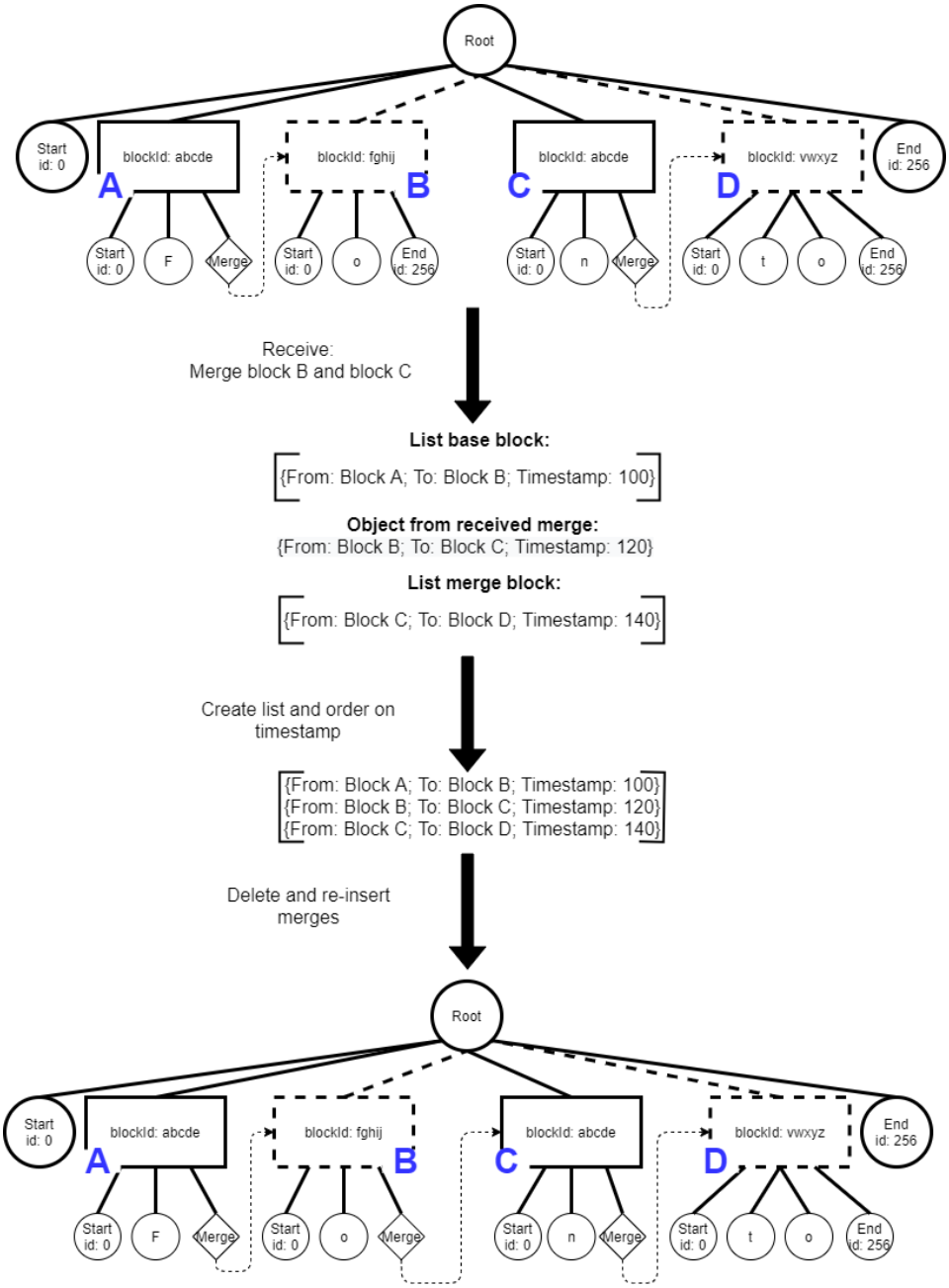


Figure 5.11: Overview of a received merge with approach 3

This approach fixed the issue of duplicates and ensured convergence when replicas merge blocks in different orders, since now every replica will order merges based on timestamp.

One issue that is still not solved with this approach is the creation of circular merges, when replica one merge block A with block B for example, and replica two merges block B with block A, then they both have a MergeNode referencing each other at the end. This creates a circular merge, a circular merge means all merges are connected to each other and there is no 'starting' point for the CRDT to enter this merged block. Therefore, when asking for the value of the CRDT, all of the content in the merged blocks is not printed, since the CRDT cannot find an entrance point. A more detailed explanation about this issue will be discussed in section 6.3.

5.2.7. Splitting blocks

As briefly explained in chapter 4, splitting blocks makes use of a so-called 'SplitNode'. When splitting block A at a certain index, the SplitNode is inserted in the same fashion as normal character nodes on that index. Afterwards, a new block B is created with a different blockId, and the content is copied over to block B. In block A, all the nodes on the right side of the SplitNode will be removed, whilst in block B, all the nodes, including the SplitNode, on the left side will be removed as explained in chapter 4. The SplitNode in block A serves as a reference to block B for operations which are delayed and performed before the split operation. If a delayed operation occurred after the index of the split, this operation will be redirected to block B using the SplitNode. In such case, when the SplitNode is absent, the delayed operation would be performed in the wrong block (figure 5.12). In this figure, TREE A and TREE B are a generalisation of Nodes. So TREE A and TREE B, could be a tree with the CharacterNodes a and b, but could also contain other CharacterNodes or SplitNodes. If delayed messages were no issue, the SplitNode was not needed, because one can assume that all received operations can be executed on the mentioned block in the received message. Delayed messages, however, are an issue since one can potentially have slow internet or work offline. The introduction of the SplitNode solves this problem. The SplitNode can be removed, when it is verified that all users have received this split operation. However, with this current approach, this is not verifiable. As described earlier, vector clocks could be used to verify if every user has received the split operation.

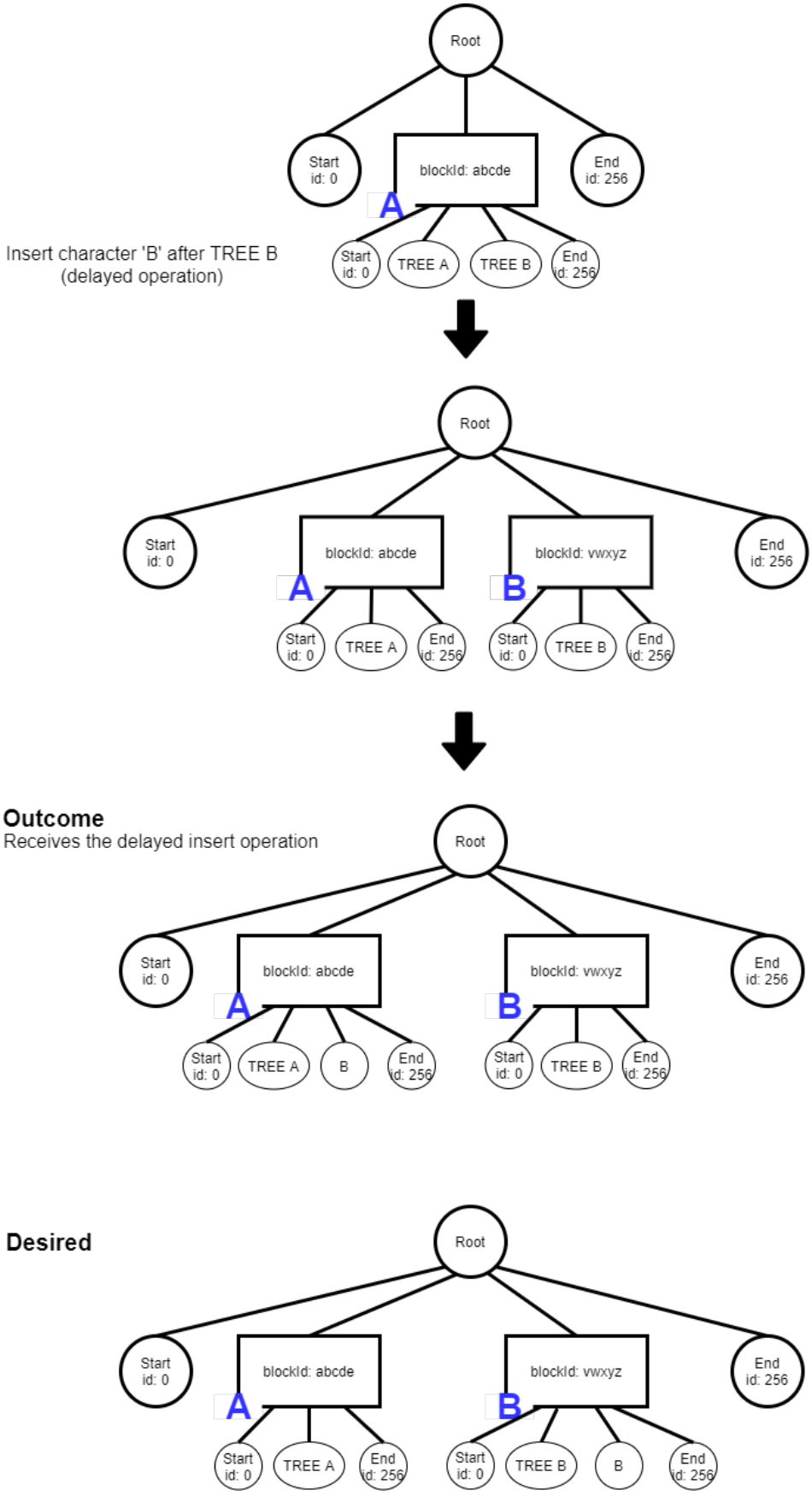


Figure 5.12: Receiving a delayed insert 'B' operation after splitting, which should be placed in block B

Zooming in on the insert operation shown in figure 5.12. The insert operation performed on the first tree sends a message that it inserted a node with value 'B' on that particular location. To illustrate this, see figure 5.13. Note that this figure includes the SplitNode. For this example, the SplitNode can be ignored. When CRDT 2 receives the message of CRDT 1, the node would be built in block A on the location with the id equal to 4. When CRDT 1 receives the split operation of CRDT 2, the node with value 'B' will also be moved to the second block. The latter outcome seems more user friendly, and easier to maintain. To ensure that both CRDTs converge to the same state, the SplitNode plays an important role. When a SplitNode is present, an operation which should be delegated to block B always has this same characteristic. This characteristic is that all paths of these operations are referring to the neighbour of the SplitNode. So in this figure, CRDT 2 receives the insertion. When the path is checked, it can be seen from the figure that the node would be inserted in block A after the SplitNode. This means that the content was already moved to block B. Since the tree structure of block B was an exact copy of the tree structure in block A, the path of the insertion is still valid, even for block B. So for every insert operation, it will be checked whether the left neighbour of the given path is a SplitNode. When this is the case, the insertion will be delegated to where the SplitNode is referenced to. The exact same scenario also applies for the delete operations. Each delete operation would check on the path whether the left neighbour is a SplitNode. If this is the case, then the deletion would, just like the insertion, be delegated to the referenced block.

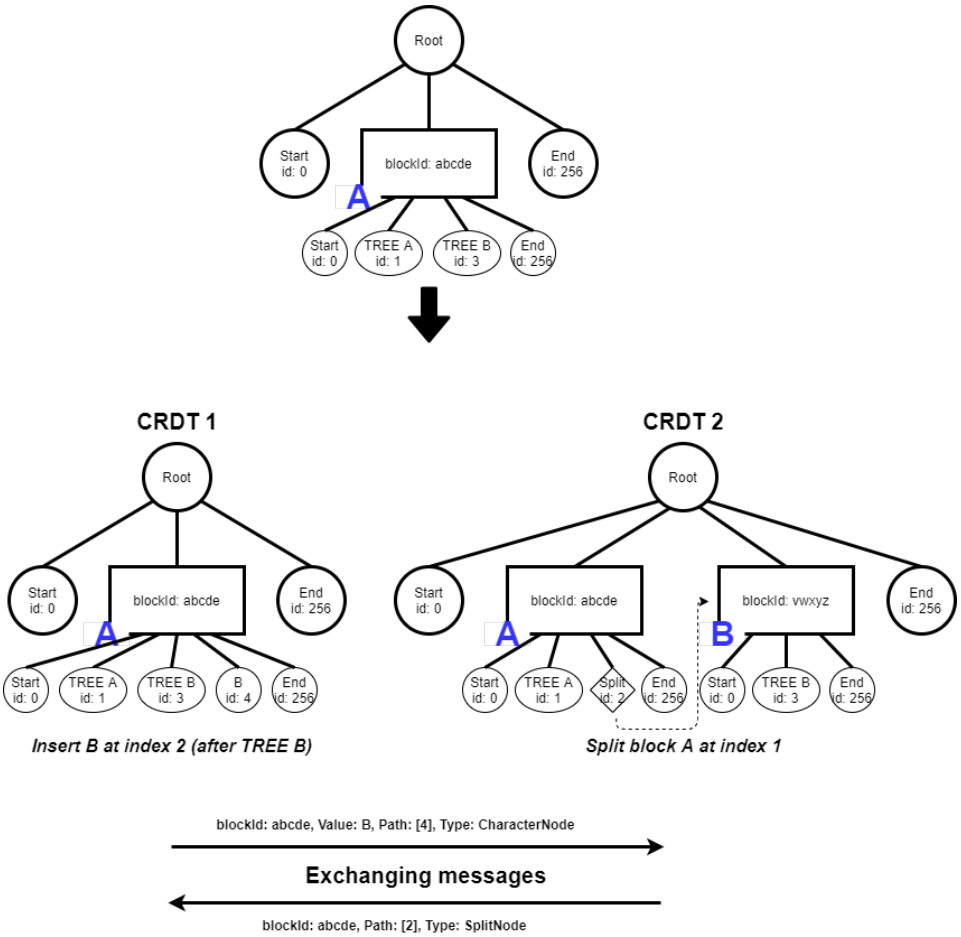


Figure 5.13: The message exchange between two CRDTs where one replica inserts the value 'B', while the other replica splits block A

5.3. Validating the block functionality

To validate the functionality described in the previous section, many tests had been written using the same Mocha test framework. When writing tests for these operations, tests had been written for the

simple cases, edge cases and combinations between different operations. Most edge cases include multiple operations in a row or operations executed when CRDTs are working offline.

When analysing time and space complexity of the block operations, there are a few operations, which might affect the performance of the block operations. The inefficiencies of these block operations will be discussed in section 7.2.1.

To make it easier to write tests where CRDTs are offline, a simple framework had been implemented to simulate offline CRDTs. This framework can easily be used without having to know how the CRDT itself sends message to other CRDTs. The framework contains functions such as createCRDT, setOnline and setOffline to create a CRDT or to set a CRDT online or offline.

After writing many tests for all the block functionality, it turned out that most functionality works **if** all CRDTs stay online and that most bugs appear when CRDTs start working offline. The chapter 7 will go deeper on what bugs have been found when CRDTs work offline.

6

Testing the prototype

In addition to writing tests, a research prototype has been developed to showcase the functionalities of the CRDT implementation by making use of a custom-made editor. This chapter will discuss what features the editor has and what bugs have been found when (manual) testing the CRDT. This editor and the written tests are being used to proof that the described concept in chapter 4 and 5 do meet the requirements.

6.1. Editor

6.1.1. Editor design

When implementing a block-based editor, one could use newlines or other identifiers to identify blocks of text. In this approach, it would be difficult to visualise merge, move and split operations. Therefore, it has been decided that the block-based editor will contain multiple text areas, where each text area represents a block of text. Using this approach, it will be clear what a block is and how the document structure changes after specific operations have been applied. The merge, split and move operation will be executed using a button click instead of a hotkey.

To send messages to different online editors, WebSockets have been used. This report will not go deeper on how WebSockets have been used in this editor, since WebSockets are not relevant for this project. WebSockets make it possible for this editor to show real-time collaborative editing on different devices. It has been decided that the server also has a CRDT, which will be updated based on all operations received from all CRDTs. This CRDT on the server will be used to give new users the current state of editor.

6.1.2. Editor features

The editor contains all functionality that is necessary to perform all implemented operations in the CRDT. These operations include:

1. Inserting text in a block
2. Deleting text in a block
3. Inserting a block
4. Deleting a block
5. Splitting a block at a given index
6. Merging two blocks
7. Moving a block to a given index

In addition to these features, offline support and some extra visualisations were added to simulate situations where one or multiple users work offline and to give users a better view of what happens with the CRDT when editing blocks.

Offline support
The first additional feature added, is offline support. In the editor, there is a button, which can be used to toggle the user to either offline or online without actually having to turn off the WiFi.

When a user is offline, it will enqueue all operations to send to other users until the user toggles the offline button to online. When a user is offline, all received operations will not be executed locally until the user toggles the offline button to online.

Extra visualisations
After that, some visualisations have been added. First, two columns have been added, which shows the blocks as HTML and parsed HTML. This has been added to show that this editor is indeed a representation of XML and that this block-based approach could indeed be used by Fonto.

Finally, the user can get the state of the CRDT of the server after every operation. This makes it easier to find out what happened with the state once a bug appears.

6.2. Testing

To ensure correctness of the CRDT, tests have been written for automatic testing, but also manual testing has been done to find edge cases.

6.2.1. Automatic tests

For testing the CRDT implementation, a test environment was created. The approach at the beginning was to create simple test cases where only text characters were inserted and deleted, like plain-text, at a certain index, where exceptions should correctly throw whenever indices were out of range. When the CRDT was further developed to support blocks, tests were written for inserting and deleting blocks at a given index. These test were pretty straightforward and tests cases were easily created.

Afterwards the tests also contained insertions and deletion of characters inside a block. The difference between adding plain text, as described above, and adding text in blocks, is that it also has to specify the blockId. For this subtle difference, tests have been written to check whether characters have been inserted and deleted from the correct block.

When testing the CRDT with offline operations, instead of immediately executing a received message, the CRDT will push this message to a queue. Once the CRDT is online again, the CRDT will execute all received messages from the queue in the received order.

Once the CRDT also supported moving, merging and splitting of blocks, it became quite harder to create test cases. Since a user can freely move, merge and split blocks multiple times, there were a lot of edge cases that had to be considered. With the introduction of offline operations it became very difficult to write tests to exhaust all possible combinations. When writing tests, it was also important to test combinations of different type of operations, since one operation could break the functionality of a different operation. An improvement for the future could be to write a utility function to create all permutations given a sequence of operations, so one does not have to think of all possible combinations.

Currently, these tests do not completely prove correctness of all operations, since it could not be concluded yet that all possible combinations of operations has been tested. To prove correctness of all operations, one could write a utility function, as described above, to generate all possible permutations of operations. If all these tests pass, it could be concluded that the CRDT is working as expected, since all possible combinations of operations are covered.

6.2.2. Manual tests

Throughout the development of the CRDT implementation, the editor was mostly used for exploratory testing. After finding a bug where the content of the CRDT did not converge or when the user intent was not captured correctly, it was much easier to create test cases for the automatic tests.

With the editor it was a lot easier to find edge cases for the test environment, since the testers could put themselves into a position as if they are working on a document. As said in the previous section about offline support, the editor has a button to quickly simulate offline and online states. This also makes it a lot easier to test the features while being offline.

6.3. Known bugs

In this section, bugs that still appear in the research prototype will be discussed. It will be discussed why the bug occurs, what has been tried to solve it and how it could potentially be solved in the future.

One bug that could not be solved yet, is a circular merge. What happens here, is that two CRDTs move the same blocks while being offline in the opposite order. So if the CRDTs have block 1,2 and 3 and one CRDT merges block 1 and 2 and block 1 and 3 sequentially and the other CRDT merges block 3 and 2 and block 3 and 1 sequentially, when applying all these merges together, the final CRDT will have blocks with references which result in a circle, see figure 6.1 for an example.

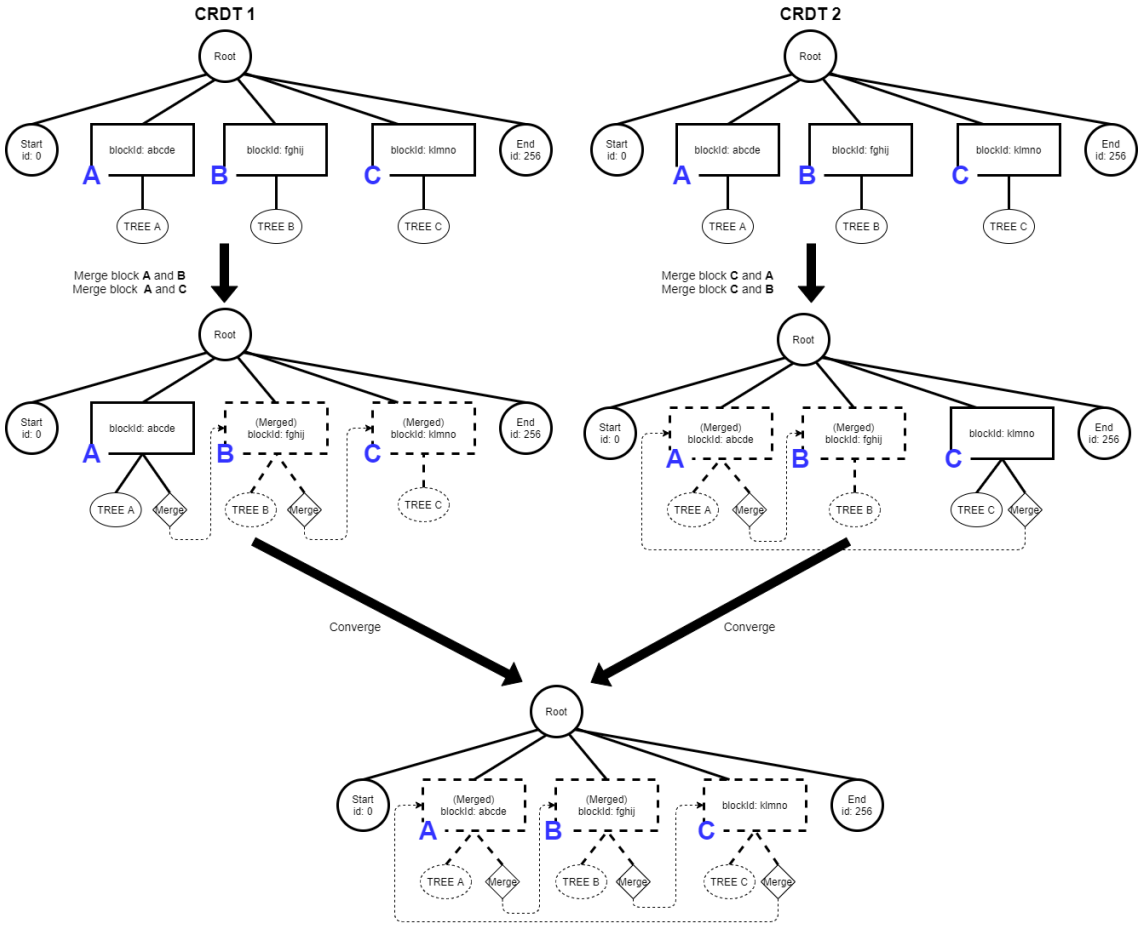


Figure 6.1: A situation in which a circular merge occurs

An approach which has been tried to solve this, is using timestamps. Each merge operation gets an assigned timestamp, and after each merge operation, all merge operations will be reverted and reordered based on timestamp. All merge operations will be applied again in this new order. Before applying all merge operations, the algorithm will check if there are circular references. It will check if for example,

block 1 has a reference to block 2 and block 2 has a reference to block 1. If a circular reference has been found, the timestamps will be compared and the merge operation with the highest timestamp will not be applied, hence removing the circular reference.

Currently, this bug has not been resolved yet, but it could not be concluded yet if this approach does not work or whether the implementation of the approach still contains some bugs, unrelated to this method.

The final bug found in the current implementation is that when splitting a block and merging the two resulting blocks from the split, that the content from the right block could be lost on some replicas, while not on other replicas, resulting in diverging replicas. Most likely, this bug occurs because there will be a MergeNode after a SplitNode. Since every insertion and deletion after the SplitNode will be redirected to the reference block, the MergeNode will be inserted in the reference block, causing content loss. It has however not been verified that this is indeed causing the bug when merging a split block.

7 Results

In this chapter, the results found for the proof of concept will be described and discussed. Then based on these results, a recommendation is given for Fonto and future research.

7.1. Test results

All automated tests created to verify the functionality of standalone operations, combinations of operations and offline functionality passed when ran 1000 times (with exclusion of the tests covering the known merge bugs).

With manual testing no bugs were found in all operations except for the merge operation when testing with the final version of the created CRDT.

The remaining bugs for the merge operation are circular references and the split and merge operation combined as described in section 6.3. The mentioned test results are only based on the known test cases. There is a possibility where some edge cases are not found yet. Generating all kinds of permutations of the available operations will potentially find more edge cases, which have never been thought of.

The remainder of this chapter will discuss the design and implementation of the CRDT, what the strengths and weaknesses are and how the CRDT can be improved.

7.2. Discussion

7.2.1. CRDT

After implementing all mandatory requirements, there are two known bugs left. One where the merge operation causes circular references and one occurs when merging and splitting one block multiple times.

The remainder of this section will further discuss the strengths and weaknesses of the design choices and implementation of the CRDT. One of the disadvantages the skeleton code has, is that each character is a node itself and therefore the tree size can become extremely large when editing large documents. Large trees will most likely result in slower insertions and deletions of characters. Since clients of Fonto can have large documents, it is important, that the tree size will be optimised.

Another weakness, also in the field of tree size, is that deleting nodes might create tombstones. A tombstone will be created if the node the user is trying to delete has child nodes or if a user tries to delete a block. For small documents, this does not matter, but once users start deleting lots of text, there could exist a lot of tombstones. This might result in unnecessary slower insertions and deletions of text, since the CRDT might have to traverse all the tombstones to find a node.

Another weakness of the current approach, is that the searchBlock method uses a BFS algorithm. This is not the most efficient algorithm to find a block as quick as possible and when the amount of blocks in the document increases, this algorithm will also get slower.

With the final approach of merging blocks, when a large amount of blocks is merged in one block, the complexity of this approach might increase since every previous merge operation related with the blocks will be removed and inserted again after every merge operation.

One weakness in the split block approach, is that for every split operation, the whole block is copied to the newly inserted block. When splitting extremely large amounts of content, every node has to be created again and this might take some time if the size of the block is large. This is unwanted behavior, since with real-time collaborative editing, users expect operations to be visible immediately.

One strength of this CRDT is how blocks have been implemented. Since basically each block has its own CRDT for the content in the block as a Logoot field. Therefore, it should not be difficult to implement nested blocks in this CRDT. This would however increase the complexity of the move, merge and split operations, but adding the functionality of nested blocks itself would not be difficult to implement since each CRDT already has the ability to create blocks.

This approach of implementing blocks also creates smaller sub-trees in the different blocks. This would help in the aforementioned problem of tree size, since the total tree size of the document will be divided over all blocks in the document. If the searchBlock algorithm is optimised as well, this will partially solve the problem of large tree size. The problem of large tree sizes would in this case only be a problem if the blocks itself have a large amount of content.

Another strength of this CRDT implementation is that new types of nodes could easily be added to the CRDT. If special nodes must be added to represent other XML tags for example, these could easily be added. Therefore, this CRDT implementation is easily extendable.

Finally, this CRDT implementation uses JSON objects to send messages to other replicas. This means that there is a lot of freedom to decide how to implement the network layer. This could be done the same way it is implemented in the editor using WebSockets, or any other way which supports sending strings over the internet.

7.2.2. Feedback Software Improvement Group

In the sixth week, code was delivered to the Software Improvement Group (SIG) for evaluation. The code base was mainly taken from an online Git repository. This repository is the skeleton of the whole code base that was submitted. The provided code could not be removed for the evaluation, since that would result in just a few functions without showing relevance. The feedback given from SIG were overall quite moderate. The metrics that are relevant for this project and the evaluation scores are: code duplication (4.0), unit size (2.9), unit complexity (3.3), unit interfacing (3.7), and module coupling (5.5). The places that needs to be improved mainly regards the sizes of the functions. The amount of if-statements, parameters and lines should be reduced in order to score higher for the second evaluation.

Regarding code duplication, duplicated code has been moved to separate methods, which can be reused. However, not all duplicated code has been refactored. Some duplicated code had small changes in indices or parameters. Although it could be possible to create different methods for this duplicated code as well, it has been decided to not add those methods, since most of the duplicated code was subject to change.

The biggest contributor to the grade of unit complexity was the receive method. This method could however not be refactored, since it contains a switch statement which checks what kind of message has been received. For each kind of message, a separate method was created. The cases of these messages can not be decreased, since each type of message needs to be handled respectively.

The unit interfacing metric could not really be improved, since for most methods, all parameters are

necessary. Although improving unit interfacing could be achieved by putting multiple parameters together into one object, it has been decided to not do this, since this will add some unnecessary space allocation to the code. The product is a prototype and improving the unit interfacing metric with the above described method, would not improve the software development cycle for this project.

Finally, the unit size of several methods have been reduced. The unit size has been reduced by moving blocks of code which tried to achieve the same sub-goal into one method. Since code has been split based on sub-goals, moving some blocks of code would not be logical to move into separate methods. Therefore it has been decided to keep these methods as is.

7.2.3. Evaluation

Overall the project went well, the tasks were clearly defined by Fonto. An overview of the entire project was made, dividing the entire project in a research (two weeks), investigation, implementation, modifying and documentation (two weeks) phase, with one week left for the presentation. This planning has been followed over the entire project, making it overall easy to poll our progress and ensure that we did not fall behind. Each week we also made a small scrum planning for that week and at the end of each week there was a meeting with our coaches from Fonto to discuss findings and asks questions.

In the first week, the planning was made and research question were though of, also some brainstorm sessions were held for how we wanted to tackle the implementation phase. During the research phase we focused the first week on CRDTs in general, what are CRDTs and how do they function? The investigation week was an in-depth analysis of CRDTs and find the type of CRDT which would be the best for our purpose. We also researched if there was an existing CRDT on GitHub which we could extend, since that would allow us to focus more on the research requirements which were not yet available, instead of already existing features.

At the start of the implementation the basis for the research prototype was created and a test environment was build. A text-editor was extended and combined with our CRDT for manual testing and the start was made with the insert and delete functionality for characters. The second week a start was made to support blocks of text and the insertion, deletion and moving of blocks, also brainstorm sessions were held for the merge, and split functionality. The modifying week were solely used for split and merge functionality. The split functionality was completed quickly, but the merge implementation was more difficult than anticipated, therefore we tried multiple approaches. At the end of the implementation phase, merge still contained some bugs and was not entirely compatible with the split operation. Still, the decision was made to stop with implementing and continue with the documentation phase.

7.3. Recommendations

In this section it will be discussed how the still existing weaknesses or bugs of the CRDT described earlier could be optimised or solved. Due to time constraints, it has not been possible to implement these optimisations, so therefore they will be described in this section.

The first optimisation is regarding the tree size. As mentioned before, the tree size could extremely grow once the document size improves. The tree size could drastically be reduced by storing a string in a node instead of a separate character in each node. However, when a user wants to insert in that string, the string still needs to be split in two nodes. Research has already been done about this in the following paper [30].

Another way to reduce the tree size, is to perform cleanups once no user is editing the document. Such clean up could function as a garbage collector and could remove tombstones from the tree, remove split references and remove all merge references from the CRDT by adding them to the block the refer to. Cleaning up the merge references will also optimise the previous described weakness regarding merges, since after every clean up, there are no merged blocks in the tree anymore. Cleanups will also optimise the split operation, since tree size will be reduced, less content has to be copied to the new block.

To optimise the searchBlock algorithm, a different algorithm could be used to find the blocks. Instead of using BFS to find the block, a CRDT could keep track of a map containing the blockId and the path to the block. This would optimise the searchBlock algorithm to O(1). To make this algorithm work however, one should ensure that the map will be updated correctly after every block operation.

The last approach for solving the circular references is to immediately remove the edge which has the highest timestamp in the circle. However, this approach is rather ineffective for solving more than one circular reference, since multiple CRDTs of multiple replicas can form different cycles, which will then converge to different circular references. An approach which might work is to save all merge references and construct a directed graph which contains all the references. The only problem left to solve, is to commonly find the same path with all CRDTs, including all the connected nodes (if possible). Since each replica contains the same references, if a common path can be found, then all replicas would merge the same blocks and therefore the replicas would converge.

As described in the previous chapter, the current bug with the merge and split operation, is that some merge nodes might be redirected since it is inserted after a split node. This however could be undesired behaviour, but also desired behaviour for some cases. When a block is already split, the merge with that block would be redirected to the new block created by the split operation. This is undesired since the block was already split and the expected behaviour would be to merge that block. To solve this problem however, some distinction should be made for whether something was done before the split operation or after. When this distinction can be made, this problem can be solved.

It is also recommended to research other types of CRDT structures. For this project Logoot was chosen as the base structure, but researching other structures could lead to more insight in the functioning of CRDTs and might be beneficial for the merge operation and increasing performance.

7.4. Prototype to Production

To make this prototype production-ready, first of all the current known bugs must be resolved. After that, one should use a method to prove the correctness of the current implementation of the CRDT. This could be done as described earlier by creating all different permutation. Once, the current functionality of the CRDT has been proven correct, nested blocks should be implemented. Finally, benchmarks should be used to check whether the mentioned optimisations in the section above are indeed necessary for Fontos purpose. If necessary, these optimisations should be implemented as well. Based on our current experience with implementing CRDTs, it could be expected that it would take about 10 weeks to achieve the above.

7.5. Ethical Implications

The use of real-time collaborative editing in the editor of Fonto does not introduce new security issues or privacy concerns. Currently, companies using the product of Fonto host the product on their own servers, so anyone who is not connected to the internet of the company or using a virtual private network, cannot access the documents on their database. Therefore, they cannot use the Fonto editor to edit these documents. Using CRDTs would not change this, since one should still be connected to the internet of the company to get access to the documents. Therefore it can be concluded that there are no ethical implications if Fonto would introduce real-time collaborative editing to their editor. Other ethical implications which can occur when using real-time collaborative editing in the Fonto editor are not caused by the CRDT itself, but by the implementation of the Fonto editor. One example is tracking document history and which user made specific changes. This ethical implication is not a part of the CRDT, but of the Fonto editor and how they decide to implement their editor.

8

Conclusion

During this research project, a CRDT has been built as a proof of concept to see whether CRDTs can be used to enable collaborative editing of block-based text in Fonto's editor.

8.1. Our CRDT implementation

After verifying the CRDT implementation, the following requirements have been met: inserting characters in blocks, deleting characters in blocks, inserting blocks, deleting blocks, moving blocks, splitting blocks, and the offline support for these operations. On the contrary, the merge operation is implemented, but does not function as expected for a small set of edge cases. There exists the possibility of having circular merges, due to offline support, and a possibility for content loss when performing multiple split and merge operations on the same block.

Implementing the merge functionality was the most time consuming functionality to implement, because merging blocks in combination with moving and splitting blocks and offline support has a huge amount of edge cases which should be covered. It could not be concluded yet if the current approach for merging blocks can cover all edge cases for merging blocks in combination with all other operations.

Regarding all other operations, it could be concluded that either the implementation is already efficient or that an optimisation could be implemented to make the algorithm more efficient. However, no benchmarks have been performed to check how fast this implementation is with different tree sizes. Therefore, it could not be concluded yet at what tree size the CRDT does not perform operations in a reasonable time.

8.2. Future Implications

This project has shown that it is indeed possible to use CRDTs for real-time collaborative editing in block-based text. Although this might be a small step for real-time collaborative editing in the Fonto product, this prototype showed that there are possibilities in using CRDTs for block-based text when implementing real-time collaborative editing in other block-based editors.

The first next step, is to research merging blocks and find a solution which can solve all edge cases. After that the time and space complexity of the CRDT should be measured using benchmarks and optimised if necessary. When it has been proven that the CRDT is indeed fast enough for a smooth user experience, one could look at adding other functionality such as undo and redo of operations as well as catching user intent to improve the editing experience.

9 Project Evaluation

To reflect on the project, the overall planning and weekly plannings made over the entire project were very helpful for conducting the research as it gave a good indication on where the difficulties were and whether we were ahead or behind on planning.

The orientation and research phase all went perfect according to the plan, the research was done in time, an already existing CRDT and a text editor which we could extend and combine with our CRDT were found.

During the implementation and modifying phase, we deviated a little bit from the original plan. Instead of implementing all functionality in the implementation phase and using the modifying phase to improve our CRDT and implement extra functionality, all four weeks were used for implementation where the last week was solely used for the merge functionality and bug fixes and the first three weeks for creating a test environment and implementing all required functionality. During the implementation phase we got a little bit overzealous. At the start it went really well, implementing insert and deletion of characters, insertion, deletion and moving of blocks all went fairly easy. So after week two of implementing we thought we maybe would even have a spare week for implementing extra functionality. During the last two weeks however, we discovered that implementing the merge functionality in combination with all other functionality was very complex. Additionally, when working on the merge operation, we learned more about CRDTs and how to implement one. Other bugs were found which also needed to be fixed. We underestimated the complexity of the merge operations and all the problems it could give in our design.

During the documentation phase all our findings, recommendations, design decisions and implementation have been documented in this report.

What we would do better is when we discover a new problem, instead of trying to immediately fix it, we should do some more extra research into the topic. For merge we used three approaches, and after implementing an approach another flaw was found with that approach. It would have been better if we would have taken more time to research the topic of merging blocks using CRDT and have a better overview of all the issues that could arise. For the rest of the project we are happy how it went and are satisfied with the result.

Bibliography

[1] Mehdi Ahmed-Nacer. Abstract unordered and ordered trees crdt, December 2011. URL https://www.academia.edu/26308619/Abstract_unordered_and_ordered_trees_CRDT.

[2] Mehdi Ahmed-Nacer, Claudia-Lavinia Ignat, Gérald Oster, Hyun-Gul Roh, and Pascal Urso. Evaluating CRDTs for Real-time Document Editing. In ACM, editor, *11th ACM Symposium on Document Engineering*, pages 103–112, Mountain View, California, United States, September 2011. doi: 10.1145/2034691.2034717. URL https://hal.inria.fr/inria-00629503.

[3] Mehdi Ahmed-Nacer, Pascal Urso, Valter Balegas, and Nuno Preguiça. Merging OT and CRDT Algorithms. In *1st Workshop on Principles and Practice of Eventual Consistency (PaPEC)*, Amsterdam, Netherlands, April 2014. doi: 10.1145/2596631.2596636. URL https://hal.inria.fr/hal-00957167.

[4] Baquero, Carlos, Almeida, Paulo Sergio, and Ali. Pure operation-based replicated data types, Oct 2017. URL https://arxiv.org/abs/1710.04469.

[5] Loïck Briot, Pascal Urso, and Marc Shapiro. High Responsiveness for Group Editing CRDTs. In *ACM International Conference on Supporting Group Work*, Sanibel Island, FL, United States, November 2016. doi: 10.1145/2957276.2957300. URL https://hal.inria.fr/hal-01343941.

[6] Murat Demirbas, Marcelo Leone, Bharadwaj Avva, Deepak Madeppa, and Sandeep S. Kulkarni. Logical physical clocks and consistent snapshots in globally distributed databases, 2014.

[7] Paul Frauzee. Crdt notes, 2015. URL https://github.com/pfrazee/crdt_notes.

[8] Simon Guindon. Convergent replicated data types, n.d. URL http://simongui.github.io/distributed-systems/crdt.html.

[9] Andrew Herron. To ot or crdt, that is the question, Jan 2020. URL https://www.tiny.cloud/blog/real-time-collaboration-ot-vs-crdt/.

[10] Joe Honour. Distributed systems: Physical, logical, and vector clocks, Dec 2018. URL https://levelup.gitconnected.com/distributed-systems-physical-logical-and-vector-clocks-7ca989f5f780.

[11] Dmitry Ivanov and Nami Naserazad. Practical demystification of crdt (lambda days 2016), 2016.

[12] Ajay Kshemalyani and Mukesh Singhal. Logical time, 2008. URL https://www.cs.uic.edu/~ajayk/Chapter3.pdf.

[13] Brice Nédelec, Pascal Molli, Achour Mostefaoui, and Emmanuel Desmontils. LSEQ: an Adaptive Structure for Sequences in Distributed Collaborative Editing. In *13th ACM Symposium on Document Engineering (DocEng)*, pages 37–46, Florence, Italy, September 2013. doi: 10.1145/2494266.2494278. URL https://hal.archives-ouvertes.fr/hal-00921633. 10 pages.

[14] Mikhail Nesterenko. Managing physical clocks in distributed systems, 2002. URL http://vega.cs.kent.edu/~mikhail/classes/aos.f02/l06physicalclocks.PDF.

[15] Petru Nicolaescu, Kevin Jahns, Michael Derntl, and Ralf Klamma. Near real-time peer-to-peer shared editing on extensible data types. pages 39–49, 11 2016. doi: 10.1145/2957276.2957310.

[16] Gérald Oster, Pascal Urso, Pascal Molli, and Abdessamad Imine. Real time group editors without Operational transformation. Research Report RR-5580, INRIA, 2005. URL https://hal.inria.fr/inria-00071240.

[17] Gérald Oster, Pascal Molli, Pascal Urso, and Abdessamad Imine. Tombstone Transformation Functions for Ensuring Consistency in Collaborative Editing Systems. In *IEEE Conference on Collaborative Computing: Networking, Applications and Worksharing - CollaborateCom 2006*, Collaborative Computing: Networking, Applications and Worksharing, 2006. CollaborateCom 2006. International Conference on, pages 1–10, Atlanta, Georgia, USA, November 2006. IEEE. doi: 10.1109/COLCOM.2006.361867. URL https://hal.inria.fr/inria-00109039. http://ieeexplore.ieee.org/.

[18] Nuno Preguiça, Joan Manuel Marquès, Marc Shapiro, and Mihai Le□ia. A commutative replicated data type for cooperative editing. In *29th IEEE International Conference on Distributed Computing Systems (ICDCS 2009)*, pages 395–403, Montreal, Québec, Canada, June 2009. IEEE Computer Society. doi: 10.1109/ICDCS.2009.20. URL https://hal.inria.fr/inria-00445975.

[19] Michal Ptaszek. Chat service architecture: Persistence, Jan 2016. URL https://technology.riotgames.com/news/chat-service-architecture-persistence.

[20] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Research Report RR-7506, Inria – Centre Paris-Rocquencourt ; INRIA, January 2011. URL https://hal.inria.fr/inria-00555588.

[21] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free Replicated Data Types. Research Report RR-7687, Inria – Centre Paris-Rocquencourt ; INRIA, August 2011. URL https://pages.lip6.fr/Marc.Shapiro/papers/RR-7687.pdf.

[22] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Research Report RR-7506, Inria – Centre Paris-Rocquencourt ; INRIA, January 2011. URL https://hal.inria.fr/inria-00555588.

[23] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Research Report RR-7506, Inria – Centre Paris-Rocquencourt ; INRIA, January 2011. URL https://hal.inria.fr/inria-00555588.

[24] D. Sun and C. Sun. Context-based operational transformation in distributed collaborative editing systems. *IEEE Transactions on Parallel and Distributed Systems*, 20(10):1454–1470, 2009.

[25] David Sun, Steven Xia, Chengzheng Sun, and David Chen. Operational transformation for collaborative word processing. In *Proceedings of the 2004 ACM Conference on Computer Supported Cooperative Work*, CSCW '04, page 437–446, New York, NY, USA, 2004. Association for Computing Machinery. ISBN 1581138105. doi: 10.1145/1031607.1031681. URL https://doi.org/10.1145/1031607.1031681.

[26] Bartosz Sypytkowski. An introduction to state-based crdts, Dec 2017. URL https://bartoszsypytkowski.com/the-state-of-a-state-based-crdts/.

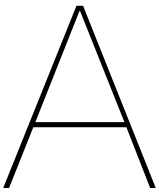
[27] Stéphane Weiss, Pascal Urso, and Pascal Molli. Logoot: A Scalable Optimistic Replication Algorithm for Collaborative Editing on P2P Networks. In *29th IEEE International Conference on Distributed Computing Systems - ICDCS 2009*, 2009 29th IEEE International Conference on Distributed Computing Systems, pages 404–412, Montreal, Canada, June 2009. IEEE. doi: 10.1109/ICDCS.2009.75. URL https://hal.inria.fr/inria-00432368.

[28] Stéphane Weiss, Pascal Urso, and Pascal Molli. Logoot: a p2p collaborative editing system. Jan 2008.

[29] Michael Whittaker. Crdt visualization, Nov 2016. URL https://github.com/mwhittaker/crdts#state-based-pn-counter.

[30] Weihai Yu. A string-wise crdt for group editing. pages 141–144, 10 2012. doi: 10.1145/2389176.2389198.

[31] Weihai Yu, Luc André, and Claudia-Lavinia Ignat. A CRDT Supporting Selective Undo for Collaborative Text Editing. In Alysson Bessani and Sara Bouchenak, editors, *15th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS)*, volume LNCS-9038 of *Distributed Applications and Interoperable Systems*, pages 193–206, Grenoble, France, June 2015. Springer International Publishing. doi: 10.1007/978-3-319-19129-4_16. URL https://hal.inria.fr/hal-01246212.



Info Sheet

Title: CRDTs for Fonto
Client: Fonto
Presentation date: 2 July 2020

Problem description:

Fonto has a rich What You See Is What You Mean text editor for structured XML content in which the client can easily create structured text with the use of an easy to understand user interface. Currently, no real-time collaboration functionality is provided for this rich editor, which led to this graduation project. Fonto wanted to know if CRDTs could be used to make collaborative editing of structured content possible. The core challenge of this project is that CRDTs are relatively new, whereas little to no information was found on block operations. Going through the research phase, lots of insights were acquired on how CRDTs work. This allowed us to understand the core of CRDTs and helped us more in problem-solving. A proof of concept prototype has been developed for showing whether CRDTs could potentially be used for the client. This prototype used automated tests to verify whether the block operations had been implemented correctly. Also, the created CRDT has been connected to a block-based text-editor for manual testing and visualising the behaviour. Whilst developing this prototype, the Scrum methodology was used, whereas each week, a list of tasks was picked from the backlog, which was created at the start of the project. The created prototype showed that most block operations can be supported by this current implementation. Only the merge functionality contained inconsistencies. Other challenges were guaranteeing convergence which had been solved by introducing timestamps for certain operations. Our recommendations are to do more research to see if the merge functionality can be supported by CRDTs and to optimise the time complexity.

Introduction team members

Quentin Lee
Interest: Programming, Data science, Web development
Role & Contribution: Back-end developer, Front-end developer, and Tester

Martin Li
Interests: Software engineering
Role & Contribution: Back-end developer, Tester, Scrum Master, and Contact person

Cas van Rijn
Interests: Embedded systems, User interaction and Web development
Role & Contribution: Back-end developer and Tester

Wang Hao Wang
Interests: Algorithms and Optimisation
Role & Contribution: Back-end developer and Tester

Client, advisors, and coach

Name of the client: Bert Willems (Fonto)
Names of the advisors: Stef Busking (Fonto) and Martin Middel (Fonto)
Name of the TU coach: Bart Gerritsen

Contact person

Name and email of the contact person: Martin Li, martinchunho@gmail.com

The final report for this project can be found at: https://repository.tudelft.nl/

B

Legend

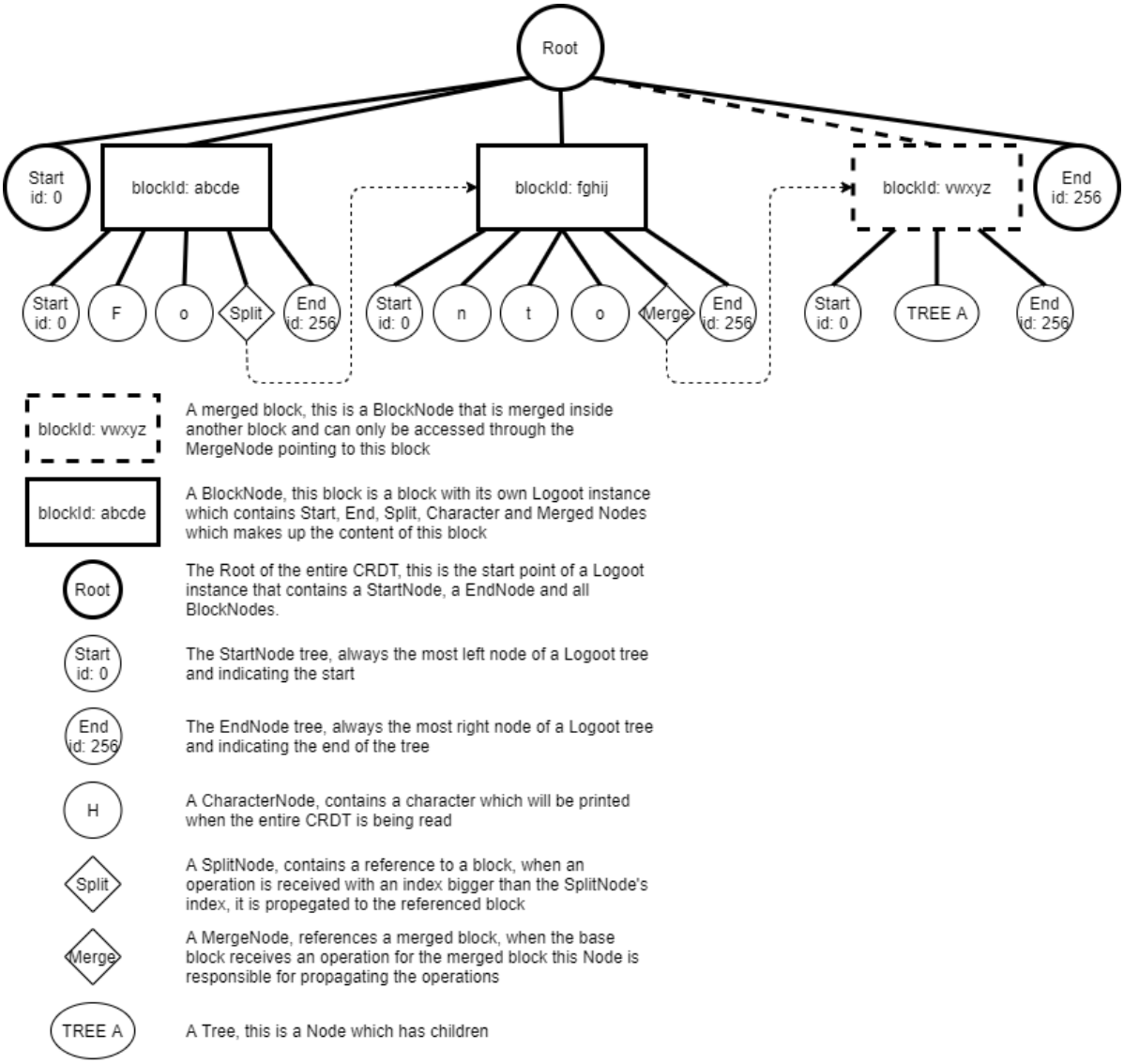


Figure B.1: Legend

C

Project Description

Bachelor project: CRDTs for Fonto

Researchers: Martin Li, Wang Hao Wang, Cas van Rijn, Quentin Lee
Client: Bert Willems (Fonto)
Supervisor: Bart Gerritsen
Advisor: Stef Busking (Fonto)

Introduction

At Fonto we aim to make editing XML as easy as editing a Word document. Our editor provides a rich text, what you see is what you mean (WYSIWYM) editing experience which can be configured to fit any XML schema. When multiple authors collaborate on the same document, Fonto currently relies on the CMS to provide locking functionality to ensure only one author can work on any part of the document at a time. To provide for a more intuitive editing experience, we would like to lift this limitation in the future and enable true concurrent editing, as seen in Google Docs. Furthermore, in our work to enable IME[1] input in Fonto we have identified a very similar problem where both the IME and Fonto will make concurrent edits on the same state, which have to be resolved to avoid discarding either. As both of these problems involve handling concurrent editing operations into a single consistent state, we believe a single solution may fit both cases.

We have previously investigated several approaches to dealing with concurrent edits, and have concluded that CRDTs, or Concurrent Replicated Data Types, provide the best fit. However, it appears there is no CRDT currently that fits our content models. In Fonto, content is stored as XML trees. In the process of editing, text within the XML tree will frequently be split, joined with other text and moved around together with the surrounding XML elements. While CRDTs exist that model documents as a run of possibly annotated text, we have not found any CRDTs that allow blocks of text to be freely split, joined and moved around while maintaining the results of concurrent edits within each block.

Requirements

1. Develop a prototype implementation of a CRDT that supports the following operations: inserting and deleting text, splitting and merging blocks of text, moving blocks around.

[1] Input Method Editor, a common way to insert characters from non-western scripts.

2. Develop tooling for manual and automated testing of the CRDT to confirm its correctness, robustness and user-friendliness in terms of preservation of user intent.
3. Document implementation and report findings.

Suggested approach

1. Investigate CRDT literature. Pick one or more existing CRDTs on which to base the work.
2. Implement prototype of the base CRDT for text-level operations, including a simplified editor environment.
3. Implement tooling to test the CRDT implementation.
4. Design and implement extensions to the CRDT to support block-level operations.
5. Evaluate and document your results.

Possible extensions

- **Extend the CRDT to implement other aspects of the XML data model, such as attributes.** In addition to movable blocks of text, representing actual XML documents requires modeling several other aspects, including the ability to nest elements and assign attributes. If time allows, the students can investigate and experiment with ways to extend the CRDT to model such aspects.
- **Automatically reorder operations into a sequence that minimizes the number of times the "current author" switches.** Fonto is usually integrated with an existing content management system that is not aware of collaboration in the true concurrent sense. Several of these CMSes track edits to a document in terms of revisions, created whenever a new author makes changes. We would like to investigate ways to minimize the number of revisions created for concurrent editing sessions by taking advantage of the possibility offered by CRDTs to reorder concurrent but independent operations.
- **Investigate approaches for dealing with restrictions placed on the document.** XML documents are usually subject to restrictions imposed by a schema. This includes not only restrictions on which elements may occur where, but also on things like the number of child elements under some parent. Even if individual edits obey such restrictions, the result of merging concurrent edits may not. If time allows, the students can investigate approaches to resolve such violations in a way that converges to the same result for all users in the editing session.

Other information

Fonto (formerly Liones) has more than 7 years of experience in offering internships to students of The Hague University of Applied Sciences. We have helped over dozens of students with their graduation over the years, and many of them are still employed by us today. This bachelor project is our first project at TU Delft, but we are super enthusiastic and convinced that we can offer them a great challenge with the best supervision possible.

The students receive an internship allowance of EUR 450.00 per month. The students can work with us on location.

References

1. CRDTs and the Quest for Distributed Consistency => https://www.infoq.com/presentations/crdt-distributed-consistency/
2. CRDTs in Production => https://www.infoq.com/presentations/crdt-production
3. A comprehensive study of Convergent and Commutative Replicated Data Types => https://github.com/papers-we-love/papers-we-love/blob/master/data_replication/a-comprehensive-study-of-convergent-and-communative-replicated-data-types.pdf
4. Notes on Splicing CRDTs for Structured Hypertext => https://lord.io/blog/2019/splicing-crdts/, https://www.figma.com/blog/how-figmas-multiplayer-technology-works/#the-details-of-figma-s-multiplayer-system
5. https://blog.acolyer.org/2019/03/11/efficient-synchronisation-of-state-based-crdts/amp/
6. Evaluating CRDTs for real-time document editing https://dl.acm.org/citation.cfm?id=2034717 Let Bert known if you need the PDF.
7. Real Differences between OT and CRDT in Correctness and Complexity for Consistency Maintenance in Co-Editors => https://arxiv.org/ftp/arxiv/papers/1905/1905.01302.pdf
8. Delta State Replicated Data Types => https://arxiv.org/pdf/1603.01529.pdf
9. Supporting String-Wise Operations and Selective Undo for Peer-to-Peer Group Editing => https://dl.acm.org/citation.cfm?doid=2660398.2660401
10. CRATE: Writing Stories Together with our Browsers => https://dl.acm.org/citation.cfm?doid=2872518.2890539
11. Real time group editors without Operational transformation - Gérald Oster, Pascal Urso, Pascal Molli, Abdessamad Imine => https://hal.inria.fr/inria-00071240/document

12. LSEQ: an adaptive structure for sequences in distributed collaborative editing=> https://dl.acm.org/citation.cfm?doid=2494266.2494278
13. Logoot: a P2P collaborative editing system - Stéphane Weiss, Pascal Urso, Pascal Molli => https://hal.inria.fr/inria-00432368/document
14. Replicated abstract data types: Building blocks for collaborative applications - Hyun-Gul Roh, Myeongjae Jeon, Jin-Soo Kim, Joonwon Lee => http://csl.skku.edu/papers/jpdc11.pdf
15. Replicated data types: specification, verification, optimality => https://dl.acm.org/citation.cfm?doid=2535838.2535848
16. Making operation-based CRDTs operation-based => https://dl.acm.org/citation.cfm?doid=2596631.2596632
17. Conflict-Aware Replicated Data Types => https://arxiv.org/abs/1802.08733v1
18. Real-time Collaborative Editing with CRDTs

D

Project Plan

Project Plan: Conflict-free replicated data types (CRDTs) for Fonto

Researchers: Martin Li, Wang Hao Wang, Cas van Rijn, Quentin Lee
Client: Bert Willems (Fonto)
Advisor: Stef Busking (Fonto)
TU Delft coach: Bart Gerritsen

22 April, 2020

Introduction

For the graduation project: CRDTs for Fonto, our task is to find and implement a prototype of a CRDT variant, with modifications, which would meet the minimum requirements given by Fonto. For this to be realized, a project plan is made which describes how the upcoming ten weeks will be spent. The Project Plan will serve as an agreement between the project leader, the formal client, and other personnel associated with or affected by the project. The following chapters will provide an overview of the project and the phases of development.

Project Description

Fonto aims to make editing XML as easy as editing a Word document. The Fonto editor provides a rich text, what you see is what you mean (WYSIWYM) editing experience which can be configured to fit any XML schema. When multiple authors collaborate on the same document, Fonto currently relies on the CMS to provide locking functionality to ensure only one author can work on any part of the document at a time. To provide for a more intuitive editing experience, Fonto would like to lift this limitation in the future and enable true concurrent editing, as seen in Google Docs. The difference in complexity between Google Docs and Fonto lies in the support of many extended features that Fonto supports, e.g. support for arbitrary XML schemata and the ability for Fonto's partners to write any custom mutation to manipulate the XML model.

Some of the limitations of using conflict-free replicated data types (CRDTs) with XML editing is that XML is a tree-based structure which needs to be kept intact. In the editor of Fonto, it is possible to swap blocks of text around. This can result in loop references when two parts are swapped with each other and start pointing to each other as parent, resulting in a loss of text.

The following questions are defined to facilitate this research:

The main research question for this project is:
How can CRDTs be used to implement concurrent editing of XML files using a text-based editor?

Subquestion:

1. Why use CRDT over other collaborative editing algorithms?
2. What types of CRDTs are there?
3. Which CRDT is the best fit for Fontos purpose?

Deliverables

During this project, there are mandatory and optional deliverables. These deliverables are either code, a prototype, document or a presentation.

Mandatory deliverables:

- Project plan (22-04)
    - This document, a plan of execution for the project during the upcoming 10 weeks.
- Research report (08-05)
    - Report that summarizes the results of the research phase and will elaborate on the approach of the rest of the project based on those results.
- Final report (TBD)
    - The final report for the entire project. This report will contain the research report mentioned above, a problem definition and analysis. The design of the solution, implementations (if applicable), conclusions, discussions and recommendations based upon the findings during the project.
- Bachelor Project Info Sheet (TBD)
    - A standard A4 sheet containing a summary and the highlights of the project.
- Code for review by the Software Improvement Group (First submission 31-05, Second submission 14-06)
    - Two times during the project we will submit our code to the Software Improvement Group of the TU Delft. This will be all our code we have written up to that point.
- Final Presentation (TBD)
    - A 30-minute presentation at the end of the project containing motivation, process, evaluation, conclusion, and demo.

Optional deliverables:

- A prototype CRDT where a collaborative editing of a tree-based document containing text is possible and the structure is kept intact when moving text around. The aim is to add as much as possible of the following functionalities:
    - adding and removing text
    - splitting of text
    - merging of text
    - moving blocks of text while preserving a tree model and without losing or duplicating text

Project Roadmap

Throughout the whole project we will document our findings and work on our report.

Week 1-2 Research Phase
The first two weeks will be focused on research. In this research, research will be done on different CRDTs, what CRDTs already exist and what CRDT is most suitable for our implementation. The findings will be written in the research report.

Week 3 Investigation Phase
After researching the papers, based on our findings a selection of CRDTs is made which we will dive further into. This means that the selected CRDTs will be reviewed on what they are capable of, what the limitations are, how it fits within Fonto's model, and the degree of difficulty. We will also start looking into existing implementations of CDRTs fitting for our purpose and see if we can use those. The main goal of this week is to find an already existing CRDT, which is sufficient enough for us to build further on. At the end of the investigation phase we will select which CDRT we are going to use and have defined a strategy for implementation.

Week 4-6 Implementation Phase
After selecting a CRDT, we will fork this CRDT, modify the implementation to our needs and add functionality if needed. In addition, we will write tests to validate the functionality of the CRDT using the Mocha testing framework. At the end of this phase, we want to have a CRDT that satisfies the minimum requirements. (inserting and deleting text, splitting and merging blocks of text, moving blocks around)

Week 7-8 Modifying Phase
Create a text editor in which we will implement our CRDT for manual testing. Improve our CRDT and implement extra functionality if there is enough time left.
Some of the extra functionality we could implement is:

- Extend the CRDT to implement other aspects of the XML data model, such as attributes.
- Automatically reorder operations into a sequence that minimizes the number of times the "current author" switches.
- Investigate approaches for dealing with restrictions placed on the document.

Finally, in the modifying phase, we will start writing the report, based on the findings from the implementation phase.

Week 9 - 10 Report and Presentation
In the last week, we will finish the report and make the final modifications on the CRDT, if necessary. Furthermore, we will work on the final presentation and give the final presentation in week 10.

Communication Plan

For, at least, the first two weeks we will be working remotely from home, because of the COVID-19 pandemic. We will mainly use WhatsApp and Slack for our internal communication.

For communication with the client, we use Slack for messaging and Google Meet for meetings. Every business day at 9:15 AM a daily stand-up is planned whereas all members are required to pitch about what they have done the past day and how they will continue to work for the next day. Additionally, every Friday, there will be a meeting with the client to discuss about encountered problems and updates about the progress. Whenever urgent problems arise, a meeting will be planned as soon as possible to resolve those problems.

For communication with the TU Delft coach, we have a weekly Skype meeting where we will discuss our progress of the project.

For the planning of the project, we will use the scrum methodology. Every week we will define new goals to achieve at the end of the week. At the end of each week we will have a retrospective in which we will review our progress and adjust our goals accordingly.