Université de Montréal

Framework for Real-time collaboration on extensive Data Types using Strong Eventual Consistency

par
Constantin Masson

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Mémoire présenté à la Faculté des études supérieures
en vue de l'obtention du grade de Maître en sciences (M.Sc.)
en informatique

Decembre, 2018

Université de Montréal
Faculté des études supérieures

Ce mémoire intitulé:

Framework for Real-time collaboration on extensive Data Types using Strong Eventual Consistency

présenté par:

Constantin Masson

a été évalué par un jury composé des personnes suivantes:

Famelis Michalis,	président-rapporteur
Eugene Syriani,	directeur de recherche
Boyer Michel,	membre du jury

Mémoire accepté le: . . . .

ABSTRACT

Real-time collaboration is a special case of collaboration where users work on the same artefact simultaneously and are aware of each other's changes in real-time. Shared data should remain available and consistent while dealing with its physically distributed aspect. Strong Consistency is one approach that enforces a total order of operations using mechanisms, such as locking. This however introduces a bottleneck. In the last decade, algorithms for concurrency control have been studied to keep convergence of all replicas without locking or synchronization. Operational Transformation and Conflict-free Replicated Data Types (CRDT) are widely used to achieve this purpose. However, the complexity of these strategies makes it hard to integrate in large software, such as modeling editors, especially for complex data types like graphs. Current implementations only integrate linear data, such as text. In this thesis, we present CollabServer, a framework to build collaborative environments. It features a CRDTs implementation for complex data types such as graphs and gives possibility to build other data structures.

Keywords: Concurrency Control, Concurrent Algorithms, Distributed Systems, Optimistic concurrency control, Software Engineering, Shared Data, Strong Eventual Consistency

RÉSUMÉ

La collaboration en temps réel est un cas spécial de collaboration où les utilisateurs travaillent sur le même élément simultanément et sont au courant des modifications des autres utilisateurs en temps réel. Les données distribuées doivent rester disponibles et consistant tout en étant répartis sur plusieurs systèmes physiques. "Strong Consistency" est une approche qui crée un ordre total des opérations en utilisant des mécanismes tel que le "locking". Cependant, cela introduit un "bottleneck". Ces dix dernières années, les algorithmes de concurrence ont été étudiés dans le but de garder la convergence de tous les replicas sans utiliser de "locking" ni de synchronisation. "Operational Transformation" et "Conflict-free Replicated Data Types (CRDT)" sont utilisés dans ce but. Cependant, la complexité de ces stratégies les rend compliquées à intégrer dans des logicielles conséquents, comme les éditeurs de modèles, spécialement pour des data structures complexes comme les graphes. Les implémentations actuelles intègrent seulement des data linéaires tel que le texte. Dans ce mémoire, nous présentons CollabServer, un framework pour construire des environnements de collaboration. Il a une implémentation de CRDTs pour des data structures complexes tel que les graphes et donne la possibilité de construire ses propres data structures.

Keywords: Algorithme de concurrence, Data distribuée, Génie logiciel Gestion de concurrence, Strong Eventual Consistency, Système distribués

CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF APPENDICES

LIST OF ABBREVIATIONS

CmRDT	Commutative Replicated Data Type
CvRDT	Convergent Replicated Data Type
CRDT	Conflict Free Replicated Data Type
EC	Eventual Consistency
LWW	Last Writer Wins
RTC	Real Time Collaboration
SC	Strong Consistency
SEC	Strong Eventual Consistency

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my advisor Eugene Syriani for the continuous support of my study and research, for his patience, motivation, enthusiasm, and immense knowledge. He gave me the opportunity to work as his teaching assistant, which was a great and constructive experience.

I thank my fellow labmates in the GEODES Group, for the stimulating discussions and for all the fun we have had in the last two years. Thanks to Thomas for being my awesome desk-mate during the last summer semester. I really enjoyed the discussions we had about code designs and software architectures, which helped me to design CollabServer, in particular its networking architecture. Thanks to Nicola, Samuel, and Remi for their devotion in the student life at UdeM and for all the fun they gave us. I would like to thank Céline Bégin for her kindness and her efficiency as well as the whole UdeM administration team. Thanks to Baila and Brice for being my "poutine eater" partners during these nights of work in the lab, and thanks to "Blanche Neige Restaurant" for being our official poutines food provider. Thanks to Adrien for his help on the off-topics projects I have been working on, such as engine programming and 3D rendering. I would like to sincerely thank Michalis Famelis for his huge help in writing my cover letter and resume, that significantly helped me getting my post-master job. I thank Edouard for the really interesting discussion we have had. Thanks to Raouf for taking the time to share his immense knowledge about Linux and systems administration. Thanks to the coffee machine that gave its best during these two years to provide us with a daily coffee. Thanks to Gnu and Panda, the two plushes on my desk, for listening to my programming reflections that would have stayed in my mind without them.

Last but not the least, I would like to thank all my family, my parents, and my friends for all the support they gave me during the whole adventure. Thanks to my SSD for crashing when I needed it, fortunately, this attempt to destroy hours of work was a fail, thanks to the git backups. Thanks to the cats, dogs, raccoons, and pandas for being super funny animals that distracted me during the breaks.

CHAPTER 1

INTRODUCTION

1.1 Context

In computer science, a distributed system is a network of independent computers that appears to the users as one unique system. Data and computing resources are replicated on multiple remote locations, which gives several advantages, such as fault tolerance, performance, and availability for remote users distributed geographically. In case of node failure, a transparent mechanism allows to automatically restore the system back to a valid state. Users are not aware of distribution and the whole system is seen as a unique entity. Distributed systems may require a consensus mechanism to deal with concurrency. Paxos [37] and Raft [46] are examples of common consensus algorithms in distributed systems.

Collaboration uses the same notions to provide its users a shared data that may be edited concurrently over the network. Real-time collaboration is a special case of collaboration where users work on the same document simultaneously and are aware of other changes in real-time. However, due to the network aspect of collaboration, operations may be applied with unexpected latency due to some replicates. Therefore collaborative editing faces the technical challenge of remaining available and consistent while dealing with physically distributed data, as stated by the CAP theorem [54] described in Section 2.1.1. In any case, at the end of collaboration, all replicates should converge to the same state. Different approaches exist in order to keep convergence. Strong Consistency enforces a total order of operations [36] so that all users see the exact same execution order, but requires a consensus mechanism. Locking [6] is a simple approach that has been used in collaborative environment [30, 45]. It allows only one user to concurrently edit a specific data element at a time, so that conflicts are removed altogether. It is designed for client/server architectures. Locking introduces an important bottleneck since users have to wait for a resource to be released before applying any modification. To overcome this issue, several algorithms for concurrency control without locking have been studied [17, 21, 55].

Operational Transformation (OT) [18] resolves the locking problem by applying transformations on received operations. Concurrent editing is supported and users may collaborate in real-time on the same document. It is designed for client/server architectures. GoogleDoc is an example of large-scale industrial software that uses OT algorithm. Conflict-free Replicated Data Types (CRDTs) [55] is another concurrency control algorithm that focuses on data design so that no locking is required. It is well suited for peer-to-peer architectures but may be used for client/server architectures as well.

Lock-free concurrency control algorithm implementations generally support only linear data, such as text (e.g., GoogleDoc, Etherpad). More complex and sophisticated data structures, like graph-based specifications, are hard to design and implement. The original authors of CRDT algorithm introduce an example of more advanced tree data structure, called TreeDoc [50]. Moreover, they discuss theoretically graph data types in their complementary CRDTs study [57]. There are many software applications that require collaboration of more complex data. This is the case of Model-Driven Engineering (MDE) [35, 53] which uses models with strong semantics. MDE models are typically encoded as graphs due to the complexity of there relations [62].

1.2 Problem Statement and Thesis Proposition

Building collaborative environments is not an easy task. One has to design and integrate features required by collaborative environments, such as network architecture, concurrency algorithm (e.g., locking, OT, CRDT), user awareness (e.g., list of current collaborators) and other possible features (e.g., undo/redo stack [11]). To ease these software design and architecture choices, we present a feature diagram for collaborative environments [42]. It may be used as a base of reflection in order to build any collaborative environment.

Lock-free concurrency control algorithms such as OT and CRDT are well-documented in the literature [17, 55], but their current implementations suffer from a growing complexity as soon as the shared data type becomes more sophisticated. Therefore, existing software have their own implementations specific for their data structures and general purpose implementations of these algorithms are still rare and uncommon (e.g., Google-RealtimeAPI [26], YATA [44]). To achieve collaboration using such algorithms, one has to do his own implementation, which is a complex and error-prone task. Our framework tries to minimise this endeavor by providing a set of CRDTs primitives that allow the developer to integrate more advanced data types, such as graph.

1.3 Contributions

The goal of this thesis is to ease the creation of collaborative software by giving tools that hide its complexity. This thesis proposes a novel framework, called CollabServer, based on CRDT to collaborate on extensive data types. It extends the current CRDT approaches and implementations by providing a set of collaborative data primitives as building blocks to construct more complex data structures. This framework helps developers build new CRDT data structures fit for their purpose. We provide an efficient implementation in C++ available online [1].

1.4 Outline

This thesis is organized as follows. In Chapter 2, we introduce distributed systems principles and present existing work on real-time collaboration. In Chapter 3, we enumerate and describe the features required for a collaborative environment. In Chapter 4, we present our CRDT implementation used by CollabServer framework. In Chapter 5, we describe the architecture of the CollabServer framework along with implementation details. In Chapter 6, we discuss the evaluation of our solution. Finally, we conclude in Chapter 7.

[1] `https://github.com/geodes-sms/CollabServer/`

CHAPTER 2

STATE OF THE ART

In this chapter, we introduce the principles for distributed systems as well as some existing algorithms for collaboration.

2.1 Consistency in Distributed Systems

Shared data is distributed on several nodes. Therefore challenges in distributed systems/databases is to keep all nodes consistent. In this section, we are investigating the principles that operations should conform to when they manipulate distributed data. The goal is not only to ensure data consistency among the nodes, but also to satisfy the user experience: delays, consistency, intent, etc. These properties have been formally discussed by different principles for distributed data, such as CAP Theorem, ACID, and BASE principles.

2.1.1 CAP Theorem

The Brewer's theorem (a.k.a., CAP theorem) [7] states that it is impossible for a distributed data store to simultaneously provide more than two of the following three guarantees.

Consistency	Every read receives the most recent write or an error
Availability	Every request receives a (non-error) response without guarantee that it contains the most recent write
Partition Tolerance	The system continues to operate despite an arbitrary number of messages being dropped (or delayed) by the network

CAP theorem does not state that one has to choose two of these three but one has to choose between Consistency and Availability in case of network failure [8, 9]. However, in normal conditions (Network with no failure and acceptable latency), all three guarantees may be provided successfully. CAP has been criticized [34, 54] for this ambiguity it may introduce.

2.1.2 ACID properties

Database systems have data entries that may be created, queried, updated, and deleted. These are the four basic operations applicable on a database, and are designated by the CRUD acronym [33] (i.e., Create, Read, Update, Delete). ACID stands for Atomicity, Consistency, Isolation, and Durability. ACID properties are widely used in relational database. Any sequence of operations that satisfy ACID properties is called a transaction and represents a logical operation.

Atomicity means that a sequence of operations that constitute a transaction are all applied successfully, or none of them. This guarantee to see the transaction as a unique atomic operation and does not partially update the database in case of failure.

Consistency means that the transaction brings a valid state to a valid state, according to the integrity constraints imposed by the database. A transaction leading to an inconsistent state fails and is not applied. As an example, a transaction `T` may try to update an SQL entry that has value range constraint. If this transaction would break this constraint upon application, then the whole transaction fails so that it remains valid.

Isolation means that transaction does not interfere with other transactions executed concurrently. Moreover, concurrent transactions leave the database in a state that is the same as if they were applied sequentially.

Durability means that, upon completion, a transaction is guaranteed to be recorded in durable storage and is visible to other transactions. As an example, transaction should not be committed if only saved in a temporary buffer (power failure would lose this transaction and lead to inconsistency).

2.1.3 BASE properties

Traditionally, ACID has been used for relational databases. However, new unstructured database model, such as NoSQL database, goes in favor of BASE [10]. Such databases rely on a distributed data to provide high availability and flexibility. For this purpose, the ACID model becomes an overkill. The principles of BASE are Basic Availability, Soft State, and Eventual Consistency (EC). The former guarantees that data remains available even in the presence of multiple failures. This is achieved by using a distributed approach of the data that is replicated on several nodes. Soft State stipulates that the database does not have to be in a consistent state all the time and, because of EC, may change even once it stops receiving operations. Eventual Consistency is the last but not least BASE property. Basic Availability through a distributed model allows data to diverge at some replicas. EC guarantees that, at some point in the future, data from all replicas will converge to a consistent state. EC is explained in further details in Section 2.1.5.

2.1.4 Strong Consistency

Strong Consistency (SC) goes along with ACID properties, concurrency is resolved by processing operations sequentially so that ACID properties are guaranteed. Operations are applied in a sequential order by a central consensus and sent to every replica upon completion [36]. This ensures that only a consistent state is observed by all users at all times but introduces needs for locking [30, 31, 45]. One central consensus server is responsible of queuing operations in a specific order to process them sequentially in that very order. Upon completion, newly applied operations are broadcasted to all users so that they all see the latest value. As soon as a user sends its operation request, he waits for an answer from consensus before applying it locally. Because of possible slow network or high consensus server load, this action often leads to poor performance. As an example, the user may experience slow User Interface (UI) reactivity because of a high amount of concurrent users. Invalid operation are refused by consensus and an error message is returned. This, however makes roll-back features easy to implement since an operation is locally applied only upon global validation.

2.1.5 Eventual Consistency

Eventual Consistency (EC) [66] allows replicas to temporarily diverge locally but eventually converge in the same state. Operations are not orchestrated by a central consensus, instead, they are applied in parallel by different replicas. Upon application, a local operation is broadcasted to all users through a propagation mechanism. The propagation must ensure eventual delivery [47]. This is independent of the network communication protocol (e.g., UDP). In other words, an operation executed at a local replica is eventually executed at all replicas. EC is a consistency model with convergence: replicas that have executed the same operations eventually reach an equivalent state. EC offers low latency at the risk of returning stale data (i.e., deprecated data) since, unlike SC, local data is not guaranteed to be up-to-date. However, replicas are guaranteed to converge when the system has been quiesced for a period of time so that all operations have been applied on each replica. Consensus bottleneck from SC is removed and relocated in each replica. In case of conflict, a global decision must exist so that all replicas apply the same resolution. In case of EC, this is called reconciliation [1].

2.1.6 Strong Eventual Consistency

Strong Eventual Consistency (SEC) is a special case of EC. Reconciliation, resolving conflicts at a local replica, is hard to design and may require manual intervention (e.g., manual merge in version control system). SEC removes this limitation by introducing rules in order to have a unique outcome for concurrent updates with a deterministic outcome for any conflict [39]. There is no longer a need for consensus or synchronization, since any kind of update is allowed and conflicts are removed altogether. This is specially appropriate for real-time collaboration. Every update is immediately applied and persisted. Replicas that have executed the same updates have equivalent state. Unfortunately, SEC may be very hard or impossible to implements for certain data type [28].

2.1.7 Optimistic Replication

Whenever a modification is applied locally, waiting for the server acknowledgement may be long (As seen in Section 2.1.4). Optimistic Replication [52] goes in favor of EC by having a data duplicate on each user's machine. Updates are first applied locally and then broadcasted using a propagation system with eventual delivery (such system has already been described in Section 2.1.5). Eventually, all replicas converge. There are two type of changes propagation: state transfer where the whole state is sent, and operation transfer which only sends the atomic edit. Operation transfer is usually preferred over state transfer for it requires less data to be passed over the network.

2.2 Real-time

In this section, we introduce the concept of real-time in software engineering, then we discuss in further details the case of real-time collaboration.

2.2.1 Real-time software

Real-time software are subject to a well defined time constraint such as a maximum response delay called deadline. Any processing must finish before it reaches this defined deadline or it will fail. This requirement may vary among software as well as failure consequences. For instance, real-time guidance systems for aircrafts have deadline in the order of few milliseconds and failures are critical, whereas real-time video games may have a similar deadline with less critical consequence in case of failure (e.g., screen freezing for a short time).

2.2.2 Real-time collaboration

Real-time software have a wide range of uses in real-world applications, such as embedded systems and communication software. Real-time collaboration is a specific kind of real-time application that focuses on shared data across multiple users and allows them to work on the same data simultaneously. It is often used through a graphical real-time collaboration editor. When real-time collaboration operates over a network, deadlines may not be satisfied in case of network failure. To avoid failure, collaborative software use replication and consistency features such as EC and SEC described in Section 2.1. The case of real-time collaboration may require a deadline in the order of few milliseconds.

Not all collaborative software are real-time. Although multi-user is supported for a single data, they may exchange (or merge) their changes at specific time. This is for instance the case of version control systems which uses Optimistic Replication (Section 2.1.7) in order to allow several users to work on the same data. Manual merging is used to reunite divergent copies and form a new version.

2.2.3 Requirements for collaboration

Updates have to respect several properties in order to be used in collaborative environment without creating unexpected behaviors to its user.

- Convergence: data end up in the same state after all users updates are applied. This is related to the notion of consistency defined in the previous sections (SC in Section 2.1.4, EC in Section 2.1.5, and SEC in Section 2.1.6. Figure 2.1 describes an example of collaboration on graph data. Alice and Bob both apply operations on their local replica and then notify the central server. This server broadcasts changes from Alice to Bob and vice versa. Upon completion, Alice and Bob see the exact same data.

- User intention preservation: original user intention must be preserved. As an example, Figure 2.1 pictures two users that apply concurrent editing. Alice adds a new vertex C with edge f that links B and C. Concurrently, Bob removes the edge e. Their intentions must be preserved regardless which algorithm is effectively used for concurrency control. Bob's deletion should not break Alice intention, which is to link B and C with edge f.

- Causality Preservation: whenever two updates are causally related (e.g., create before delete), the user should see them in this valid order. As an example, Alice

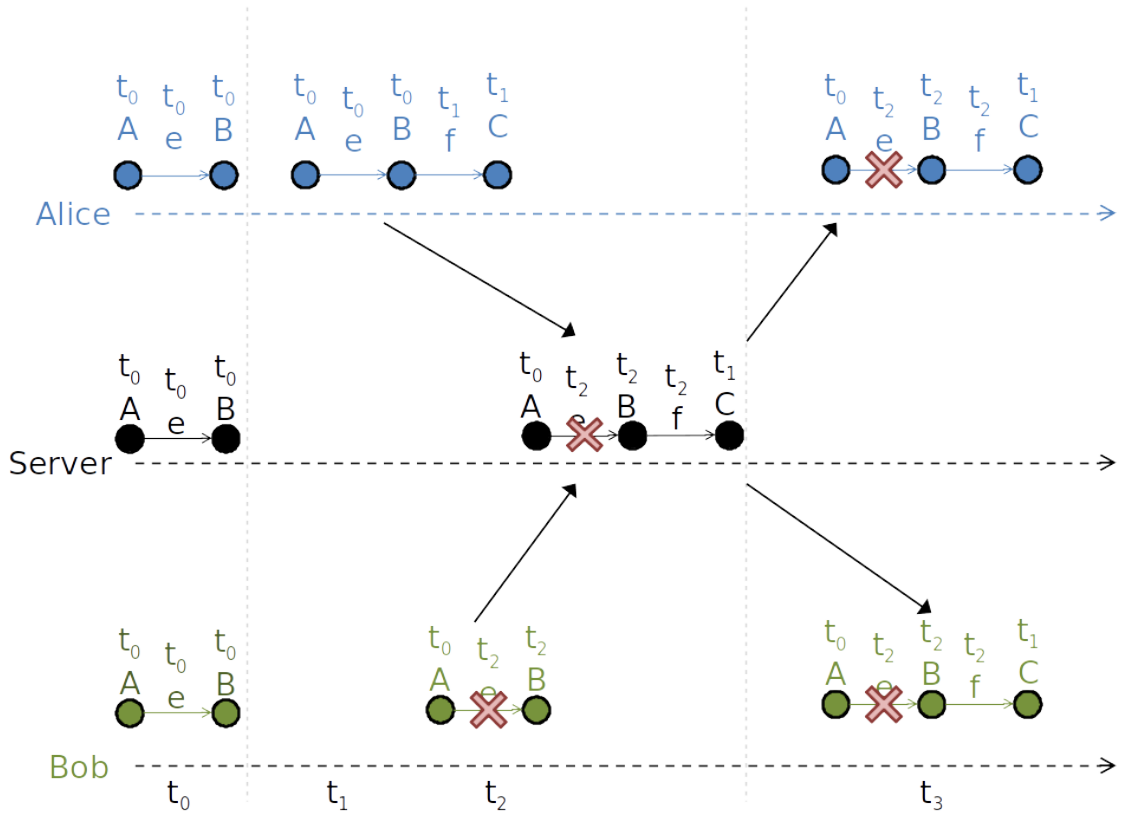


Figure 2.1: Example of collaboration on graph

from Figure 2.1 adds vertex C and edge f. These operations are causally related, meaning that add edge f can't exists without the previous operation add vertex C. This order must be respected by all replicas so that Bob and the server receives add vertex C before add edge f.

2.3 Concurrency Control Algorithms

Concurrent updates may conflict: although an operation alone is valid, it may change the context and, hence, be in conflict with concurrent operations. This case requires a concurrency control to resolve this situation. Pessimistic Locking solutions use different levels of locking to ensure SC and avoid conflicts preemptively. Locking a resource makes it unavailable to the others for modification. The user owning the lock must release it so others can modify it. This goes at the cost of productivity since only one user may own a lock at the same time, which introduce a bottleneck. All other users are required to wait until the resource operated on is available. On the other hand, Optimistic concurrency control uses more advanced concurrency control mechanisms to allow users to work on a whole data at the exact same time. This goes at the cost of algorithm complexity. Because of this growing complexity, these algorithms are more error-prone.

2.3.1 Pessimistic Locking

2.3.1.1 Coarse-grained locking

The whole data is locked until user releases it. This technique is not the most appropriate choice for real-time collaboration since only one user may work at the same time on the data (which goes against the very notion of collaboration). This is still relevant solution for collaboration that does not require real-time, such as environments with very few users or users working on a data at intervals (e.g., checklist shared by an office where updates are occasional). As an example, this solution is used as the base of concurrency control by *WebGME* [41] tool which uses global locking on the current data. In order to allow several users to work on the same data, they introduce a complex branching system that checkout a new branch whenever a user tries to work on a locked data. This diverging branch may be manually merged with the original data as soon as the lock is released.

2.3.1.2 Fine-grained locking

To avoid the whole data locked problem, this solution uses more locks to protect smaller portions of data. This may be implemented as a fragment lock where the root data is subdivided into several smaller fragments, each one with a distinct lock. Although only one user may work on one segment at the time, a resource fragmented in `N` partitions can accept up to `N` users with write operations simultaneously, assuming they each work on a distinct fragment. Obeo Designer is an industrial example of Fine-grained locking implementation that divides models serialized as XML documents into several locked fragments [45].

2.3.2 Optimistic concurrency control

2.3.2.1 Compare and Swap

Compare and Swap (CAS) is a multi-threading algorithm used to achieve synchronization without locking any resources. It is an atomic operation that, upon completion, checks whether the resource's original value has not changed during execution. A mismatch means that the resource has been used by another processes and Compare and Swap should fail instead of updating the resource. In order to work, CAS uses three parameters: the old resource value, the new resource value and the resource's memory location. Although this algorithm was originally designed for multi-threaded computing, it may be used for collaboration. As an example, the software Flip from Irisate uses a variation of CAS in order to achieve real-time collaboration without locking [28].

2.3.2.2 Three-way merge

This is the core algorithm used by version control systems such as Git and SVN. It uses Optimistic Replication so that each user has his own copy of the data and applies updates locally, giving the possibility to work offline. Diverging data are merged in order to create a new version with all users changes. Three-way merge [43] name comes from the algorithm's pattern: two diverging copies are compared against their common base data. Unfortunately, this is not appropriated for real-time collaboration since merge may requires manual intervention when conflicts cannot be resolved automatically. Moreover, during the merging process, data is locked and not accessible to any user The local user merge his data with current server's data , therefore the shared data on the server must be locked.

2.3.2.3 Differential Synchronization

Originally developed by Neil Fraser (Google) in 2009 [21, 22], Differential Synchronization (DS) uses Optimistic Replication in order to achieve real-time collaboration. DS relies on the Diff / Patch / Match algorithm. Cached modifications are compared with a copy of the previous known version called a shadow copy (diff algorithm [19]) in order to generate an edit which is sent to a remote replica. The former update its local copy using this received edit (Patch algorithm [20]). DS has a client / server architecture, where the server keeps a shadow copy for each client. Whenever the server receives an edit, it patches its local copy and broadcast the change to the other users. This is a symmetrical algorithm, nearly identical code is running on server and clients. DS is suitable for any content for which semantic diff and patch algorithm exists. Although DS only sends minimal edits, DS is a state-based algorithm and doesn't require that applications maintain a history of edits. This makes DS appropriate for applications with synchronization features. In case of network failure or unreliable network, DS implements a version checker system to detect dropped edits in order to re-send them. Hence, DS is highly fault-tolerant.

2.3.2.4 Operational Transformation

Operational Transformation was first introduced in 1989, primarily for text [17, 18]. It uses optimistic concurrency control to deal with real-time collaboration and is based on operation transformations [61]. Every edit is an operation (e.g., `Add 'x' at pos 3`) which is broadcasted to all other users. Upon reception by a replica, an operation is compared to local operations and transformed before being applied [5], [26]. This mechanism guarantees that, upon successful application of all operations, replicas converge to the same state. This is an operation-based strategy. GoogleDoc is a great example of a real-time collaborative software based on OT. Unfortunately, OT is considered as a complex and error-prone algorithm. Each possible transformation has to be defined, which may grow indefinitely as soon as the number of possible operations grows, therefore OT is often restricted to simple linear data, such as text. Google Wave is a relevant example of this issue. Because of its extended available data types, its OT implementation has grown nearly impossible to fully and successfully create for all possible transformations and Google Wave has been cancelled [23]. Moreover, OT has scaling problems in peer-to-peer environments and is best suitable for client/server architectures.

2.3.2.5 Conflict-Free Replicated Data Type

CRDT algorithm was presented in 2011 [55] by INRIA researchers (French Institute for Research in Computer Science and Automation). CRDTs are data structures that are replicated concurrently at all replicas using Optimistic Replication with SEC. They can be state-based (CvRDT for Convergent Replicated Data Types) or operation-based (CmRDT for Commutative Replicated Data Types). The CRDT algorithm focuses on operations design in order to accomplish concurrency control instead of using transformations (e.g., OT). An operation represents the smallest atomic edits that a user is able to apply on a data structure (e.g., add one value in a set). Concurrent users may apply operations simultaneously. The idea behind CRDT is that all operations have the following properties (with OpX for operation X. The notation $Op1 + Op2$ means that $Op1$ is applied, then $Op2$ is applied on the same data):

- Commutativity:
  $Op1 + Op2 = Op2 + Op1$
  Applying $Op1$ followed by $Op2$ produces the same result as the other way round.

- Associativity:
  $Op1 + (Op2 + Op3) = (Op1 + Op2) + Op3$
  Applying $Op1$ followed by the result of $Op2$ and $Op3$ produces the same result as applying the result of $Op1$ and $Op2$, followed by $Op3$.

- Idempotent:
  $Op1 + Op2 = Op1 + Op1 + Op2$
  Applying the same operation multiple times produces the same result as applying the operation exactly once.

This gives the possibility for operations to be applied in any order (Commutativity, Associativity), and be applied more than once without altering the result (Idempotent). Thanks to such properties, CRDT removes the need for a central consensus bottleneck. Each replica follows the exact same rules to apply operations, therefore they are guaranteed to eventually converge to the same state. It is highly fault tolerant and remains available even in case of network failure. It was originally designed for peer-to-peer asynchronous collaboration. Each operation has to be designed for a specific data type, which may become hard to implement for more complex data types. Unfortunately, even simple data such as an ascending integer counter is already complex. Therefore, CRDT may not be the most appropriate algorithm for all problems and data. The original research study presents several common CRDTs such as counters, set, and graph [57].

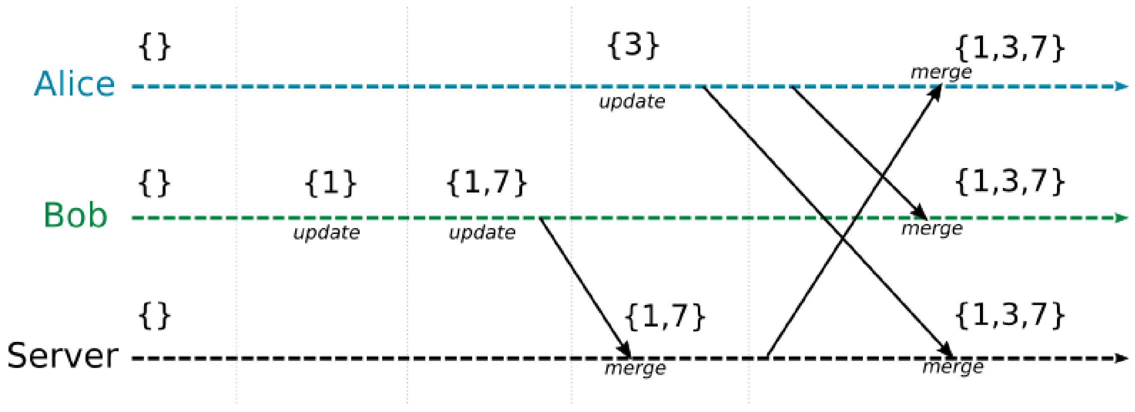


Figure 2.2: State-based CRDTs: example with a set of numbers

State-based object (CvRDT): updates are first applied locally, then the whole data state is sent to all other replicas and merged. An official example of implementation is available online [56]. Formally, CvRDT is defined as the tuple $\langle S, s_0, q, u, m \rangle$, where S is the global state, s_0 is the state at the beginning, q is the query method, u is the update method, and m is the merge method [55]. $s_i \in S$ is the local state at instant i. It may be read with query $q(...)$ and modified with update $u(...)$. At some point, the entire state is sent to other replicas. Received state is merged with local state using merge method $m(...)$, which is commutative, associative and idempotent. Figure 2.2 illustrates the state-based CRDTs. The merge method follows the same rule at all replicas: number are added in the list in ascending order. The state represents the current set of integers at a replicas.

Operation-based object (CmRDT): unlike for CvRDT, only operations are broadcast to and applied on other replicas instead of sending the whole state. An operation represents a possible data editing (e.g., add element). Operation-based object has the

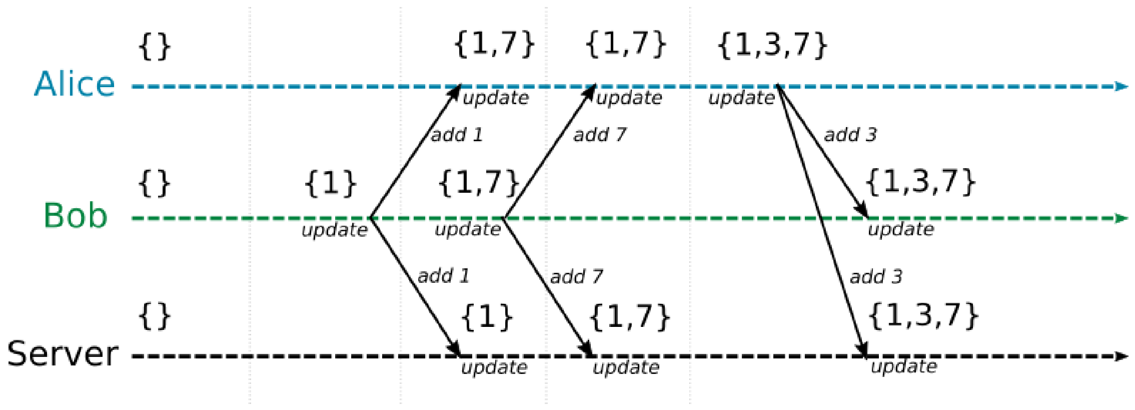


Figure 2.3: Operation-based CRDTs: example with a set of numbers

advantage of using less network traffic. All concurrent operations are commutative and associative. Idempotent property is assured by the propagation mechanism with eventual delivery as explained in Section 2.1.5. This mechanism ensures that operations are received exactly once at each replica. Formally, CmRDT is defined as the tuple $\langle S, s_0, q, t, u, P \rangle$. S,s_0, and q are the same as for CvRDT. There are additional components in CmRDT. Method update $t(...)$ is a side-effect-free update only applied on local replica and generates an operation to broadcast. Method update $u(...)$ is then applied on all replicas (local replica as well) and actually does the operation. P is a reliable causally-ordered broadcast communication protocol. Such protocol guarantee that messages are received exactly once by replicas and causally-ordered (idempotent). Any operations that are not causally-ordered must commute. Figure 2.3 illustrates the Operation-based CRDTs. Upon update application at local replica, the operation is sent to all the other replicas. Each update method follows the same rule: number are added in the list in ascending order.

2.4 Existing tools

Google Doc is one of the widely used collaborative tool for text editing. It implements OT algorithm in order to work with lock-free (Optimistic locking) concurrency control. It supports online collaboration with several collaborators in real-time.

Google Realtime API [26] is a library to build real-time collaboration on extensive data type using OT. A central server keeps the long-term storage data and several clients may collaborate on the same data in real-time. The API provides with a set of ready-to-use built-in data such as String, List, and Map.

Yjs [29] is an open source framework for offline-first peer-to-peer shared editing on structured data like text, richtext, or XML. It introduces a variation of the CRDT algorithm called YATA [44] in order to achieve SEC. YATA stands for Yet Another Transformation Approach. It uses a linked list as its internal representation and can be extended to achieve collaboration on new shareable data types. Operations are placed in this linked list according to a predefined set a rules which creates a total order of operations and removes possibility of conflicts. To support offline collaboration, each operation has a set of metadata used to determine its position in the list (total order) and doesn't rely on time based value such as timestamps. Yjs is a web-based tool written in JavaScript and may be pluggable with several kinds of databases such as in-memory storage. YATA introduces an implementation of garbage collector to avoid ever-growing memory.

Irisate Ohmstudio is a real-time collaborative digital audio workstation. It has a client/server architecture which mixes both SC and Optimistic Replication. Updates (called transactions) are applied locally before being propagated to the server. A transaction may be refused by the central server consensus and rolled back locally. This gives fast responsiveness with guarantee of consistency across all users. In order to remove the consensus bottleneck, ohmstudio uses its own concurrency control based on CAS algorithm. Collaborative features are internally bundled inside a framework called Flip which is reusable for extensive data types. Advanced real-time collaboration features are well-supported by flip, such as user undo/redo stack, history with owner, and collaborators informations (e.g., cursor position, selection…).

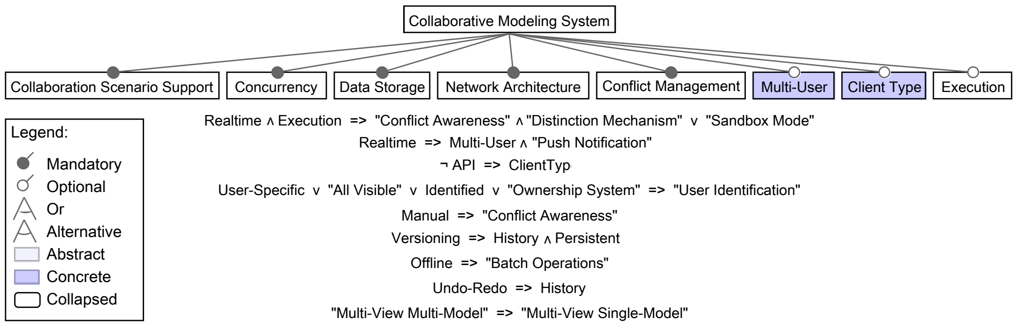
CHAPTER 3

FEATURE MODEL FOR COLLABORATIVE ENVIRONMENTS

In this chapter, we present a feature model for collaborative environment. This chapter was originally part of a paper presented during the CommitMDE 2017 workshop [42] and focused on collaborative environments for Model-Driven Engineering (MDE).

3.1 Features for collaborative environments

Modeling environments that directly support the collaboration of many stakeholders on the same model(s) working independently are collaborative modeling environments. These environments may be offline systems utilizing features similar to a version control system to manage the shared artifacts or may allow collaborators to interact remotely in realtime. We explored a variety of existing tools and potential solutions to identify a set of features for the implementation of collaborative modeling environments. In this section, we introduce and briefly discuss each feature. Figure 3.1 shows the top-level feature diagram and the constraints of the model. The complete feature model is available online in ReMoDD [12].



Collaborative Modeling System
Collaboration Scenario Support
Concurrency
Data Storage
Network Architecture
Conflict Management
Multi-User
Client Type
Execution
Legend:
Mandatory
Optional
Or
Alternative
Abstract
Concrete
Collapsed
Realtime ∧ Execution => "Conflict Awareness" ∧ "Distinction Mechanism" ∨ "Sandbox Mode"
Realtime => Multi-User ∧ "Push Notification"
¬ API => ClientTyp
User-Specific ∨ "All Visible" ∨ Identified ∨ "Ownership System" => "User Identification"
Manual => "Conflict Awareness"
Versioning => History ∧ Persistent
Offline => "Batch Operations"
Undo-Redo => History
"Multi-View Multi-Model" => "Multi-View Single-Model"

Figure 3.1: Top-level features and constraints

3.1.1 Methodology

The feature model presented in this paper is the outcome of an iterative process. To identify the feature of collaborative modeling systems, we investigated all kinds of collaborative environments. Our sources are: our own software (AToMPM [14]), MDE-specific environments (such as OBEO [45], MDEForge [4], GenMyModel [16] and, CDO), and other collaborative environments (such as GoogleDoc [26], Eclipse Che, Ohm Studio [28], Overleaf and, DropBox [27]).

We relied on published articles related to the collaboration aspect tools when available. We also reviewed technical documentation as well as relevant blogs, tutorials, and videos that explain the technical implementation of the tools. Finally, we experienced each tool ourself when freely available. In some cases, we studied specific algorithms, in particular conflict management algorithms used by several collaborative environments, such as Operational Transformation, locking mechanisms, and the DropBox synchronization algorithm explained.

From the collected set of data, we identified which features are specific to MDE collaborative environments.

3.1.2 Collaboration Scenario Support

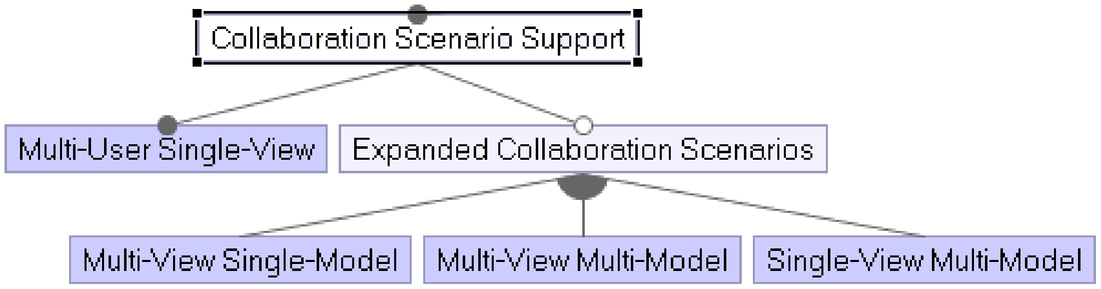


Figure 3.2: Collaboration scenario features

An abstract model is the abstract syntax of a model conforming to the metamodel of a given DSL. Conceptually, a view is a projection (in whole or part) of an abstract model utilizing the most appropriate representation of a subset of the model's elements for the needs of the modeler. An environment that does not support views could be categorized as supporting only a single complete view for each model. Therefore, we can detail the possible collaboration scenarios for both tools explicitly supporting views and those not supporting views in similar terms. In collaborative modeling, we previously identified four possible collaboration scenarios [14], that we briefly describe here. In the following, we commonly refer to scenarios with only two collaborators, but recognize the scenarios can scale to an arbitrary number of collaborators.

3.1.2.1 Multi-User Single-View

Users are working on the same view of the model. They both see the same information in the same language and with the same concrete syntax. The changes made by a user are reflected automatically to others.

3.1.2.2 Multi-View Single-Model

Users work on distinct views of the same model. The views may present the same, overlapping, or disparate sets of elements in the same or different concrete syntax. Modifications on the abstract model of elements present in both views are perceived by the other user.

3.1.2.3 Multi-View Multi-Model

Each user is working on a different view and each view is a projection of a different model. These models have some dependency or satisfy a global constraint. Only changes on elements related between the abstract models are perceived by both users.

3.1.2.4 Single-View Multi-Model

This is similar to Section 3.1.2.3, but the dependency is defined at the view level, not at the model level, such as an aggregation of elements from both models. A user may work on a view that projects several models while another is working on one of the

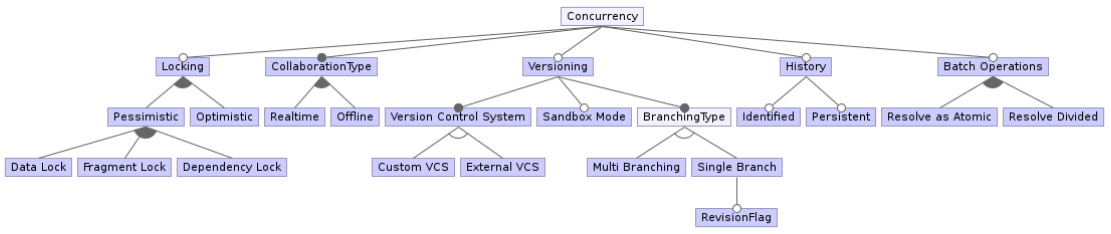


Figure 3.3: Concurrency features

projected models. A change on an element used in the view is propagated to all views with the same element.

3.1.3 Concurrency

Concurrency is concerned with issues related to multiple operations occurring at the same time or in parallel.

3.1.3.1 Locking

When several users are working on the same model(s) concurrently, conflicting updates may occur. Simple strategies, such as retaining only the last modification, could result in losing work from one user and goes against the goals of a collaborative environment. One approach to this issue is to avoid conflicts entirely allowing users to work together transparently. However, the collaborative aspect requires that users are able to work concurrently on components even closely linked. A general overview of the file-sharing problem in the MDE environment is presented in the subversion book [2].

Pessimistic locking is a strategy widely used in concurrent systems. It is based on data locking, which only allows one user to modify the locked data. A naive but simple solution is to use a global *Data Lock*. All of the model is locked to a specific user and other users cannot access the model until the lock is released. However, this reduces collaboration, because only one user can work on the model at a time. Another solution is to use a *Fragment Lock* that applies the lock on a fragment (i.e., subset) of the model. In this solution, each user is able to modify a distinct fragment concurrently. The fragments should be as small as possible, minimizing the locked portions of the model. However, the fragment lock approach requires a well structured data format supporting well defined fragments. In the case of XML for example, we might lock the current XML tag and its children. Additionally, fragment locks do not consider dependencies that may indirectly affect the elements locked (e.g., metamodel relationships). OBEO Designer implements data and fragment locking. Another approach relies on *Dependency Locking*. This provides finer granularity that locks the element being modified and its dependencies. Though this technique is seen as only an improvement here, taking dependencies into account may be seen as required by the semantic nature of models. This is discussed in more detail in Section 3.3.

Though we try to minimize the set of elements to be locked, pessimistic locking always blocks access to a set of data. This might lead to a situation where a user waits for a resource to be free. OBEO shows an overview of these locking techniques in their documentation [45]. *Optimistic Lock* tries to resolve this issue without locking. Instead, it allows the users to modify, possibly the same, elements concurrently and then merges all changes to create one unified new version of the model. For example, this is how Google Docs allows for concurrent changes and relies on the Operational Transformation (OT) algorithm [60] for merging. Unfortunately, though OT works well for text based data, model merging requires merging graphs, which makes this approach hard to apply [63].

3.1.3.2 Collaboration Type

We differentiate two scenarios of collaboration: *Realtime* and *Offline*. In the former, the model is modified by several users at the same time; changes are applied on the data immediately; and users are updated of changes made by others immediately. The model is the unique source of truth that all users alter concurrently. Here even the changes from a given user may not be considered complete until acknowledged by a central authority or a set of peers. On the other hand, offline collaboration presents an asynchronous approach. Users may apply modifications on the model without sending the changes right away. This principle is often seen in version control systems (VCS) that allow working locally and pushing changes at a later time. This prevents immediate conflicts between users, but local versions may diverge and result in complex merging processes later, as in Git.

3.1.3.3 Batch Operations

All modeling systems support a set of atomic operations (e.g., creating or deleting an element) that provide the ability to develop and manage models. We recognize that some systems also allow collecting together these atomic operations in batches. The processing scheme for these batches becomes significant when discussing collaborative systems and handling potential conflicts. *Resolve as Atomic* resolves a set of operations in an all-or-nothing approach. Every operation in the batch must succeed or none of the operations can be applied. Thus, a single failing operation may result in the need to roll-back changes from a set. *Resolve Divided* allows processing each operations separately. However, this may result in partial sets being applied thereby generating unexpected or even invalid results.

3.1.3.4 History

In a collaborative environment, storing the complex history of operations applied to a given element can be beneficial. The series of events leading to a conflict or failure may be complex, and not easily understood without a record of the operations. Here we intentionally separate history from *versioning*. For instance, a basic feature of any VCS is to manage the project history. Therefore, the presence of versioning mandates the presence of some form of history (this is represented as a constraint in the feature diagram). However, a collaborative environment might store some portion or form of history without the presence of a VCS. GenMyModel [16], for example, has a full and complex history feature, that allows replaying the history from a defined start time. History is *Persistent* if it persists when the environment halts. An example of non-persistent history is an undo / redo stack that only lasts until a given session terminates. Using a VCS implies that the history is persistent. However, it may not be enough to use the VCS history directly, but a user interface (UI) wrapper might be required. VCS integrates the set of changes from the whole project. While working on a specific model, the user does not need all of this information. Therefore, an environment might implement a history wrapper that shows only the current data models history. *Identified* history adds the user identity information to each modification. This is often the default behavior of a VCS. Allowing attributing a change to a specific user helps in diagnosing a series of events from multiple users that may have led to a conflict or failure.

3.1.3.5 Versioning

VCS are tools that trace all changes for a system and have facilities for managing this history. A single user working on a data model has a complete view of its state at all points and understands the full history of the model naturally. However, several users working on the same data model may lead to misunderstandings and inconsistencies. When trying to merge artefact(s), the data model(s) might be significantly altered since the last connection. Changes performed by other users may be difficult to understand. This introduces the need for versioning systems with history management. It adds the possibility to look back at the changes performed over the intervening time and to understand the full series of changes. Moreover, VCS also support other features such as documentation, reverting previous changes, or listing the added features for a release. A survey on model versioning was provided by Altmanninger et al. [3].

3.1.3.6 Version Control System

VCS play an important role in software development. Tools like Git and SVN are largely used and are very efficient. Therefore, collaborative modeling environments may opt to integrate an *External VCS* into the environment rather than reinventing the wheel. This places the data under the VCS management and each change is added to the history. However, these VCS use compare and merge mechanisms to integrate the changes into a new version, which require manual intervention. For instance, to integrate new changes made under Git, changes must be commited and pushed manually. Moreover, these systems are subject to conflicts, which require manual resolution. Employing locking with a version control system may avoid conflicts. Merging is then performed automatically in the background. This approach is utilized by MetaEdit+ that integrates Git with their MDE collaboration [31]. They use fine granularity locking to avoid conflict, that allows processing the merge without risk of conflicts. Furthermore, these VCS are specialized for versioning, comparing and merging text files as opposed to complex semantical data structures in MDE. For these reasons, some modeling systems may opt to build *Custom VCS*. We discuss these issues in more details in Section 3.3.

3.1.3.7 SandboxMode

Sandbox is used to divorce a user from the typical collaborative environment. This supports experimental or debugging processes. Working in a sandbox environment allows developing components that temporary break other components or violate general rules/expectations, without disturbing other users. The modifications may then be integrated when complete, potentially resulting in complex merges.

3.1.3.8 Branching Type

Multi Branching is used to divide the project into several branches. In VCS, branches are a divergent copy of the project where users may work independently from other branches. Branches are often used for new incoming feature that are not stable yet. As soon as the feature is stable and needs to be integrated, a merge with the main branch is done. This eases the team work and separates the maintenance from the new release components. However, a manual merge is required and the new feature(s) may be difficult to integrate with the main branch. New features are built on top of the old base code that may be deprecated. This issue appears when maintenance largely diverges from the main branch. Moreover, dependency ambiguities are complex to resolve, which is emphasized by the semantic nature of models.

Single Branch avoids these issues. No branches are used and every change is performed on a single version of the system. In order to separate new features, e.g., to exclude them from the release, *Revision Flags* may be used. In this way, new features depend on the up to date state of the code, while still hidden from release. One great example is Google with the Piper VCS which uses only one trunk for all its teams and flags for new features [49]. It is important to note that the use of branches is not always relative to the actual VCS branching support. Though the used VCS may support branching, users (or even environments employing an external VCS) can make the decision not to use branching. This is the case with MetaEdit+ that uses Git as a VCS but does not use Git branching feature.

3.1.4 Data Storage

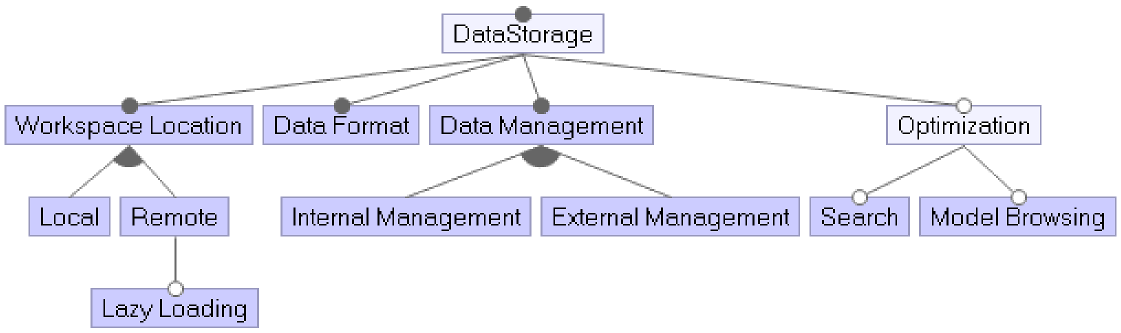


Figure 3.4: Data Storage Features

Data storage is concerned with how models are stored and managed in the system to enable reuse among collaborators.

3.1.4.1 Workspace Location

In a collaborative environment, data is often saved in a remote place (i.e., the "cloud"). The workspace location is where the data is stored while users are modifying it. Data may be *Local*, meaning that the relevant models are on the users local machine. This may support an asynchronous workflow as with an offline collaboration type or be a response to other constraints on the system. Collaboration is often reduced to active screen sharing (single-view single-model) as supported by AToMPM. Since network latency is sometimes high, having a local copy may remove the delay between an operation being requested and being applied on the data. In contrast, *Remote* locations do not require loading the project locally. This is useful in the case of a project using a high quantity of memory. *Lazy loading* may even be used to load specific data only when required.

3.1.4.2 Data Format

Models use a specific data storage format. The choice of format is important and may influence or be influenced by other factors (e.g., the VCS). Some formats may be hard to use with a text-based VCS, hard to fragment for locking, or mandated to be compatible with a distributed database utilized by the system. Popular formats are JSON (for web-based modeling environments), XMI (for Eclipse-based tools), or NoSQL database formats when scale is an issue.

3.1.4.3 Data Management

Data management refers to managing the basic operations and long-term storage in the system (e.g., CRUD operations). *Internal Management* implies data is processed by tools internal to the software, whereas *External Management* reuses existing tools like a distributed database. Namespaces (e.g., URI) are crucial to access a model.

3.1.4.4 Format Optimization

Format Optimization identifies the way a system might optimize the model representation/storage for certain actions; e.g., *Model Browsing* or model *Search*. Basciani et al. [4] overview different supported query mechanisms for several systems according to the managed artifacts.

3.1.5 Network Architecture

Collaboration requires the use of the network so that instances of the modeling system on different machines communicate and exchange data.

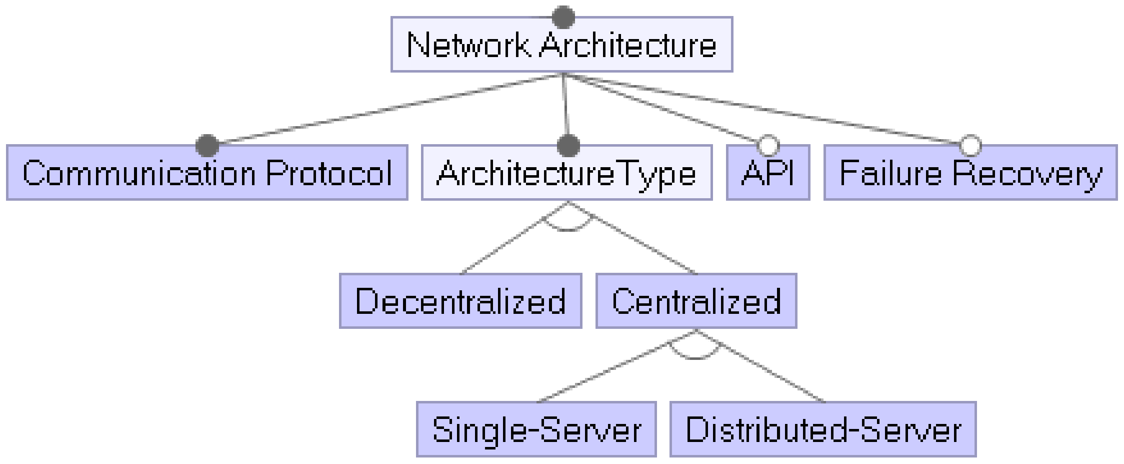


Figure 3.5: Network Architecture Features

3.1.5.1 Communication Protocol

Clients and servers must be able to communicate using a common protocol. TCP/IP is a widely used network protocol. It may be relevant to investigate reliable networks to guarantee no data loss, but at the cost of time performance. Client and server must then use the same data format to communicate information.

3.1.5.2 Architecture Type

This is the network architecture chosen for the collaborative environment. We distinguish two fundamental types of architectures: *Centralized* and *Decentralized*. *Centralized* uses a central authority. It has the advantage of having one source of truth: the centralized storage is the true version of the work. Alternatively, *Decentralized* systems distribute authority. An example of *Decentralized* architecture is Git, though often used in a *Centralized* way (e.g., Using Github server as the main storage location).

Centralized architecture may be *Single-Server* or *Distributed-Server*. This is an internal detail, since end-users see the cluster as only one single-authority. *Distributed-Server* adds complexity to handle data synchronization across all storage locations. Nevertheless, it adds a layer of security (a single server crash will not affect the whole system) and performance (distributing processes and storage across a large set of nodes).

This is useful for servers with high traffic demand. An industrial example of this is the Google Piper VCS [49], which is duplicated into 10 servers across the world using the Paxos algorithm [37]. This divides the number of request performed on each unique server and speeds up response time.

3.1.5.3 API

An API, though optional, may be integrated to allow extension of the system and other client implementations. For example, we are working toward an API for collaborative modeling services to allow building and integrating potentially many distinct clients [13]. Each client may support their own needs and rely upon the common modeling services provided by the API to simplify system design and interoperability. APIs are the basis for modeling as a service systems, such as MDEForge. Additionally, an API may be provided purely for internal use to manage operations between layers or nodes in the architecture. For example, there is a simplified API between the MVC and MvK within the architecture described in our prior work [13].

3.1.5.4 Failure Recovery

Hardware and network systems are subject to failure and error, but user experience should not be affected by a technical issue within the system. The possibility of recovery is closely relative to the chosen *Network Architecture*. Decentralized architectures mitigate this issue as each user owns a state of the project. Though the error might be disconnected from many users, consistency schemes must exist to manage the decentralized authority.

Centralized architectures are notably subject to failure in the case of *Single-Server* where there is a single point of failure sufficient to take the entire system down, and recovery can be difficult to impossible depending on the severity of the failure. On the other hand, *Distributed-Server* may implement a failure recovery system. If one server crashes, the system may use another server instead to maintain availability and redundancy may be used to prevent the loss of data.

3.1.6 Conflict Management

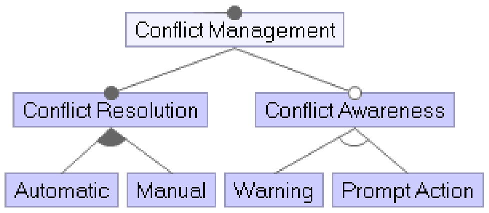


Figure 3.6: Conflict Management Features

Divergent modifications on the same data model may lead to a conflict when trying to resolve all modifications to a single consistent model state. It is important to detect conflicts and act upon resolving them.

3.1.6.1 Conflict Resolution

Conflict resolution is the process by which all modifications are combined in order to create a new version of the model. *Automatic* conflict resolution is the ideal solution, where conflicts are resolved automatically. Other features, such as collaboration type and locking, strongly impact the complexity of managing automatic resolution of conflicts. A live collaboration environment with fine grained locking may be conflict free. High risks of conflict appear when using offline collaboration and versioning. Each user works separately on a divergent version of the work. To share his change with others, the user needs to perform a merge action. This could be accomplished through a diff / merge algorithm. In case of unambiguous changes, this action may be transparent to the user. For example, if each user changed different parts of the model(s) without cross dependencies. However, often *Manual* conflict resolution may be needed. This is the case of WebGME tools that refuse a push request if the server has already been modified [40]. The user must then first pull the latest changes and manually merge them with his own changes before pushing them to the server.

3.1.6.2 Conflict Awareness

When working in collaboration, users should be aware of other users changes. *Conflict Awareness* is the mechanism that warn users of conflicts. Notification may be sent in response to conflict (e.g., 2 users move the same element) or to warn users about potential conflict (e.g., elements being edited by other users). It is disruptive for a user to perform a modification that results in unexpected behavior, because the user was not notified of a conflict.

Warning conflict awareness are only mechanisms to inform user about conflicts and concurrent actions. This feature provides only information about conflicts or potential conflicts. This is often used by the GUI, which uses combinations of colors and animation to display the information. OBEO applies this solution by drawing a lock icon at the bottom of any locked element. However, this is not only restricted to GUI, this can also be applied in an API. For instance, it can be a special return value of a function in case of conflict, or an object state that changes according to it's conflict state. The case of a GUI is discussed in more details in Section 3.3.

Prompt Action conflict awareness on the other hand, informs about conflict and requests the user take some action, such as in MedaEdit+. The user is prompted to update the model with the server to remove existing locks.

3.1.7 Multi-User

This feature is concerned with the fact that multiple users are interacting with the same or related models.

3.1.7.1 Authentication Method

Collaboration implies that several users are able to access the data models. An authentication mechanism can be incorporated to give access only to registered users or to simply track who is responsible for a given operation. *User Identification* requires such an authentication method. The method may involve utilizing an external authentication method. For example, GenMyModel allows users to connect through their Github ac-

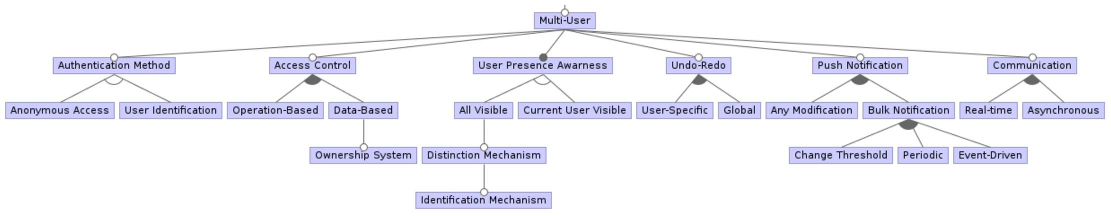


Figure 3.7: Multi-User features

count. Alternatively or in addition, *Anonymous Access* may be provided to allow users to connect without having to register. This can be used for example to allow read-only access. One example outside MDE is the IRC channels that do not require registration.

3.1.7.2 Access Control

Users working in collaboration must be able to access the data model. However, each user may be provided a distinct set of permissions. This is handled by the *Access Control* feature. There are two distinct alternatives: *Operation-Based* and *Data-Based*. They may be mixed together in order to have more fine-grained control. *Operation-Based* restricts what a given user is able to do on any model, based on the possible set of operations (e.g., CRUD operations). A user is granted a specific operation permission that applies on any model from the project. This is similar to a database administrator that has full access to all elements. On the other hand, other users may have a restricted set of allowed operations. *Data-Based* access control may be provided to control the elements (at model or element level) and operations for those elements available to the user.

Access Control can be augmented with an *Ownership System* that adds the possibility to restrict exceptional permission to only a set of users, as in GenMyModel. A project can be shared with a team and permissions are given to certain users. They also add project visibility. A public project is visible to everyone, though read only, any user can clone the project in its own session, thereby creating a totally new project copied from this public repository.

3.1.7.3 User Presence Awareness

When collaborating, it is often preferred to know who is working concurrently. *User Presence Awareness* is the mechanism used to know which users are currently working on the same model. *Current User Visible* does not display any user other than the current one. This is the behavior of non collaborative software. This might be useful in collaboration in case of high number of users, to remove the information overload, but instead only show the total number of current users. To fully use the power of collaboration environment, it is recommended to use *All Visible* feature. This implies that the presence of all users working concurrently is known. One implementation would be to highlight the mouse cursor of all users (e.g., Google Docs) or highlight the graphical element actively used by each user. Nevertheless, a large number of users may cause confusion on the canvas. This introduces the use of a *Distinction Mechanism* that adds special attributes to differentiate users. It may only differentiate local user from all others (e.g., using two colors), or differentiate each user (e.g., one color per user). Using a distinction mechanism for each user can also lead to information overload. We can add additional information by using an *Identification Mechanism* to identify the operation/focus of each user distinctly. This is used by GenMyModel, which lists all current collaborators for a model.

3.1.7.4 Undo / Redo

Undo / Redo is a fundamental feature of professional software. The common and expected behavior of undo is to revert the last action performed with further invocations of undo reverting prior action in reverse order of original application. The standard behavior considers only a single user, providing *User-Specific* Undo / Redo. Common patterns are known for implementing this feature (i.e., Command pattern [25]). Though this pattern works well for a single-user environment, additional complexities must be handled in collaborative systems. We need to take into account not only the current user changes but also other user changes. Using a local undo stack is not sufficient: reverting a command in a collaborative environment must consider all commands. Otherwise, unexpected behaviors might appear since our previous state has been altered by others. Consider the following scenario. Alice adds the attribute `name` in an empty element. Bob adds the attribute `age`. Let us assume our undo implementation resets the element to its exact previous state. If Alice undoes her change, the element ends up being empty again. The `age` attribute has also disappeared because the implementation neglected modifications from other users. A real-time API must implement a more complex stack system for its undo / redo management that takes into account the sequence of operations and resulting interdependencies introduced by concurrent collaboration. An example Undo / Redo in a collaborative environment is discussed in detail by Cheng et al. [11]. Another way to handle Undo / Redo is to use *Global Undo / Redo*. The stack is shared across all collaborators. GenMyModel can handle both alternatives: a user undo / redo is present, while history features allow global undo / redo using the general stack of changes.

3.1.7.5 Push Notification

This is the mechanism by which user modifications are automatically propagated to others collaborating on the same model(s). Whenever an operation is executed, all users must be aware of it and their local data model updated. We distinguish between two kinds of *Push Notifications*. *Any Modification* is a continuous notification scheme. Every operation generates a push out to every user. This ensures the server is up to date and all users see what others are doing in real-time (with some tolerance for network latency delays). This technique is used by OBEO, GenMyModel, and WebGME.

Alternatively, *Bulk Notification* regroups changes and sends them only when specific conditions are reached. Bulk notifications may be sent manually in an *Event-Driven* approach (e.g., on a save action), in a *Periodic* approach (e.g., each 2 minutes), or in a *Change Threshold* approach. MetaEdit+ chooses the event-driven option and sends changes only when the user saves. This gives several advantages over *Any Modification* notification. Network load is notably reduced and users can hide from others the temporary broken or intermediate state of the data model. Working on models often requires moving a set of elements, temporarily destroying links, and other temporary or intermediate states. Though the goal of one's action may be acceptable, others may be disturbed by this temporary/intermediate state. Other users are only interested in the result. However, the *Sandbox* feature can be provided to handle similar scenarios.

3.1.7.6 Communication

Communication tools are used to contact other collaborators, ask them questions, add comments on elements, or discuss conflicts detected and merges. Communication tools are closely related to *User Presence Awareness* that allows identifying who is working concurrently, and then communication tools enable contacting them without using external tools. *Real-time* tools allow for direct communication. The users must communicate in real-time. This is the case of video call, audio call, and instantaneous chat. On the other hand, *Asynchronous* communication allows delayed communication with other users (e.g., chat box, comment, or annotations on components). A complete taxonomy of communication tools in MDE collaborative tools is provided by Davide Di Ruscio in his systematic mapping of collaborative model-driven software engineering [15].

3.1.8 Client Type

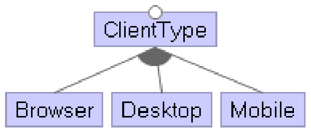


Figure 3.8: Client Type Features

The end-user typically works on model(s) through a graphical representation. Modeling tools often make heavy use of mouse cursor. Therefore, *Desktop* (MetaEdit+, OBEO) and *Browser* (AToMPM, WebGME) applications are common choices. On the other hand, very few *Mobile* application are utilized for modeling(e.g., FlexiSketch [67]). *Browser* clients have an advantage in portability. Collaboration involves multiple users that often use different operating system. Supporting a cross platform desktop application introduces additional complexity and in modern systems in-browser environments seem to be preferred. In-browser environments also eliminate the need for installation and dependency management for end-users. However, a system may be provided without an explicit client type being specified, but an appropriate API must be provided to enable access to the system.

3.1.9 Execution

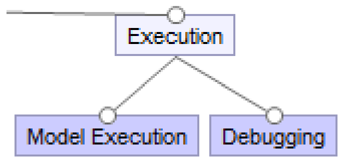


Figure 3.9: Execution features

A special attention should be given to executable models in a collaborative context. Let it be model transformation or model simulation, *Executing Models* can be done locally or have an impact on the remaining users' experience. Many issues arise, such as whether the transformation should happen in batch or every step is visible to all users. The latter is supported by AToMPM. Similarly, *Debugging* executable models is often done in a different mode than when editing. This may raise conflict where the state of a model element is modified by the execution while another user is modifying it. Note that supporting execution within a realtime environment imposes additional constraints (included in Figure 3.1). The constraints address the need to support users working concurrently by preventing or mitigating the impact of conflicting operations on the workflow of a given user.

3.2 Examples and instantiation

We present four tools, among those we examined, chosen because of specific variants of the features they use and because there was enough documentation to support the claims.

An instantiation of the feature diagram will include all mandatory features as well as specifying those features with alternative sub-features or any optional features included.

The following discussion assumes all mandatory features are included. In our feature diagram, the top level mandatory features are *Collaboration Scenario Support*, *Concurrency*, *Data Storage*, *Network Architecture*, and *Conflict Management*. Without one of them, a collaboration environment wouldn't be possible. On the other hand, *Mutli-User*, *Client Type*, and *Execution* are not mandatory. Though feature such as *Multi-User* are important, some cases may not need it. This is for instance the case of API for MDE collaboration without a provided client.

3.2.1 Obeo Designer

OBEO is an MDE collaborative modeling tool in Eclipse based on CDO with *Centralized* server and continuous integration. Concurrency in OBEO uses *Pessimistic Locking* with both *Data Lock* and *Dependency Lock*. Any modification on a model locks the whole model for this user (*Data Lock*) restricting the ability for two users to work on the same model concurrently. However, they can work on separate models, even with strong dependencies, using *Dependency Lock*. Whenever a model is locked, its dependencies are also locked. This is fine grained locking allowing users to work concurrently without conflict. OBEO uses four Easy-To-Customize representations, diagrams, tables, matrices and trees. Collaborators may use any of these representations to work on the same data model, therefore, both scenarios *Multi-User Single-View* and *Multi-View Single-Model* are supported.

3.2.2 WebGME

WebGME is an open source project that implements collaborative modeling with a *Centralized* server. Concurrency is integrated with a *Custom Versioning System* that also handles *Multi Branching*. Several users may work on different branches and then merge together, using the integrated merging tool. Each modification automatically creates a commit. *Offline modification* is supported: the working branch is automatically merged with the synchronized version at the next connection. Concurrency is resolved by an interesting *Optimistic Locking* mechanism such that no actual locks are required. In most cases, conflicts are resolved automatically using, for example, a first-win rule. However, some conflicts require manual intervention. In that case, a conflict is automatically resolved by creating a new branch for the user in conflict. His changes are then local to its branch and a manual merge to the trunk is required. WebGME supports the *Single-View Multi-Users* scenario.

3.2.3 GenMyModel

GenMyModel is a browser-based modeling tool that supports collaboration. GenMyModel introduces a complex sharing system similar to that provided by GitLab and Github; i.e., *Data-Based Access Control* with an *Ownership System*. A user is the owner of his project and has all rights on it. Collaborators may then be added to the project with specific rights. An *Operation-Based Access Control* is introduced with the public or private state of a project. In public projects, any user has read-only access, whereas private projects are visible and accessible only by their owner and collaborators. GenMyModel provides *Live Collaboration*, implementing a *Centralized Network Architecture*. Data can be accessed from anywhere and is not cloned on the client side. *History* is implemented without an external VCS. All changes made by users are saved creating a modification timeline. Users can therefore browse the change and go back to a previous version. GenMyModel allows several collaborators to work on the same model with the same view. This is the first scenario *Multi-User Single-View* that is supported. Moreover, it also adds possibility to have a view that project several models, therefore *Single-View Multi-Model* scenario is also supported.

3.2.4 MetaEdit+

The collaborative infrastructure of MetaEdit+ is similar to OBEO. It uses a *Centralized* server with continuous integration and applies *Pessimistic Lock* with a fine granularity and distinguishes dependencies. Therefore, only a minimal set of elements is locked and only when needed [32]. It integrates a *Version Control System* using an *External VCS*. The VCS working directory is managed as a separate repository and uses the MetaEdit+ *API* to process the requests. In that sense, MetaEdit+ implements a full support of existing VCS to version control models. *Single-View Multi-Users* scenario is supported by MetaEdit+.

3.3 Discussion

3.3.1 Locking and dependencies

Locking a model may seem safe, since only one user can apply modifications on it. However, dependencies must be taken into account. The current model Alice is modifying (and locked to her) is safe from Bob's updates. But this model may have dependencies with another model that is not currently locked. If Bob modifies this other model, we would have altered Alice's model, even if it was locked. For instance, Alice can divide a UML diagram into several subsets linked together. Changes that affect another subset should spread. Therefore, when an element is modified, other affected elements must be locked as well. However, dependencies are common in models and even small fragments might end up locking a huge set of elements. Dependencies should be taken into account, but supporting transitive links may be too restrictive. Maroti et al. emphasize how simple operations, such as copy or delete, may end up locking a significant portion of a model [40].

3.3.2 Versioning for Models

Version control systems are widely used for source code projects. The syntactic nature of source code works efficiently with the text based nature of this diff & merge. The VCS detects modifications in the files and create a new version by merging both changes. EMF Compare is a good example of an MDE tool using text-based comparison for models. A recent article shows a way to use EMF Compare with EGit on Eclipse in order to integrate MDE versioning [48]. However, only diff & merge is often not the most relevant for MDE because of the semantic nature of models. This issue of integration with VCS is explained in details by MetaEdit+ [31].

3.3.3 GUI for Conflict Awareness

We have explained the importance of having a *User Conflict Awareness* mechanism. Here, we discuss concerns relevant to GUIs. The following examples emphasize the importance of *User Conflict Awareness* in a GUI. Assume that Alice drags and drops an element from a model. Bob tries to drag and drop the same element at the exact same time, but to a different location. Alice releases the element before Bob. If the conflict management system is optimistic, the last to drop is retained. Therefore, Bob's final action prevails, and the element is placed according to his action. From Alice's point of view, the element disappears or shifts when she releases it. This is confusing and unexpected behavior. Alice may even see this as a bug. In such situations, a GUI warning may be displayed. One solution is to blink the element on Alice's GUI and play an animation that moves the element to the final position (from Bob's action). This way, Alice understands that her change was immediately followed by Bob's. OBEO applies this solution by annotating with a lock icon at the bottom of any locked element. Google Drive, places a special red cross icon on a deleted element. The cross lasts long enough for the user to see the element being deleted before it disappears.

3.3.4 Why should we use User Presence Awareness

Knowing who is currently working on the same document has several advantages. For a communication purpose, this warns of others currently working on the same project. This eases the general knowledge of the team work and who to contact if an element needs to be discussed. For instance, if a fast feed-back is required for a modification, it is easy to request others. Moreover, it has an impact on the learning process: users are able to help each other. If Bob makes use of an action unknown to Alice, asking him is easy and fast via the communication mechanism in place.

3.3.5 Note about Push Notification

We discussed in the *Push Notification* feature that user modifications are sent either continuously (*Any Modification*), or regrouped and sent manually or periodically (*Bulk*

Notification). In practice, it is often a mix of both. For example, in GoogleDoc, modifications are sent at upon saving the document, which is triggered automatically. The frequency of saves usually starts after a significant change is made in the document. This result in an almost continuous *Push Notification* and releases the network load at the same time.

CHAPTER 4

CONCURRENCY CONTROL ALGORITHM

We propose a variation of the CRDT algorithm. In this chapter, we present our CRDT implementation used by our CollabServer framework.

4.1 CollabServer CRDT algorithm implementation

We introduced the CRDT algorithm in Section 2.3.2.5. In this section, we discuss the details of implementation and the choices we made for our CRDT algorithm integration in CollabServer. Although CollabServer framework uses a central server that users may connect to collaborate with each others, our algorithm is designed for client / server as well as peer-to-peer architecture.

4.1.1 Operation-based CRDT

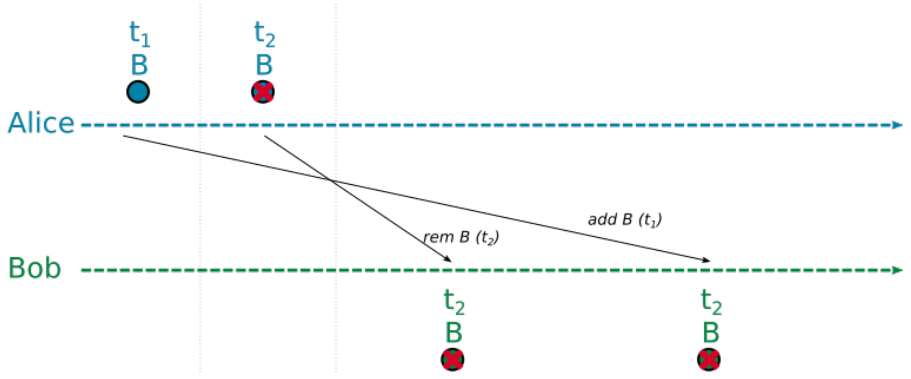


Figure 4.1: CmRDT operations commutativity

We choose the operation-based approach of CRDT (i.e., CmRDT). This approach requires less network bandwidth since only the minimal representation of an operation is to be sent over the network. Moreover, the concept of operation is convenient to use with external components, such as a database or a GUI (e.g., components that directly use the CRDT to display or save it). As an example, each operation on a CRDT set (e.g., add, remove in Figure 4.1) is easily translated into a SQL database statement (e.g., INSERT B, DELETE B). The case of GUI is similar and may use an observer pattern to notify received operations. Operation-based CRDT is formally defined in Section 2.3.2.5 as the tuple $\langle S, s_0, q, t, u, P\rangle$. Our variation of the original CRDT description comes from P, which is a reliable causally-ordered broadcast communication protocol. CmRDT requires that concurrent operations (not causaly related) are commutative. Idempotent and causal-order are formally ensured by P. In our implementation, we decided to make all operations commutative and idempotent, regardless of their causal-order. This removes the need of P and gives the possibility to apply operations in a totally arbitrary order. Moreover, upon reception by a replica, operations are applied immediately. This removes any need for synchronization or complex delay mechanism. In Figure 4.1, Alice applies `add B` at t_1, then `remove B` at t_2. These two operations are causally-ordered. Formally, CmRDT requires that P delivers these operations with the same causal order at all replicas (in contrast with concurrent operations that must commute). Bob receives the remove operation before add, which is valid in our implementation since all operations are commutative. Upon `add B` reception by Bob (with timestamp t_1), our algorithm detects that vertex B has already been deleted at t_2.

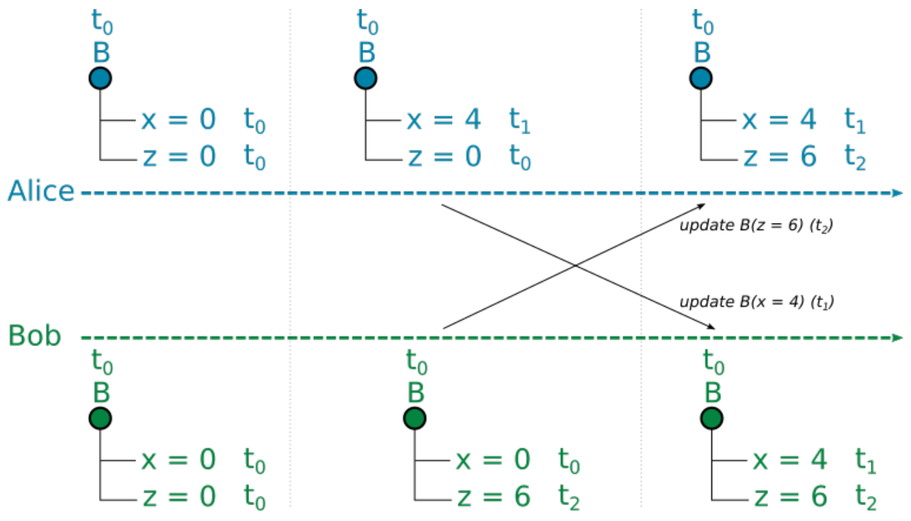


Figure 4.2: Example of fine-grained timestamps in CmRDT Graph with attributes

4.1.2 Last-Writer-Wins (LWW) and timestamps

LWW stands for Last-Writer-Wins, meaning that timestamps are used to choose the prevailing operation in case of conflicts. A fine-grained use of timestamps is required to support complex components with internal attributes as described in Figure 4.2. A graph vertex B has two attributes x and z, with an integer value. Each attribute is an atomic entity with its own timestamps so that, updates on one attribute will not affect the others. One may use a global timestamps for the whole vertex. However, this would lead to a misleading behavior, since vertex B would end up with $x = 0$ and $z = 6$ for Bob has the latest timestamp. To avoid this, we use fine-grained timestamps: each attribute has its very own timestamp.

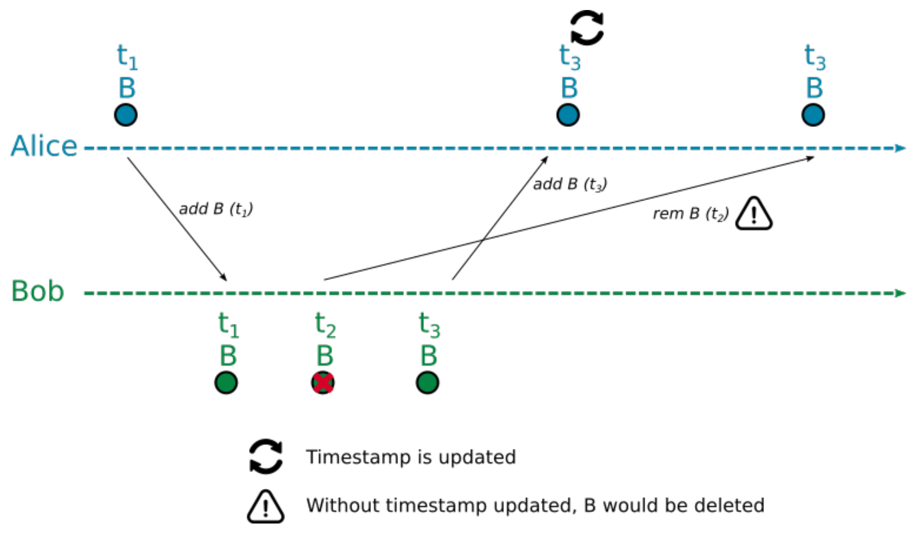


Figure 4.3: Timstamps requirements in case of identical operations

Another important requirement when using timestamps with CRDT is to update timestamps values even in case of identical operations on the same element. One may think as an optimisation to first check if two operations on the same element are identical (e.g., add B) and possibly bypass this operation. This, however, may lead to an inconsistent state. In Figure 4.3, Alice adds a vertex B at t_1. Bob decides to delete this vertex a t_2, and then re-adds it later at t_3. Because of unexpected broadcast system (e.g., network issue), Alice receives Bob's `add B` first. Although B already exists at Alice's replicas, we update its timestamp from t_1 to t_3. Thanks to this, the late received `rem B` at t_2 is not applied since t_3 is newer. Without this timestamp update, vertex B at Alice's replicas would be deleted and leads to an inconsistent state.

4.1.3 CRDT tombstone metadata: isRemoved

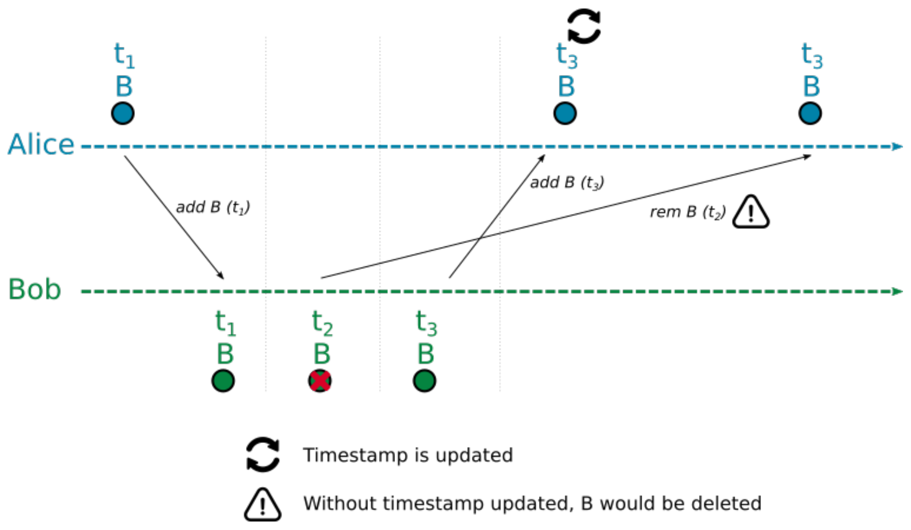


Figure 4.4: CRDT tombstone metadata isRemoved illustrated

CRDT is known as an monotonically increasing only data structure in terms of memory space: elements are never deleted but only flagged as removed. This is formally named the tombstones. In our implementation, it is integrated as a boolean metadata named isRemoved. A value of true means that this elements is flagged as deleted. As an example, in Figure 4.4, Bob is the first to create vertex B at t_1. Because of unexpected broadcast system (e.g., slow network), Alice receives Bob's `add B` operation after she applied `add B` and `rem B`. Thanks to the timestamp and the tombstone, our algorithm detects that this `add B` attempt is older than `rem B` and B remains deleted at all replicas (isRemoved = true as seen on the figure). A naive implementation without tombstones would remove B when Alice applies `rem B` at t_3. Upon reception of `add B` at t_1 from Bob, there would be no possibility to know if this operation was older than the already applied `rem B` operation and B would be re-added, which leads to inconsistency. Therefore, tombstone is mandatory and integrated in our CRDT algorithm.

4.1.4 Return value for CmRDT update method

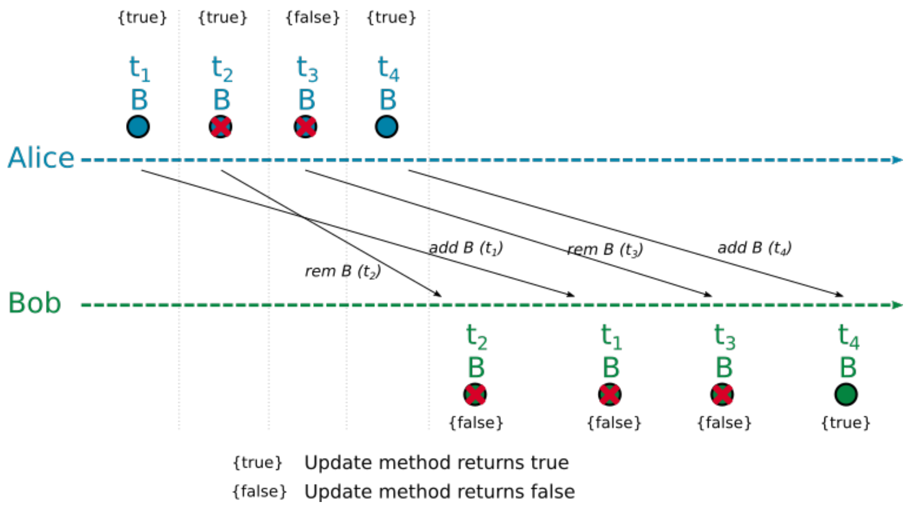


Figure 4.5: Boolean return type for CmRDT update methods

Operation-based CRDT is formally defined in Section 2.3.2.5 as the tuple $\langle S, s_0, q, t, u, P\rangle$. This chapter focus on the update method $u(...)$, which is responsible for applying an operation. Update method has a boolean return value to notify the user about completion of the operation. In the Figure 4.5, it is illustrated by $\{true\}$ and $\{false\}$ marks. Although operations are always applied internally, a user may not see it. Figure 4.5 is an example where Alice applies `add B`, then `remove B`. Both methods return true. The third application `remove B` is technically valid from the CRDT point-of-view since all our operations are idempotent. However, B is already marked as deleted, therefore false is returned so that Alice does not see that operation as applied. Another example to illustrate the use of a return value is the case of delete statement in a SQL database. Without this return value, one would call DELETE statement a second time, which is not the expected behavior. Bob receives `remove B`, then `add B`. Since `add B` is earlier than `remove B`, both methods return false and Bob does not see vertex B until he receives the last `add B` with timestamp t_4 where true is returned.

4.1.5 CRDT internal representation, iterators and special query methods

We saw in Section 4.1.3 that CRDT works with a tombstone metadata that we implemented as a boolean we named isRemoved. In practice, this means that any CRDT element requires at least two slots in actual memory: one for the element and another for its CRDT metadata: timestamp and tombstone. This introduces the need for two versions of the query methods and iterators. In the first version, the query method returns the element only if it is marked as alive (i.e., isRemoved is false), and an iterator iterates over all the alive elements. We refer to these methods as the "normal" version of queries and iterator. Another version returns the internal CRDT metadata along with the requested element, even in case of element marked as deleted. We refer to these methods as the "CRDT" version. In our listings, CRDT version of query methods and iterators have the suffix "CRDT" (e.g., `queryCRDT(...)`).

4.2 CollabServer CRDT Primitives

CollabServer implements a set of CRDTs called built-in CRDT primitives. They all use Last-Writer-Wins paradigm as described in Section 4.1.2. The concrete implementation in CollabServer framework is in C++ 2011 standard. We use C++ templates to hold the final developer data type. However, this chapter explains our algorithm in a common manner without focus for any specific language. All the algorithm listings are in pseudo code.

4.2.1 LWWRegister

Algorithm 1: `lwwregister_update`(value, timestamp)

```
if $timestamp > current\_timestamp$ then
    current_value = value
    current_timestamp = timestamp
    return true
else
    return false
```

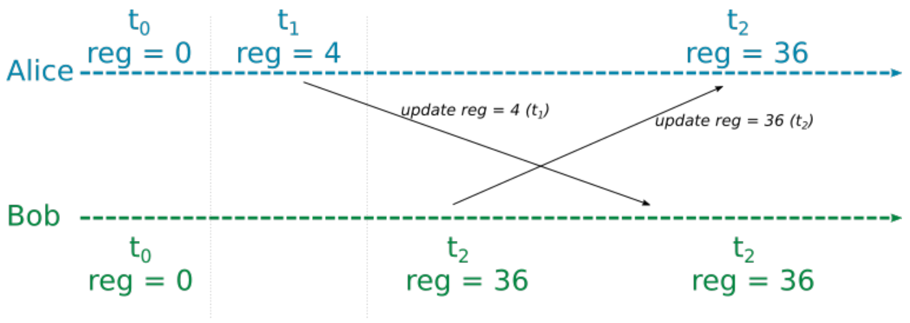


Figure 4.6: LWWRegister: example of collaboration with the update method

This is the simplest CRDT primitive we implemented in CollabServer. It holds one atomic value along with its CRDT metadata (timestamp and isRemoved booean). The `update` method changes the content of the register as a whole, as presented in the algorithm in Listing 1. The `query` method returns the content of the register.

4.2.2 LWWSet

The CollabData LWWSet is a monotonically increasing set data structure that keeps any key added along with its CRDT metadata. Technically, we store our set in a hashmap. The user keys are stored as the hashmap keys whereas the hashmap values hold the CRDT metadata as a couple $\langle timestamp, isRemoved \rangle$. In our algorithm listings, for an element recovered by $element = hashmap[x]$ we assume that $element.key$ is the hashmap key and $element.value$ the CRDT couple.

4.2.2.1 LWWSet query method

Algorithm 2: `lwwset_query`(key)

```
1 element = hashmap[key]
2 if element is not None AND is not element.value.isRemoved then
3     return element.key
4 else
5     return None
```

`Query` returns the key only if it exists in the internal hashmap and it is not marked as deleted (i.e., isRemoved = false), as described line two of the algorithm in Listing 2.

4.2.2.2 LWWSet clear method

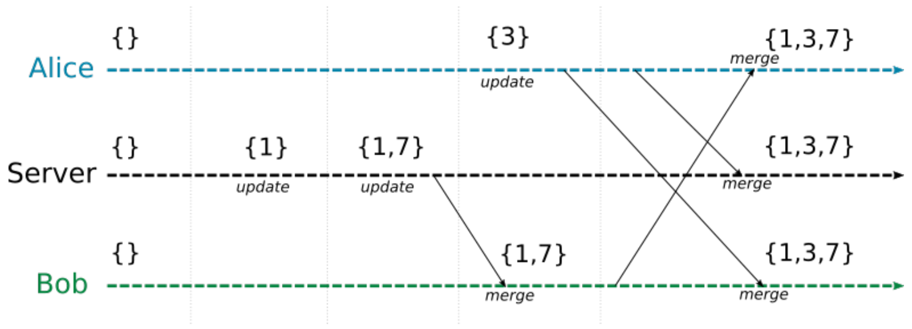


Figure 4.7: LWWSet: example of collaboration where clear call is earlier than add timestamp

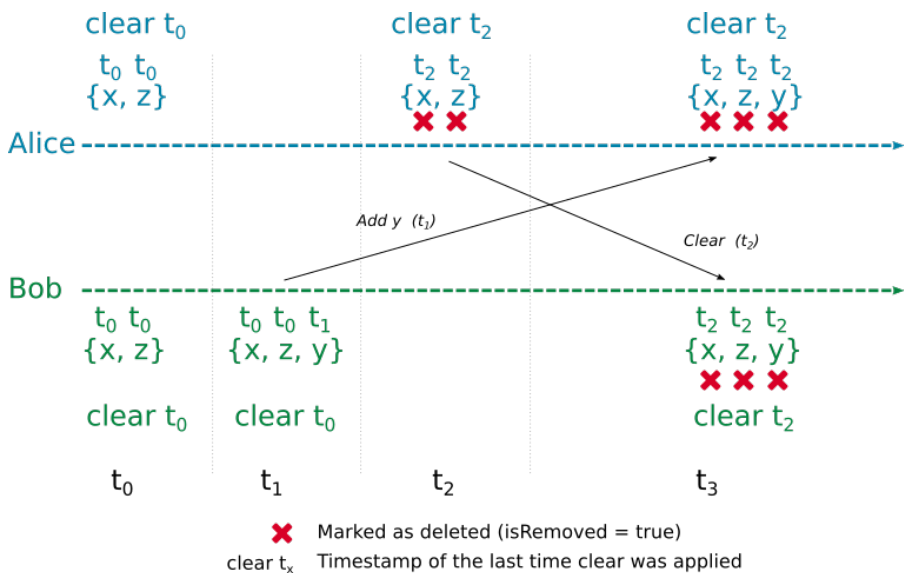


Figure 4.8: LWWSet: example of collaboration where clear call is older than add timestamp

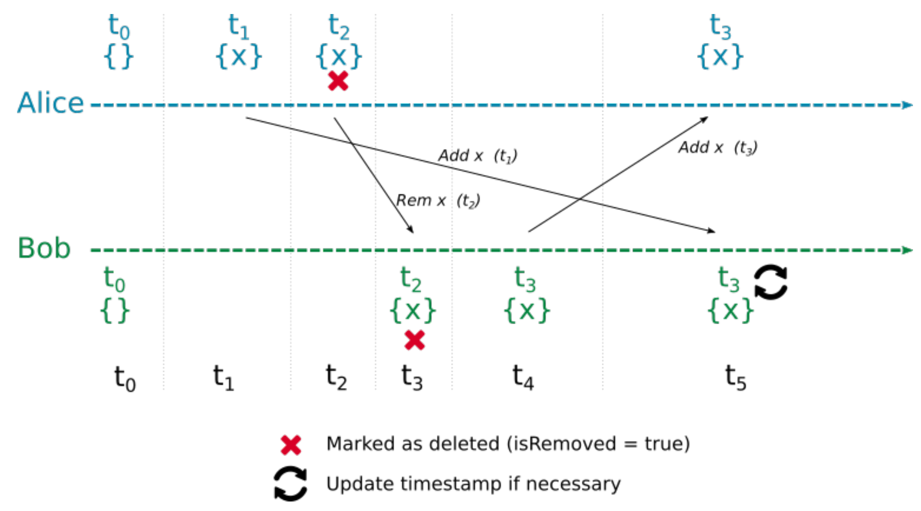
The method `clear` deletes all the elements from the set (i.e., marks elements as deleted). However, `clear` may be concurrent with other add operations, therefore we cannot simply apply a conventional clear. We distinct two scenarios: one scenario where `clear` is older than `add`, as illustrated in Figure 4.7, and a second where `clear` is newer

Algorithm 3: `lwwset_clear`(timestamp)

```
if timestamp > lastclear_timestamp then
    last_clear = timestamp
    for element in hashmap do
        if timestamp > element.value.timestamp then
            element.value.timstamp = timestamp
            element.value.isRemoved = true
    return true
else
    return false
```

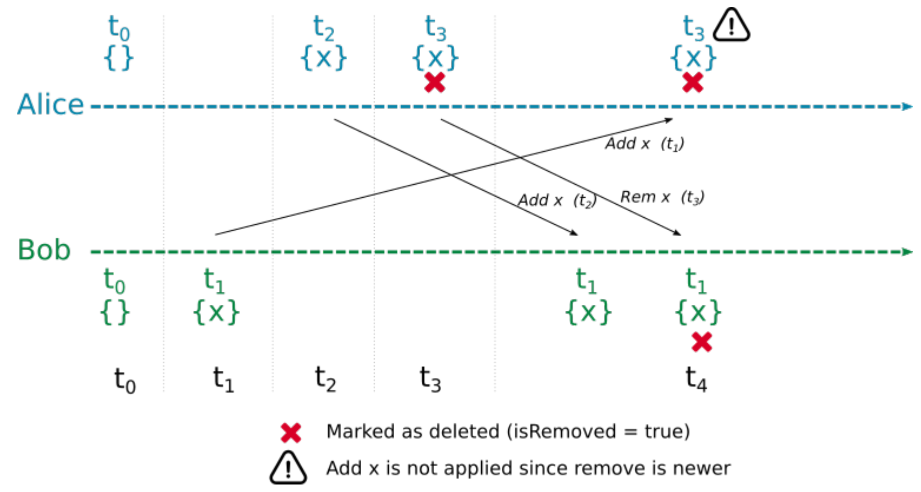
than `add`, as illustrated in Figure 4.8. In the first scenario, Alice clears the set at t_1. At t_2, Bob adds an element y before he receives the Alice's reset. Upon reception of this reset at Bob's replicas, our algorithm detects that y is not to be deleted since its timestamp t_2 is newer than the timestamp t_1 associated with `clear`. In the second scenario however, Bob applies `add y` at t_1, whereas Alice applies `reset` later, at t_2. Upon reception of Bob's operation `add y` at Alice's replicas, our algorithm detects that this `add y` is actually older than Bob's `clear`, therefore y has to be delete, to account the effect of clear. In our algorithm, we keep the timestamp of the last effective clear. This is described at the second line of the algorithm in Listing 3. We consider a `clear` has being successfully applied if it is newer than the last applied clear and true is returned, regardless the actual deletion of an element in the set.

4.2.2.3 LWWSet add method



Alice
Bob
Add x (t1)
Rem x (t2)
Add x (t3)
Marked as deleted (isRemoved = true)
Update timestamp if necessary

Figure 4.9: LWWSet: example of collaboration with concurrent add || add calls



Alice
Bob
Add x (t1)
Add x (t2)
Rem x (t3)
Marked as deleted (isRemoved = true)
Add x is not applied since remove is newer

Figure 4.10: LWWSet: example of collaboration with concurrent add || remove calls

The method `add` inserts an element in the set. In a collaborative context, two scenarios may arise: either two `add` operations or one `add` and one `remove` operations are to be applied concurrently. In the former case, only timestamps are updated and we return false, meaning the key was not added. This situation is illustrated in Figure 4.9 where Bob applies `add x` at t_3, then receives another `add x` with t_1 from Alice, therefore timestamp stays unchanged. The second scenario is illustrated in Figure 4.10: Alice

Algorithm 4: `lwwset_add`(key, timestamp)

```
element = hashmap[key]
if element is not None then
    if $timestamp > element.value.timestamp$ then
        element.value.timestamp = timestamp
        if element.value.isRemoved then
            element.value.isRemoved = false
            return true
    return false
else
    element = {key, {timestamp, false}}
    hashmap.add[element]
    if $element.value.timestamp <= lastclear\_timestamp$ then
        element.value.timestamp = lastclear_timestamp
        element.value.isRemoved = true
        return false
    else
        return true
```

adds x at t_2, then removes x at t_3. Concurrently, Bob added element x at t_1, but Alice receives this operation later at t_4. Upon `add x` reception at Alice's replicas, our algorithm detects that a previous `remove x` is more recent than this `add x`, therefore, nothing is to be done and x remains deleted with its timestamp set to t_3. Our `add` algorithm is described in Listing 4.

4.2.2.4 LWWSet remove method

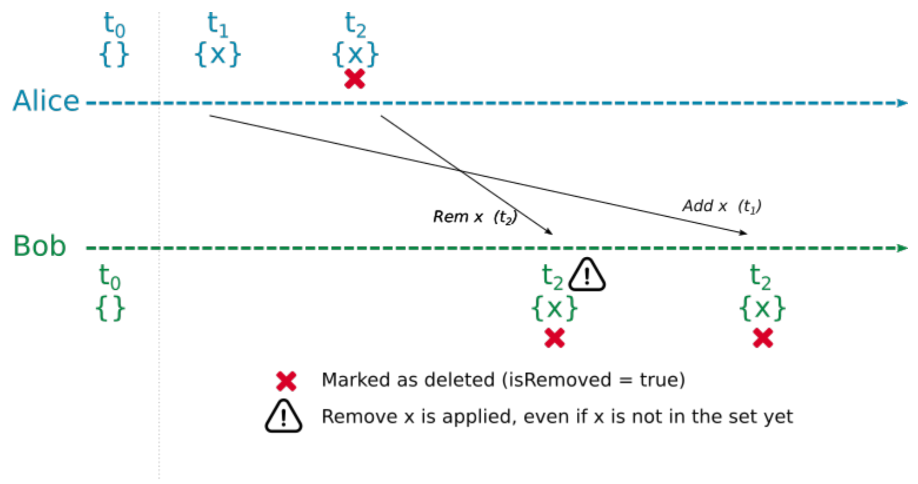

t0
{}
t1
{x}
t2
{x}
Alice
Rem x (t2)
Add x (t1)
Bob
t0
{}
t2
{x}
t2
{x}
Marked as deleted (isRemoved = true)
Remove x is applied, even if x is not in the set yet

Figure 4.11: LWWSet: example of collaboration with remove received before add

Algorithm 5: `lwwset_remove`(key, timestamp)

```
1  element = hashmap[key]
2  if element is not None then
3      if timestamp > element.value.timestamp then
4          element.value.timestamp = timestamp
5          if not element.value.isRemoved then
6              element.value.isRemoved = true
7              return true
8      return false
9  else
10     element = {key, {timestamp, true}}
11     hashmap.add[element]
12     return false
```

The method `remove` marks element as deleted if the `remove` timestamp is higher than the current one. In case this element is already deleted, `remove` simply updates the timestamp and returns false since, from a user point of view, the element was already deleted. If the requested element is not yet present in the set, it is added first, then `remove` is applied. This case is illustrated in Figure 4.11 where Bob receives `remove x`, whereas x is not yet in the local set. All operations are commutative and removed is applied. When Bob receives `add x` with an older timestamp t_1, our algorithm detects that a newer `remove x` has already been applied. This is described in details in Listing 5.

4.2.3 LWWMap

A map is a data structure that defines mappings in the form of a set of key-value pairs. We designed LWWMap as a LWWSet of keys, where each key is mapped to a value. LWWMap follows the exact same algorithm as LWWSet, therefore, only the key is a CRDT. This allows for any value type. If the developer wishes to make values CRDTs as well, he may encode them as LWWRegister, other CRDT primitives available, or create his own CRDT primitive.

4.2.4 LWWGraph

Graph is the most complex CRDT we implemented in the CollabServer framework. It is a directed graph that uses an adjacency list representation: vertices are stored in one hashmap, which is integrated as a LWWMap. The hashmap keys are for the vertices IDs and the hashmap values hold the vertices. ID may be any data type that works with the actual hashmap implementation (e.g., we use C++ templates as key for an unordered_map). Vertex is described as the couple $\langle content, edges \rangle$ where content is the content of the vertex and edges is a LWWSet of all the edges going from this vertex. Each LWWSet key is the ID of the destination vertex. The vertex content type is defined by the developer, which allows for any value type. If the developer wishes to make the content CRDTs as well, he may encode it as LWWRegister, other CRDT primitives available, or create his own CRDT primitive. The most important operations are querying, removing and clearing vertices. Most operations on vertices have an edge operation counterpart (e.g., adding, removing, and counting edges). In our LWWGraph algorithm listings, we refer to the adjacency list by `adj`.

4.2.4.1 LWWGraph queryVertex method

Algorithm 6: `lwwgraph_queryVertex`(vertexID)

```
  /* adj refers to the LWWMap adjacency list        */
1 return adj.query(vertexID)
```

The `queryVertex` method calls the `query` method on the LWWMap adjacency list of vertices and returns the vertex if exists. Vertices marked as deleted are not returned. Listing Listing 6 shows the logic of the queryVertex operation.

4.2.4.2 LWWGraph addVertex method

The `addVertex` method calls the `add` method on the LWWMap adjacency list.

4.2.4.3 LWWGraph removeVertex method

Algorithm 7: `lwwgraph_removeVertex`(vertexID, timestamp)

```
1 removed = adj.remove(vertexID, timestamp)
2 vertex = adj.queryCRDT(vertexID)
3 vertex.edges.clear(timestamp)
4 for vertex in adj.iteratorCRDT do
5     if vertex.edges.has(vertedID) then
6         vertex.edges.remove(vertexID, timestamp)
7 return removed
```

The `removeVertex` method removes a vertex from the graph as well as all edges adjacent to it. If the vertex does not exist yet, it is added in the LWWMap adjacency list, then `remove` is applied. This is important in case `remove` may be received before `add` is called (all operations are commutative). The case when addEdge and removeVertex occur concurrently is handled in the addEdge operation that follows. The `removeVertex` algorithm is described in the listing Listing 7.

4.2.4.4 LWWGraph addEdge method

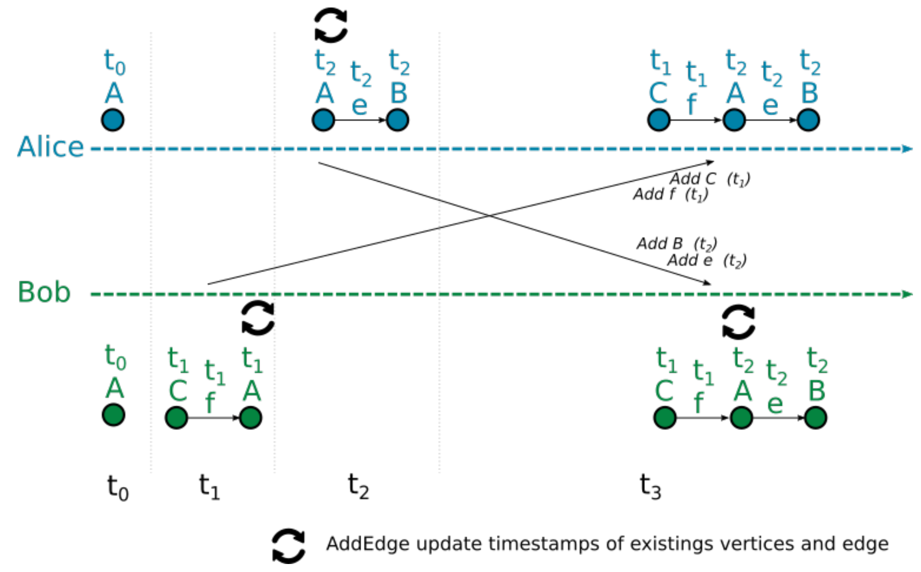


Figure 4.12: LWWGraph: example of addEdge with a simple case of already existing vertices. Vertices timestamps are updated

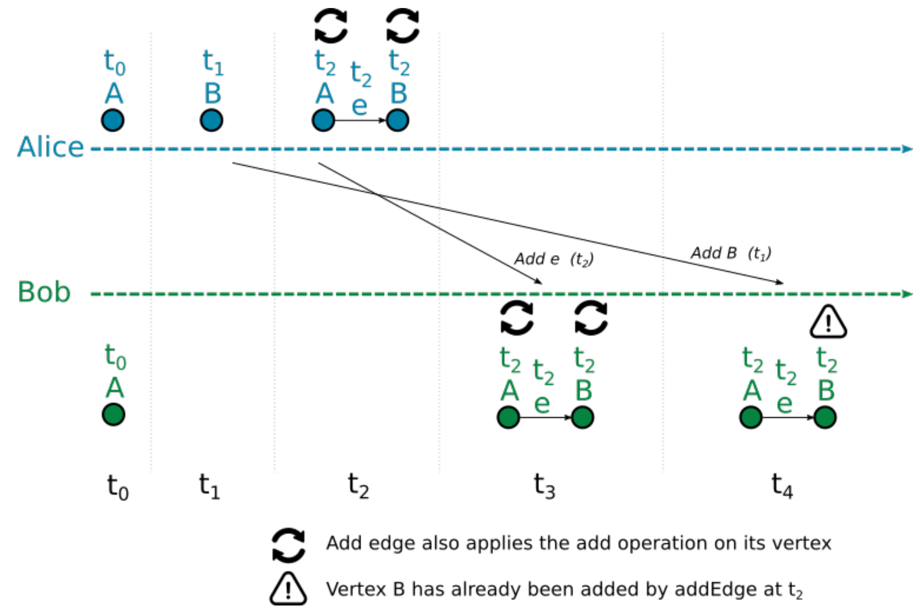

Alice
Bob
t0 A
t1 B
t2 A t2 e t2 B
Add e (t2)
Add B (t1)
t0 A
t2 A t2 e t2 B
t2 A t2 e t2 B
t0 t1 t2 t3 t4
Add edge also applies the add operation on its vertex
Vertex B has already been added by addEdge at t2

Figure 4.13: LWWGraph: example of addEdge with addEdge received before addVertex. Method addEdge re-add the vertices source and destination

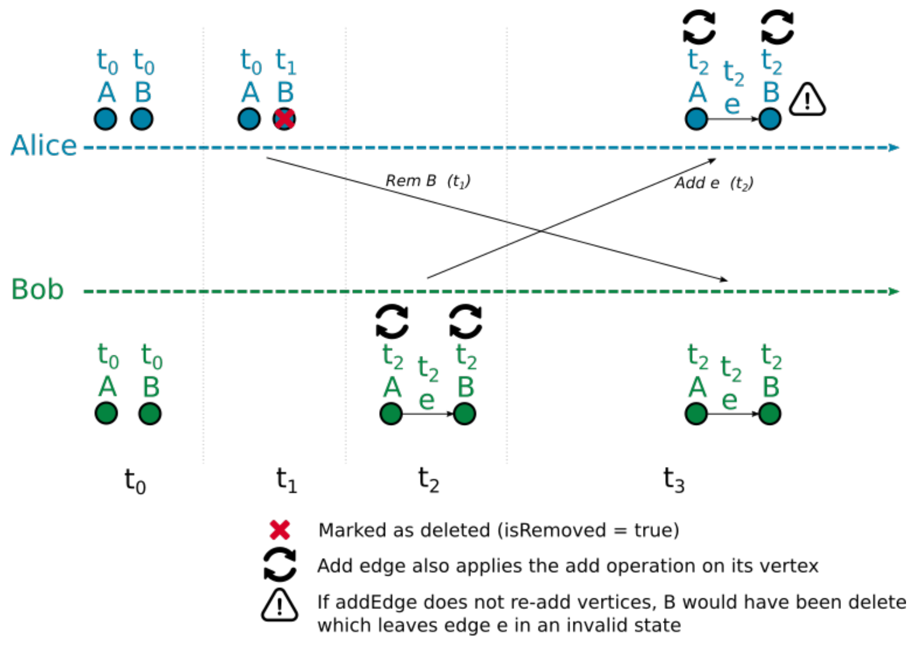

Alice
Bob
Rem B (t1)
Add e (t2)
t0 t1 t2 t3
Marked as deleted (isRemoved = true)
Add edge also applies the add operation on its vertex
If addEdge does not re-add vertices, B would have been delete which leaves edge e in an invalid state

Figure 4.14: LWWGraph: example of concurrent addEdge and removeVertex. Illustrate why addEdge also re-add the vertices source and destination

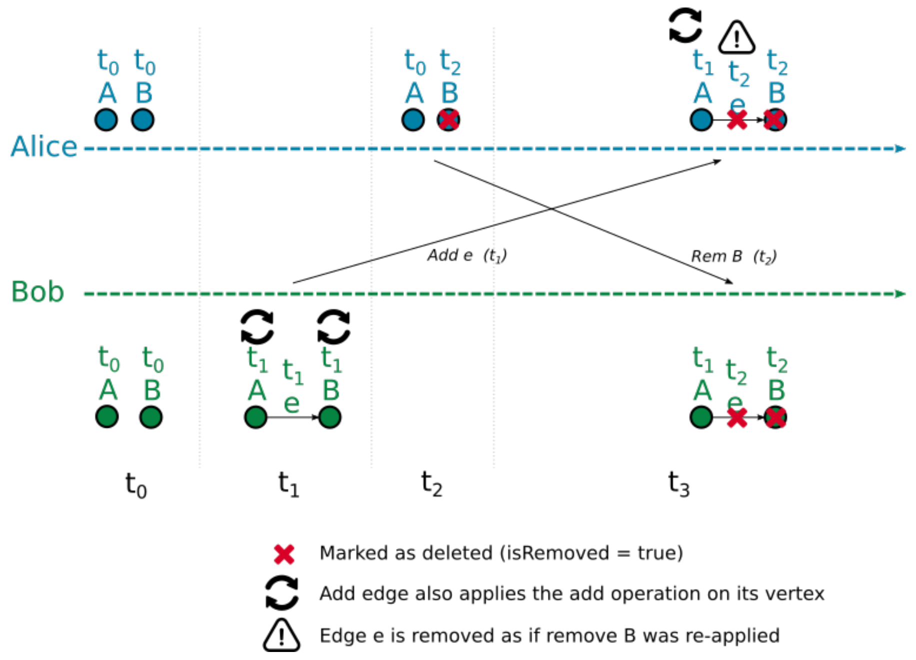

Alice
Bob
Add e (t1)
Rem B (t2)
t0 t1 t2 t3
Marked as deleted (isRemoved = true)
Add edge also applies the add operation on its vertex
Edge e is removed as if remove B was re-applied

Figure 4.15: LWWGraph: example of concurrent addEdge and removeVertex. Method addEdge may delete the newly created edge if one of its vertices are marked as deleted

Algorithm 8: `lwwgraph_addEdge`(source, dest, timestamp)

```
info = {'srcAdded': false, 'destAdded': false, 'edgeAdded': false}
info['srcAdded'] = adj.add(source, timestamp)
info['destAdded'] = adj.add(dest, timestamp)
vertexSrc = adj.queryCRDT(source)
info['edgeAdded'] = vertexSrc.edges.add(dest, timestamp)
if not vertexSrc.edges.queryCRDT(dest).isRemoved then
    vertexDest = adj.queryCRDT(dest)
    if vertexSrc.isRemoved OR vertexDest.isRemoved then
        t = max(vertexSrc.timestamp, vertexDest.timestamp)
        vertexSrc.edges.remove(dest, t)
        info.edgeAdded = false
        return info
return info
```

The `addEdge` method creates a new edge going from a source vertex to a destination vertex. We differentiate three scenarios. The simplest one applies `addEdge` on two existing and alive vertices (i.e., not marked as removed). In this case, the edge is added and, if it already exists, the timestamp is updated as illustrated in Figure 4.12. Another scenario appears when one or both vertices are missing. This means that operation(s) `addVertex` has not been received yet. Because of the commutativity requirement, we must consider this possibility. We decided that `addEdge` also applies `addVertex` on source and destination vertices. Missing vertices are then simply added along with the edge. Receiving a later `addVertex` operation will simply update the timestamps (see Listing Listing 8 [add]). This scenario is illustrated in Figure 4.13 where Bob receives `addEdge e` from A to B before `addVertex B`. Since `addEdge` also re-add vertices, both A and B are added with timestamp t_2 along with edge e, so that Bob applies `addEdge e` without any delay or synchronization. The last scenario is when we receive `addEdge` with the source and/or the destination vertex already deleted. The case where `removeVertex` is older than `addEdge` is trivial since `addEdge` also applies `addVertex` as seen earlier. This is illustrated in Figure 4.14. However, the opposite case (`removeVertex` older than `addEdge`) requires some extra steps. We first naively create the edge as shown in Listing 8. Then, we check if the newly created edge is dangling (i.e., with one or no vertex) and remove it if it is as illustrated in Figure 4.15. With this design, `addEdge` is commutative. Note that, as shown in Listing 8, this operation returns more information since additional actions can be performed on the edge or the vertices.

4.2.4.5 LWWGraph removeEdge method

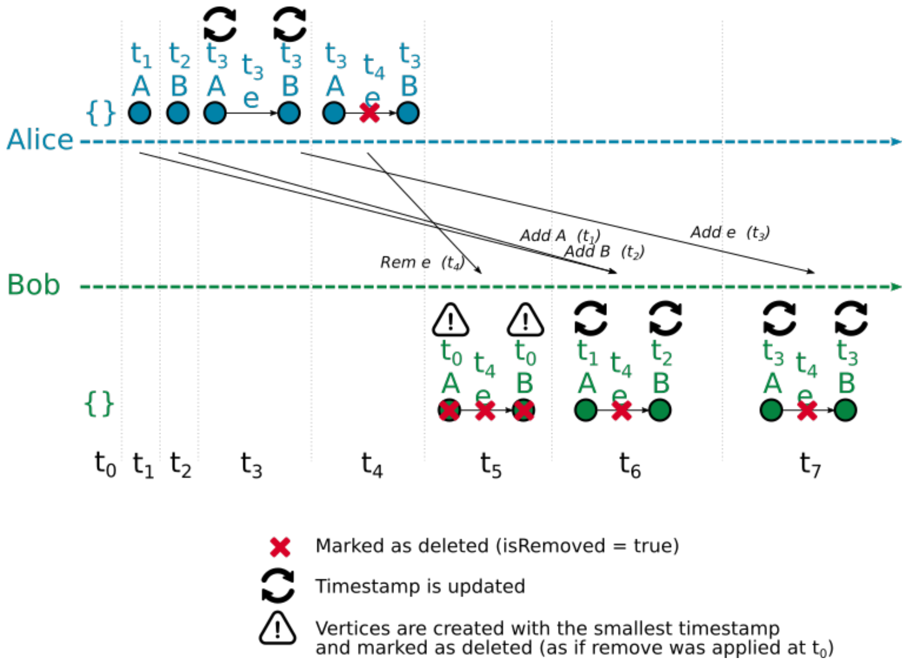

Alice
Bob
{}
Add A (t1)
Add B (t2)
Add e (t3)
Rem e (t4)
t0 t1 t2 t3 t4 t5 t6 t7
Marked as deleted (isRemoved = true)
Timestamp is updated
Vertices are created with the smallest timestamp and marked as deleted (as if remove was applied at t0)

Figure 4.16: LWWGraph: example of removedEdge operation received before addVertex and addEdge

The `removeEdge` method removes an edge from the graph. Removing an edge that has already been re-added will not do anything and returns false (see LWWSet `remove` method in Section 4.2.2). However, this operation may encounter a tricky situation where the source vertex does not exist yet in the graph. This is illustrated in Figure 4.16 where Bob receives the operation `removeEdge e` at the very beginning. All operations

Algorithm 9: `lwwgraph_removeEdge`(source, dest, timestamp)

```
/* Timestamp.MIN gives the minimal timestamp */
1 adj.remove(source, Timestamp.MIN)
2 if $source\,!= dest$ then
3     adj.remove(dest, Timestamp.MIN)
4 vertex = adj.queryCRDT(source)
5 return vertex.edges.remove(dest, timestamp)
```

have to be commutative, therefore, the source vertex is created with the smallest timestamp and the deleted flag set to true so that such operation is hidden from the user, as depicted in Listing 9. This makes `removeEdge` commutative in any situation.

4.2.4.6 LWWGraph clearVertices method

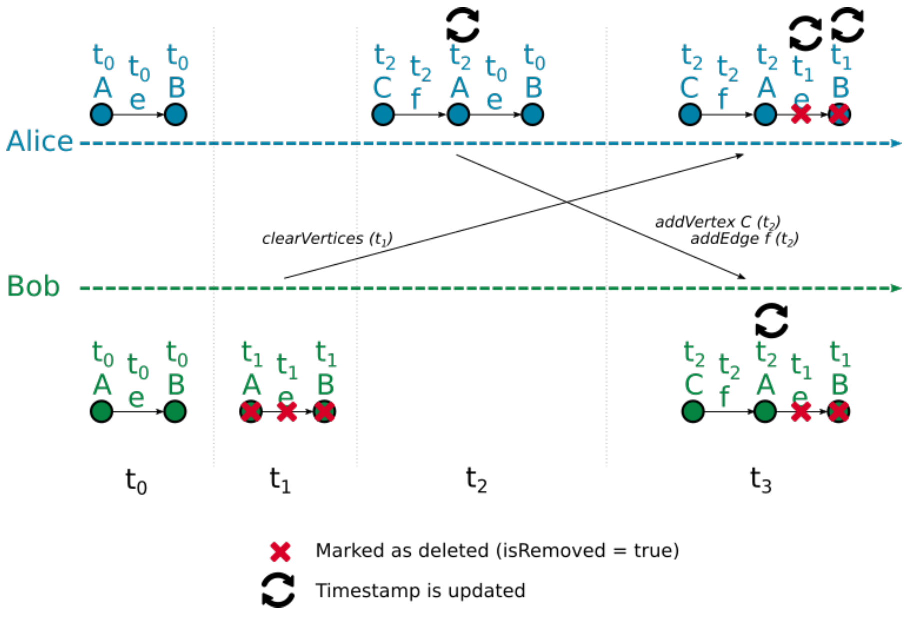


Figure 4.17: LWWGraph: example of clearVertices operation

The `clearVertices` method removes all vertices and their edges from the graph. As seen in `clear` method from LWWSet (Section 4.2.2), clear removes only elements that have smaller timestamps. Vertices with higher timestamps were semantically added after `clear` call and shall not be marked as deleted as illustrated in Figure 4.17. It is not

Algorithm 10: `lwwgraph_clearVertices`(timestamp)

```
1 for vertex in adj.iteratorCRDT do
2     vertex.edges.clear(timestamp)
3 return adj.clear(timestamp)
```

enough to call `clear` on the adjacency list, LWWSet of edges must be cleared as well for each vertex (by calling LWWSet `clear` method on each vertex). This returns true if `clear` has actually been done, otherwise, it means that a more recent `clearVertices` has already been called. Method `clearVertexEdges` is a variation that clears only the edges of a given vertex. This simply calls `clear` on the vertex LWWSet of edges. Listing Listing 10 shows the algorithm for `clearVertices`.

4.3 Summary

In this section, we summarize the design choices of our CRDT algorithm variation and list the CRDTs available in CollabServer. All the methods presented in Table 4.II are designed to have the CRDT properties. As explained in Section 2.3.2.5, CRDT operations have to be commutative, associative, and idempotent. Commutativity requires that a received operation is always valid and applicable on the local replicas without any need of synchronization. This is problematic when an operation is causally related with an oldest operation which is not received yet at the local replica (e.g., receive `remove` before `add` in a `LWWSet`). To address this issue, our general rule is to create the missing elements set with the default CRDT metadata (presented in Section 4.1.5), such as the minimal timestamp possible and the tombstone set to deleted. This guaranties that our methods are commutative in all situations. To have our operations associative, we order them by their timestamp. The state of a data structure upon application of two operations is the result of the newest operation (e.g., the `LWWRegister` content is set with the value of the newest `update`). Therefore, the order in which operations are applied does not change the final result. Idempotent is guaranteed using the timestamp value. Our timestamp are required to be strictly unique. Two operations with the exact same timestamp are, by design, le same operation, therefore the operation is not applied again.

Design choices	**Short description**
Operation-based CRDT	Changes are propagated as atomic operations.
Last-Writer-Wins (LWW)	Timestamps are used to choose the prevailing operation in case of conflicts.
CRDT tombstone metadata	Deleted elements are only flagged as deleted.
CRDT update method return value	Update method has a boolean return value to notify the user about completion of the operation.
CRDT internal representation	Any CRDT element requires at least two slots in actual memory: one for the element and another for its CRDT metadata: timestamp and tombstone. This introduces the need for two versions of the query methods and iterators.

Table 4.I: Summarize the CollabServer CRDT algorithm design choices

CRDT Data Structure	**Methods**

LWWRegister	`query()`
	`update(value, timestamp)`

LWWSet	`query(key)`
	`clear(timestamp)`
	`add(key, timestamp)`
	`remove(key, timestamp)`

LWWMap	`query(key)`
	`clear(timestamp)`
	`add(key, value, timestamp)`
	`remove(key, timestamp)`

LWWGraph	`queryVertex(vertexID)`
	`addVertex(vertexID, timestamp)`
	`removeVertex(vertexID, timestamp)`
	`addEdge(source, destination, timestamp)`
	`removeEdge(source, destination, timestamp)`
	`clearVertices(timestamp)`

Table 4.II: Summarize the CollabServer CRDTs

CHAPTER 5

COLLABSERVER FRAMEWORK

Our variation of CRDT algorithm is presented in chapter 4 and describes how concurrent editing is resolved using optimistic concurrency control and guarantees Strong Eventual Consistency. As seen in Section 2.3.2.5, CRDT is difficult to implement for more complex data types, such as graph. Its complexity makes hard to design and implement new data types from scratch. In this chapter, we present the CollabServer framework which gives tools to build new collaborative data types and setup a real-time collaboration with Strong Eventual Consistency. We describe its architecture and usage, illustrated by our example of graph editor.

5.1 Overview

CollabServer is a framework for real-time collaboration on extensive data types. It supports theoretically any kind of data structure that fulfills the requirements described in this section. It is designed to help users create a collaborative environment for their specific data structure with minimizing the effort needed. The whole system is implemented in C++ 2011 Standard and available online[1].

5.1.1 Requirements

On a local setup, only one user works on his data, which is often located on the same physical system. Operations are applied sequentially by this unique user, which removes possibility of conflicts altogether. This is not the case for distributed data since several remote users may apply operations concurrently. There is no longer a unique sequential order of operations, therefore operations may conflict with each others. Remote users are located on multiple distant nodes and operations are sent over the network. This requires a system that is resilient to high latency network as well as user disconnection

[1] `https://github.com/geodes-sms/CollabServer/`

// reconnection. In a collaborative environment, local users must receive changes from each other so that data eventually converge to the same state.

5.2 CollabServer framework architecture

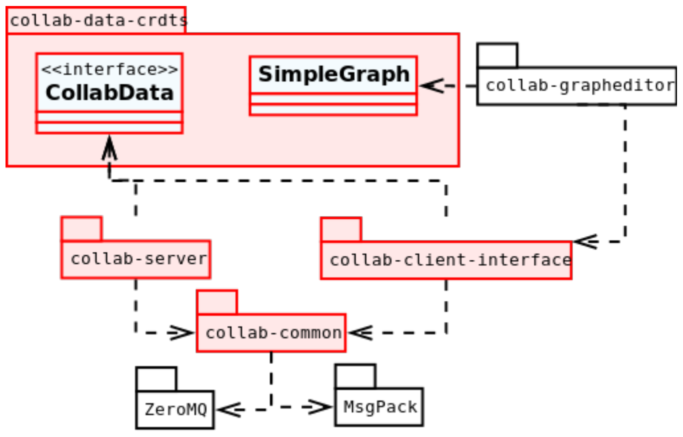


Figure 5.1: UML Package diagram of the CollabServer framework showing dependencies. Packages in red are part of CollabServer.

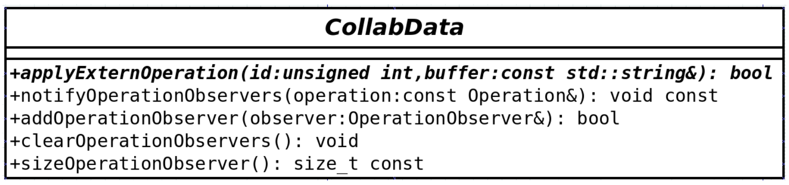
We have designed the CollabServer framework using good object-oriented design principles, such as GRASP [38], SOLID [51], and design patterns [24]. The CollabServer framework is divided into several packages, each of them with a precise and specific task to complete to maximizing their cohesion. Dependencies between them are ensured through interfaces to minimize coupling. The *collab-data-crdts* package defines concurrent data structures and hides its complexity from the rest of the system through simple interfaces. The *collab-server* package is only responsible for multi-user collaboration on one or several data structure. The *collab-client-interface* package is the developer interface to connect with a *collab-server*. The *collab-common* package is an internal component used by both *collab-server* and *collab-client-interface*, mostly responsible of the networking and messaging system. Figure 5.1 shows the dependencies between packages. We rely on two external libraries, ZeroMQ[2] (for network communication) and MsgPack[3] (for message bit-packing). The *collab-grapheditor* package is

[2]https://zeromq.org/
[3]https://msgpack.org/

our example of CollabServer framework and is not part of the framework. SimpleGraph is an example of end-user graph data structure used by our *collab-grapheditor*. Is is integrated in the framework as a built-in example.

5.2.1 Data Structures: collab-data-crdts

Data structures for CollabServer framework are isolated in this distinct and independent package called *collab-data-crdts*. It is in charge of concurrency control and data structure operations. Real-time collaboration involves several users working on the same data structure, but *collab-data-crdts* only deals with concurrent editing and conflicts resolution. This is were our algorithm described in Chapter 4 is implemented and all its complexity is hidden from the rest of the system. It supports the following entities: CollabData, Operation, OperationHandler, OperationObserver, LWWGraph, LWWMap, LWWSet, and LWWRegister.



```
CollabData
+applyExternOperation(id:unsigned int,buffer:const std::string&): bool
+notifyOperationObservers(operation:const Operation&): void const
+addOperationObserver(observer:OperationObserver&): bool
+clearOperationObservers(): void
+sizeOperationObserver(): size_t const
```

Figure 5.2: UML Diagram for CollabData abstract class

CollabData is an abstraction of collaborative data usable in our framework and presented in Figure 5.2. Any data that implements this abstract class is recognized by both *collab-server* and *collab-client-interface*. As an example, *collab-client-interface* expects to register a CollabData and does not know about its concrete developer implementation. These components only deal with abstraction. CollabData is a CRDT, applying the same set of methods on two replicas in any arbitrary order will converge to the exact same state. To guarantee CRDT, one may implement CollabData. A concrete CollabData has a set of operations (defined by developer) which describes all possible modifications (e.g., add, remove value in a set).

Operation is an interface. It describes an atomic change (e.g., add element in map). Any concrete CollabData has a user-defined set of operations that represents all possible modifications. Each of them implements the Operation interface so that it is usable in our framework.

OperationObserver is an interface. CollabData uses the observer pattern [24, 58] to inform about changes. A modification on CollabData generates an Operation to describe this change. It then notifies all its registered OperationObserver with the newly applied Operation. A common usage is to update a GUI, or a database. Thanks to the design of CollabData, notified operations are always valid with respect to causality order (e.g., add must have been applied before remove). This behavior is explained in further details in Section 5.3.1.

OperationHandler is an interface that helps setting up an OperationObserver using a visitor pattern [24, 59]. When an Operation is notified, the exact concrete Operation must be recovered. A simple solution uses a switch-case statement on the operation unique ID (IDs are defined by the developer). This strategy has the advantage that some developer may be interested only in a subset of operations but may be hard to maintain. On the other hand, OperationHandler gives a faster, and safer way to deal with all possible operations whenever notified. It is a variation of the visitor pattern.

LWWGraph, **LWWMap**, **LWWSet**, and **LWWRegister** are CRDTs. In our CollabServer framework, these built-in CRDTs are called CRDT primitives and are described in Chapter 4. The complexity of CRDT algorithm is hidden inside these primitives so that the developer is able to create his own data structure on top of these primitives. The developer has to build his end-user data on top of these primitives.

5.2.2 Client: collab-client-interface

This package provides a set of methods to easily communicate and work with a *collab-server*. It allows developer to connect and disconnect with a running *collab-server* instance. In case of temporary network failure, disconnect // reconnect is supported. Upon successful connection, developer may start a collaboration for one data artifact or join an existing one. Existing collaboration is selected using its unique ID generated at creation time. One may leave a joined collaboration at any time. It is only possible to join one collaboration at the same time. Thanks to the CollabData interface abstraction described in Section 5.2.1, the CollabServer framework does not need to know about exact developer data structure implementation and collaborative complexity is hidden from the developer.

5.2.3 Server: collab-server

The *collab-server* package enables end-users to load and work on the same data together in real-time. Multiple data may be loaded simultaneously by *collab-server*. They are not restricted to a specific concrete implementation since *collab-server* only knows about the CollabData and Operation abstractions. On the server side, an ongoing collaboration is called a room of collaboration which is composed of one CollabData and several users. A user registered in a room receives operations that others apply and *collab-server* ensures that all operations are received by all users at least once. In case of slow network or disconnected user, messages are asynchronously queued until the connection is back online. This strategy is also implemented in the Yjs collaboration framework [44]. This way, users with network failure do not affect the global collaboration speed. Technically, a room keeps a log of all applied operations to provide new user joining the room with all previous operations and converge its data to the latest state.

5.2.4 Network: collab-common

Several components are used by both *collab-server* and *collab-client-interface* such as networking and messaging system. To avoid code clone and ease maintenance, they are regrouped into a unique package hidden to the developer. The whole messaging system is implemented here and defines all possibles messages. To be sent over the network, messages are bitpacked using the open source library *msgpack*. Interface abstraction of the network allows the rest of the system to uses the messaging system without any knowledge about concrete network specificities and implementation. The *ZeroMQ* library is used for networking, but is wrapped into an interface. All these components are internally used by *collab-server* and *collab-client-interface* and never required by the developer. Technically, *collab-common* is compiled as a C++ object library and placed directly inside *collab-server* and *collab-client-interface* source code so that no external linkage is required.

5.3 CollabServer framework integration

In this section, we describe how a developer would integrate the CollabServer framework in a new or existing project to add collaborative features. This may be separated into several sub tasks as follow:

1. Create timestamp with total-order
2. Build custom CollabData
3. Add OperationObserver in end-user application

5.3.1 Understanding CollabData Operation notification system

5.3.1.1 Notification system overview and purpose

CollabData uses a notification mechanism to separate "CRDT world" from "non-CRDT world". In the former, operations may be received in any arbitrary order thanks to CRDT properties (commutativity, associativity, idempotent). On the other hand, "non-CRDT world" respects causally related order of operations: one cannot apply `remove` before the `add` in database or GUI update. To achieve this during collaboration, OperationObserver only notifies valid operations from a "non-CRDT" point of view so that the developer may simply process it as a valid sequential operation. This allows to use the CollabServer framework with databases without the need to change its internal structure. This is because permanent storage does not save CRDT internal metadata (such as deleted items) and only valid operations from OperationObserver are applied. CRDTs are known to grow without bound [39]. Restarting new collaboration removes all CRDT metadata, which reduce this issue. This however would make offline collaboration hard to integrate, therefore our current implementation of CollabServer only integrates real-time collaboration.

5.3.1.2 Notification system details

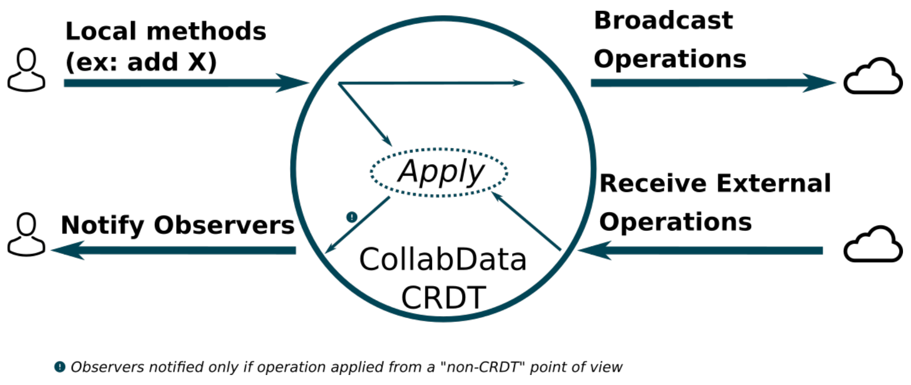


Figure 5.3: CollabData Operations notification system

Technically, all operations are internally applied immediately in CollabData regardless their original source (SEC property in Section 2.1.6). This is described in Figure 5.3: local methods (e.g., add / remove element in set) generate the related operation, which is both broadcasted to others and applied locally, whereas received operations from remote users (i.e., external operation) are only applied locally. Upon application, CRDT primitives are able to detect whether an operation has been successfully applied from a "non-CRDT" point of view, as explained in Section 4.1.4. This is depicted in the figure where user is notified only if operation was applied from a "non-CRDT" point of view. As an example, receiving `remove X`, then `add X` in a LWWSet is internally valid and applied, but each method returns false as if the operation was not applied (the OperationObserver is not notified). This is because `remove X` delete `X` which does not exists on this replica yet. Then, LWWSet detects that `add X` does not need to be applied, by comparing the timestamps of remove and add. As a result, "non-CRDT" will not see any of these operation.

5.3.2 Create timestamp with total-order

As explained in Section 5.2.1, our built-in CRDTs primitives use Last-Writer-Wins paradigm for conflict resolution. These CRDTs use timestamps with total-order so that all operations are ordered in the same exact sequence on all replicas. Because two timestamps may be equals (e.g., concurrent update at exact same time), another metadata should be mixed with time, for instance a unique user ID. Such cases may be rare, but have to be resolved with total-order. Total-order is important to ensure that, ultimately, all data converge to the exact same state. CRDT primitives from *collab-data-crdts* use C++ templates for timestamps so that the developer can implement his very own[4]. As an example, Timetstamp is a fully working example implemented for our SimpleGraph. (See Section 5.4.1 for its description). Another example would be to use the C++ chrono time mixed with the user MAC address.

Although these CRDT primitives are meant to be used with LWW semantic, the developer may technically create timestamp with another paradigm, as far as a total-order exists. For instance, a simple operation counter mixed with user ID. The YATA algorithm [44] from Yjs introduces a fully working example of variation using position metadata of an operation in a list. By applying the same rule on all replicas, the list of elements ultimately converges.

5.3.3 Build custom CollabData

The CollabServer framework works with extensive data type, meaning that developers are not limited to a restrictive set of data structures. This may range from already implemented and ready to use SimpleGraph to user specific data (e.g., Todolist, XML...). To be used in the system, data must fulfill the following requirements:

1. Extend, Define CollabData (described in Figure 5.2)
2. Defines a set of Operations
3. Edits have CRDT properties (associativity, commutativity, idempotent)
4. Optional: provide with an OperationHandler

Custom data is built using the *collab-data-crdts* package described in Section 5.2.1. The developer has to extend CollabData to be usable in our framework. This abstract class

[4]One has to overload C++ operators `operator>` and `operator<` as explained in the technical code documentation online

gives ready to use methods to add observers and be notified about operations as explained in Section 5.3.1. Then, the developer has to define a set of all possible operations on his data. Next task is to design internal data content on top of our built-in CRDT primitives. Custom abstract data-types can be constructed by combining the available CRDT primitives. As an example, a CRDT `LWWMap<int,string>` may be built from `LWWMap<string, LWWRegister>`. LWWRegister itself being set with string as content type (C++ template). However, it is possible that a required primitive is missing from our implemented set (e.g. LWWTree). In such case, it must be added in *collab-data-crdts*, which is a complex task and requires extensive knowledge about the CRDT algorithm [39, 50, 57]. Fortunately, it is often possible to create custom data from our built-in CRDT primitives.

5.3.4 Integrate *collab-client-interface*

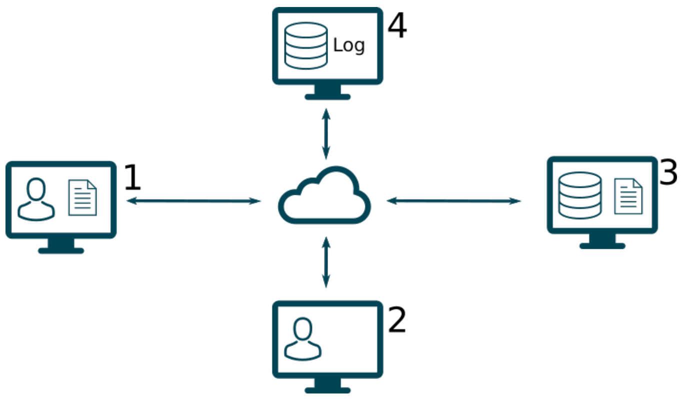


Figure 5.4: Example of collab-client-interface integration by several end-user clients with different roles

As explained in 5.2.2, *collab-client-interface* provides all the required methods to collaborate on one CollabData. Developers must integrate *collab-client-interface* and *collab-data-crdts* in their software. CollabServer does not differentiate between clients. Each of them is seen as a simple user (component that implements *collab-client-interface*) regardless its actual end-user role (e.g., database, editor). Figure 5.4 presents an example of a possible configuration where several clients work on a data, each with a specific goal. User 1 applies modifications and keeps a local copy of the data, whereas user 2 only works on a live instance, which is lost at the end of the collaboration (e.g., gives temporary help to the user 1). Users 3 is a database that saves changes in a permanent storage. The last user integrates a login system that keeps trace of each operation applied during the ongoing collaboration. The common element with these four clients is that they all integrate *collab-client-interface*. It is only a matter of how operations received from OperationObserver are interpreted. A user may updates his GUI and saves the data in a file (user 1) whereas others only update their GUI (user 2). It is also possible to process operations and save the data to a local storage (user 3). As an example, to support a SQL database, one may translates the received operations into SQL statements. (e.g., `add X` in a LWWSet may be translated into an SQL INSERT statement).

5.4 A complete working example: GraphEditor

In this section, we present GraphEditor, a command line tool for graph editing that supports real-time collaboration using the CollabServer framework. It relies on *collab-client-interface* to connect with a running *collab-server* instance and implements CollabData (from *collab-data-crdts*) to create the SimpleGraph data structure. The GraphEditor package is named *collab-grapheditor*.

5.4.1 Timestamp

Algorithm 11: operator<(current, other)

```
1 if if(current.time == other.time) then
2     return current.id < other.id;
3 return current.time < other.time;
```

GraphEditor has its specific implementation of timestamps called Timestamp. It uses the C++ chrono library mixed with a unique user ID (*collab-server* gives unique ID for each user and may be recovered by client). In most of the cases, only the chrono value is enough to create a total-order of timestamps. (at the milliseconds). However, in case of several users working in real-time, two users may end up with the exact same chrono value. In such cases, the ID is used to select a winner to ensure timestamps have a total-order. Although this situation is rare, users would end up with divergent data if not supported. The selection of the ID is deterministic to ensure repeatable behavior (this is mandatory for CRDT properties). As an example, the Listing 11 shows the method `operator<` that returns true if current Timestamp is smaller than the other Timestamp.

5.4.2 SimpleGraph

SimpleGraph is an example of fully working CollabData built from *collab-data-crdts* as explained in Section 5.3.3. It is a graph data structure that uses our CRDT primitives to deal with concurrent editing. Each vertex has a name (C++ string) and a map of key-value (keys and values are both C++ strings). Internally, it is made of one LWWGraph. Each vertex is a LWWMap of (key:string, value:LWWRegister), register being itself of type string. All of them use Timestamp. Timestamps are explained in Section 5.3.2.

5.4.3 Integration with CollabServer

The GraphEditor uses a command pattern to expose a set of commands. A user calls them to start or enter a room of collaboration and edits a SimpleGraph. Changes are updated in the GUI thanks to our observer pattern described in Sections 5.3.1 and 5.2.1. At that point, SimpleGraph may be seen as a normal local data. Each command on SimpleGraph is merely a call to the right method in SimpleGraph. All the collaboration (e.g., broadcast operation, receive operation) is taken care of behind the scenes.

5.4.4 Support with Modelverse database

The Modelverse [64, 65] is a database for Model-Driven Engineering. The Modelverse considers all artifacts as models. They can be manipulated with typical CRUD operations. It runs as a standalone Python code on a server and may be accesses through its REST API. From CollabServer point of view, our Modelverse database is yet another client that implements *collab-client-interface* as explained in Section 5.3.4. Figure 5.5

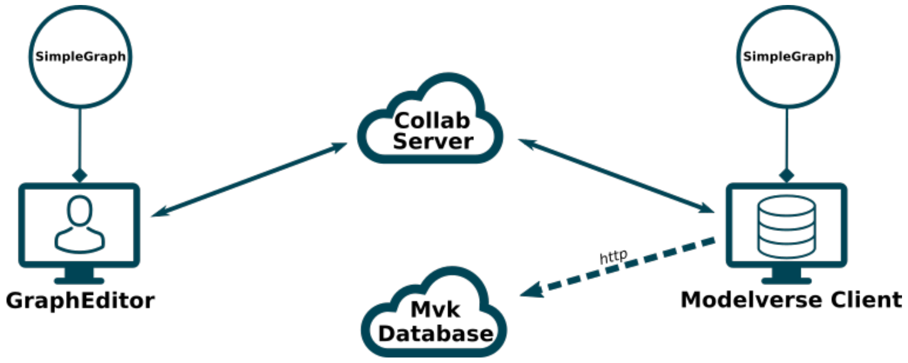


Figure 5.5: GraphEditor collaboration scenario using CollabServer framework and Modelverse database

shows that GraphEditor and Modelverse clients are connected to the *collab-server* and both run an instance of SimpleGraph. Upon reception, a SimpleGraph Operation notified by the OperationObserver is translated into the Modelverse semantic and sent to the Modelverse database. To have a SimpleGraph saved in the Modelverse during a collaboration, a user has to start a Modelverse client first, then, he has to create the collaboration from the Modelverse client before joining it.

CHAPTER 6

EVALUATION

In this chapter, we evaluate our CollabServer framework to answer the following questions:

"Does CollabServer framework successfully help building collaborative environments?". To answer this question, we created a fully functional example of collaborative software called GraphEditor and described in Section 5.4. We then evaluated the modularity of CollabServer.

"Is CollabServer reliable and conforming to the technical requirements for collaboration?". To answer this question, we designed unit tests covering a maximum range of possible scenarios. Moreover, we discuss threats to validity and possible improvements.

6.1 CollabServer framework analysis

In Chapter 5, we saw that our CollabServer framework is divided into several distinct packages, each with a specific purpose. In this section, we present a study analysis of our framework to validate this design choices using several software design metrics, such as abstractness, and instability. We present our measures and discuss their relations and implications in our design such as coupling and cohesion of our packages.

6.1.1 Packages metrics description

In Software Engineering and Object-Oriented Programming (OOP), a set of classes may be regrouped in a so called package. We distinguish four domains of classes for packages: foundation, architecture, business, and application. Foundation domain regroups classes with lowest dependencies such as primitive types, data structures, and generic components (e.g., Boolean, Integer, Stack, Tree, Date, Time…). They me be used by all the other domains and do not depend on other packages. The architecture domain regroups classes that are more specific to a kind of component such as network library (e.g., port, socket), database (e.g., transaction, backup), or user interface (e.g., window, button, widget). The business domain is specific for one kind of business such as classes with unique role (e.g., account, car...). These classes may still be used in several applications. The last domain of classes is called application domain and regroups classes that are unique to one specific application.

In our CollabServer analysis, we are interested by three metrics: abstractness, instability and distance. Abstractness is defined as follow

$$A = Na/Nc \tag{6.1}$$

with Na the number of abstract classes in the package and Nc the total number of classes in the package. Instability is defined by

$$I = Ce/(Ca + Ce) \tag{6.2}$$

where Ce represents the efferent couplings (outward dependencies) and Ca the afferent couplings (inward dependencies). $I = 0$ means that the package is very stable (it depends on no other package) whereas $I = 0$ is for really dependent packages. Distance is calculated from A and I as follow:

$$D = |A + I - 1| \tag{6.3}$$

It is ideally equals to zero. All these metrics ranges are 0 to 1.

6.1.2 Packages metrics analysis

The diagram of dependencies between packages is presented by Figure 5.1, we are only interested in the CollabServer components (marked in red on the figure). Since *collab-grapheditor* is an example of end-user application, we omit it in this study. Our measurements are placed in the Table 6.I. Figure 6.1 presents the resulting AI graph. There is no critically abnormal values (value far from "The Main Sequence", which is

Packages	Abstractness	Instability	Distance
collab-data-crdts	0.40	0.00	0.60
collab-common	0.67	0.25	0.08
collab-client-interface	0.00	1.00	0.00
collab-server	0.17	1.00	0.17

Table 6.I: CollabServer framework packages: table of metrics for abstractness, instability, and distance

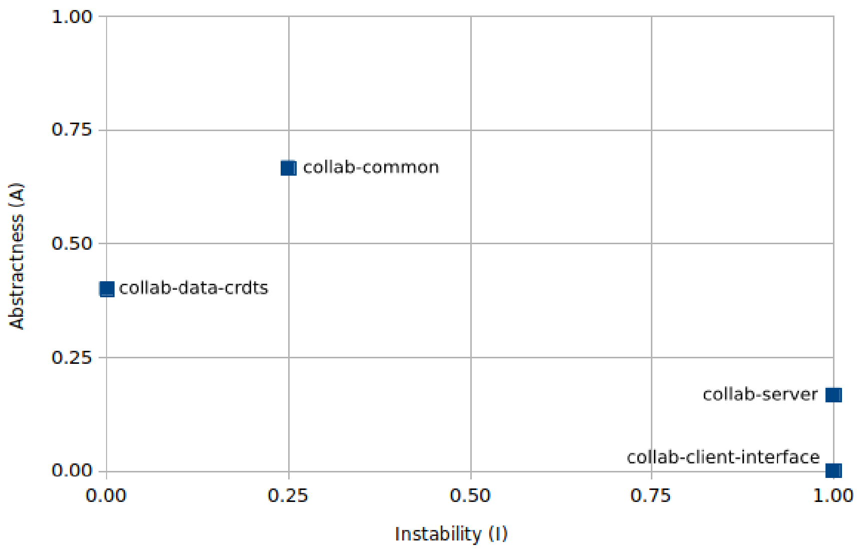


Figure 6.1: AI Graph of the CollabServer packages

a line from `(0,1)` to `(1,0)` positions). Only one value at `(0,0.40)`, which is for the *collab-data-crdts* package, has an unexpected results.

The package *collab-data-crdts* is a business domain package and is designed to be independent and reusable. It is not limited to a specific application. In our analysis, we measured the following values for abstractness, instability and distance: $A = 0.40$, $I = 0$, and $D = 0.60$. Instability result indicates that *collab-data-crdts* is very stable, which is the expected value since it is designed to be reusable and depends on no other packages. However, the distance of 0.60 is far from 0 which warns about a possible incoherency due to its low value of abstractness $A = 0.40$. This is an unexpected value since stable packages tries to be more abstract. Technically, our defined primitives are not abstract classes but, instead, use C++ templates to deal with the user specificity content type. Therefore, they are not counted as abstract in the formula. For that reason, this package appear less abstract, although this is not an actual issue in practice.

The package *collab-common* is a library internally used by the CollabServer framework and is described in Section 5.2.4. In our analysis, we measured the following values: $A = 0.67$, $I = 0.25$, and $D = 0.08$. This package has an high abstraction $A = 0.67$. This is explained by the *collab-common* purpose, which is to provides CollabServer components with the common classes to avoid duplicates. It is meant to be reused by our framework and has a fairly low value of instability $I = 0.25$. Its only dependency with *collab-data-crdts* is achieved through an interface.

The package *collab-client-interface* is a business domain package, it is meant to be used by the applications that use CollabServer. It is described in further details in Section 5.2.2. It has $A = 0.00$, $I = 1.00$, and $D = 0.00$. Although this measurements appears to be optimum, they have to be taken with care, since this package only has one class. This explains the apparently perfect values. One may expect to have higher abstract value since it is reused by several applications. This low abstraction and high instability are due to the fact that the measurements only focus on the CollabServer packages, in which *collab-client-interface* is the less stable. The same measurement with the end-user application would lead to better results. We designed *collab-client-interface* as one unique user interface, similar to a facade design pattern, which explains one concrete class used be several applications.

collab-server is an application domain package. It is a standalone server code and is not meant to be reused. This is emphasized by our measurements $A = 0.17$, $I = 1.00$, and $D = 0.17$ where *collab-server* is very stable and not abstract.

6.2 Usability

We evaluated the usability of our framework through a fully working example of end-user application called *collab-grapheditor*. We distinguish three components to build: a collaborative graph called *SimpleGraph*, a graph editor, and a database proxy. The features and technical details of this example are already described in Section 5.4. The editor and the database proxy have been implemented by two developers with no precedent knowledge about CollabServer and collaborative environments. In the following section, we are interested to understand their feedback and the effort it took them to understand and integrate CollabServer. We only provided them with the official CollabServer and SimpleGraph documentation to evaluate its usability. However, we developed the SimpleGraph as explained in Section 5.4.2.

CollabServer framework makes collaboration complexity transparent to the developers. They should not care about the technical aspect of collaboration and simply use their data as a common local data. For this goal, the concept of Operation described in Section 5.3.1 is critical to understand. Thanks to our exhaustive documentation, it has been easily understood by the two developers. The implementation of *collab-grapheditor* ran out without issues and the two developers where able to successfully conclude their tasks without requesting help. We used this external feedback to analyse our framework design and orchestrate the possible changes, although no refactoring was actually required. This however pointed out several minor issues such as misleading methods name or imprecise documentations. As an example, the `accept` methods for Operation was previously named `handle`, which confused the developers during the OperationHandler component integration.

6.3 Correctness

6.3.1 Tests covering

We built unit tests for each individual CollabServer packages. The package *collab-data-crdts* has the highest number of unit tests to ensure that CRDT properties are satisfied, which is the most critical part in our framework to guarantee Strong Eventual Convergence. Although collaboration software make heavy use of network, our CRDT implementation can be entirely isolated locally on one single thread and fully tested against all the possible collaboration scenario without network. This removes the danger of unexpected network behaviors and network failures (e.g., receive message queue full, dropped message). This allows to fully dissociate network code from collaboration algorithm code, which ease error detection and algorithm validation. Although in practice, many users may generate concurrent operations, they are applied in a sequential order at each replica (this order however may change). Knowing this, any concrete scenario of collaboration may be isolated and tested locally by applying operation in a specific order. Package *collab-data-crdts* has 250 tests which are all successfully passing. Each test covers a scenario and tests the data integrity. The most common tested scenarios are the following (with simplified example of pseudo code):

- Idempotent (e.g., duplicate calls)
  ```
  set.add("e1", T1)
  set.add("e1", T1)
  TEST(set.query("e1").timestamp = T1)
  TEST(set.query("e1").deleted = False)
  TEST(set.size() = 1)
  ```
- Commutative (e.g., remove before add)
  ```
  set.remove("e1", T2)
  set.add("e1", T1)
  TEST(set.query("e1").timestamp = T2)
  TEST(set.query("e1").deleted = True)
  TEST(set.size() = 0)
  ```
- Duplicate add (or remove) with different timestamps
  ```
  set.add("e1", T1)
  set.add("e1", T3)
  set.add("e1", T2)
  TEST(set.query("e1").timestamp = T3)
  TEST(set.query("e1").deleted = False)
  TEST(set.size() = 1)
  ```
- Concurrent operations (e.g., add || remove)
  ```
  // User1:  normal order
  ```

```
setU1.add("e1", 1);
setU1.add("e2", 3);
setU1.remove("ve", 4);

// User2:  remove before add
setU2.remove("e1", 4);
setU2.add("e2", 3);
setU2.add("e1", 1);

TEST((setU1 == setU2) = True)
```

In terms of software maintenance and upgrade, they play an important role to ensure that our algorithm remains validated after any modification. Each commit automatically generates the whole set of tests, using Travis Continues Integration and Github. As an example, one may try to upgrade our CRDT algorithm to get better size or speed performances. Running our tests allows to know whether the algorithm correctness is preserved.

6.3.2 GoogleRealtimeAPI, Yjs, and CollabServer comparison

GoogleRealtimeAPI and Yjs are both javascript frameworks for real-time collaboration. In this part, we make a simple comparison between our CollabServer framework

Data type	**CollabServer**	**Yjs**	**GoogleRealtimeAPI**
register	Yes	No	No
set	Yes	No	No
map	Yes	Yes	Yes
graph	Yes	No	No
array	No	Yes	No
xml	No	Yes	No
text	No	Yes	Yes
richtext	No	Yes	No

Table 6.II: Comparative table of the supported data types in CollabServer, Yjs, and GoogleRealtimeAPI

and these two frameworks. Figure 6.3.2 lists a set of primitives with their support status by each framework.

6.3.2.1 GoogleRealtimeAPI

GoogleRealtimeAPI and CollabServer have both a client/server network architecture. GoogleRealtimeAPI is a web-only API that provides the possibility to collaborate on data that is located in the Google Drive cloud storage. It does not allow the developer to work with his own database or permanent storage whereas CollabServer introduces this possibility. GoogleRealtimeAPI uses Operational Transformations (OT) to achieve collaboration. It has well support for linear data types such as text and list, as seen in Figure 6.3.2. GoogleRealtimeAPI is now deprecated since January 2019 and is no longer in development.

6.3.2.2 Yjs

CollabServer has a client/server network architecture whereas Yjs supports client/server architecture as well as peer-to-peer. Both frameworks use the operation-based version of CRDT to achieve real-time collaboration. The variation of the CRDT algorithm implemented in Yjs is called YATA (Yet Another Transformation Approach). It is based on the list data structure. Each element in the list holds several metadata, such as a unique element ID, the ID of the original left element (i.e., element at the left at the moment of insertion), the ID of the current left element, and the ID of the current right element. The insert operation follows a strict set a rules, and uses these metadata to determine the actual position in the list for this insertion. This is explained in further details in the literature [44]. All replicas apply the same rules so that the insertion method has a deterministic outcome, therefore all replicas ultimately converge. Since this is not based on timestamps (e.g., LWW used in CollabServer), this gives the possibility to fully support offline collaboration out-of-the-box while still supporting real-time collaboration. CollabServer, on the other hand, only works with real-time collaboration. Although its core algorithm is based on the list data structure, it is possible to integrate any data type on top of this list. Yjs has built-in support for several linear data types, such as text and array as well as more complex structured data types, such as XML. However, it has not integrated graph yet. CollabData has built-in support for graph using CRDT algorithm which is a major contribution since only a theoretical graph implementation was introduced in the literature [57]. Yjs is open source and available online[1], moreover, it has an extensive documentation with several hands-on tutorials[2]. As for CollabServer, it is divided into several packages, each with their specific purpose (e.g., Yjs core, xml data, array).

6.4 Potential improvements

6.4.1 Garbage collector

As explained in Chapter 4, CRDT algorithm uses tombstones to mark deleted items. This is a powerful strategy to achieve Strong Eventual Consistency, but also known as its main drawback since the memory usage only grows[28]. To avoid such issue, some systems try to implement a garbage collector. As an example, Yjs introduces a garbage collector [44] to remove items that are marked as deleted at all replicas. This however requires a temporary synchronization. Treedoc is another example that presents a garbage collector [39]. It removes deleted items on "cold" regions of the document. We did not implement such mechanism in our solution and objects with important amount of add / remove operations may become large in terms of memory size. Our data structure size is always at least equal to the size of all the items (alive and deleted). However, our CollabServer framework keeps CRDT metadata (such as deleted items. . .) only during the time of a collaboration. Thanks to our OperationObserver and notification system (as seen in Section 5.3.1), permanent storage does not know about CRDT and works without these CRDT metadata. Therefore only actual living items are saved in permanent storage. A new collaboration reload the data from permanent storage with new CRDT metadata, therefore our implementation is a specially good solution for occasional collaborations.

[1]`https://github.com/y-js/yjs`
[2]`https://y-js.org/`

6.4.2 LWW and offline collaboration

Our CollabServer framework uses the Last Writer Wins rules. This often reduces the amount of metadata and tries to avoid permanently deleted items. As a comparison, Yjs keeps a growing only linked list where deleted items are never reused [44]. In case of doing `add B` between A and C, then `remove B`, and then `add B` again, the resulting linked list is a 4 items length list $\{ABBC\}$, where one B is marked as deleted. CollabServer, in the other hand, reuse existing elements by only changing the timestamps along with its deleted status. As an example, for the same previous scenario, we end up with $\{ABC\}$, with B marked as alive again with timestamp of the last `add` operation. Another illustration is our LWWRegister implementation (which contains one unique atomic value, as seen in Section 4.2.1). Only the last value is saved along with its timestamps. Applying several delete and add operations reuse the same register, with the timestamp and deleted flag updated. Although no garbage collector is implemented, this allows CollabServer to avoid overwhelm of deleted items. Last Writer Wins however, is not adapted for offline collaboration since the last working user would always be the winner of any conflict. In case of such offline setup, the exact edit time may not be the most appropriated solution. Algorithms such as three-way merge 2.3.2.2 or Differential Synchronization 2.3.2.3 are often more appropriated for offline collaboration. Yjs has good result in case of offline collaboration thanks to its monotonic linearization function which is not based on timestamp but relies on a total order of operations. Another solution called Observed-Remove is presented in the original CRDT technical report [57].

6.4.3 Server optimization

As seen in our framework architecture Section 5, a running CollabServer server does not know about any concrete implementation of CollabData, instead, it only keeps the raw operations. This makes the server code totally independent from the clients and may run as a service for several unrelated clients. This however, goes at the cost of loosing several possible optimizations on the server side. In Section 5.3.1, we described the operation mechanism that allows database to receive only the valid operations (from a non-CRDT viewpoint). This may be used at server side as well to reduce the amount of operations sent over the network by only broadcasting whose that are actually applied. For instance, at T_1, `add X` is received, then at T_4, `remove X` is received. Any `add` / `remove` operation on X with T inferior to T_4 will not do anything since X is already removed at T_4. Because of slow network, such operations may be received anyway. With an actual CollabData implementation running on the server side, such operations may be detected as "deprecated" and not broadcasted to others. In the current state, server naively stacks the operation in their raw serialization format (which uses small amount of memory) but it broadcast all received operations to the room of collaboration, regardless its content.

6.4.4 Security

CollabServer does not integrate any security. As an example, messages are bit packed for performance purpose but not encrypted. One may edit a message on the fly and change its content (e.g., netcat) so that an invalid operation is received (e.g., wrong ownership, bad values...). On message reception, there is currently no authentication check and the running CRDT simply tries to unserialize and apply the received operations. Such attack may leads to divergent state at some replicas. In practice, this issue would not brake our system but only introduces unexpected diverging state at some replicas. This is because, in case of invalid operation (e.g., operation which does not exist for the requested concrete CRDT), the serialization process fails and simply bypass this operation. The content of a CollabData remains valid from CRDT and data point of view.

6.4.5 Extensive Data Type

We designed CollabServer to work with extensive data types. We hide the complexity of CRDT behind a set of primitives so that the users are able to create their own data type easily, on top of these primitives, as explained in Section 5.3.3. Our current implementation integrates built-in Graph, Map, Set and Register data types. Although such primitives are enough to build the end-user data type for a wide range of use cases, one may require another primitive such as Array, or List. In this case, the primitive itself has to be built, which is an hard task and requires actual knowledge about CRDT. In all situations, building a new CollabData remains a fairly complex process. The developer has to understand our notification system presented in Section 5.3.1.

6.4.6 Timestamps with different timezones

Collaborative systems may have users from several locations with different local time. For that reason, distributed systems cannot rely on local time for operations that use a unique time value for the whole system. As an example, if Alice is in Montreal and Bob is in Berlin, they both have a different local time. Two operations applied concurrently would have several hours of difference in practice, if only the local time was used. One possibility to fix this issue is to use the same time referent at all replicas (e.g., UTC). In our CollabServer, the developer is in charge of providing its own timestamp implementation, as explained in Section 5.3.2. However, our example of timestamp implementation presented in Section 5.4 (used by our GraphEditor example) uses the local time. This restraints the use of our Grapheditor example in one timezone only.

CHAPTER 7

CONCLUSION

We conclude by summarizing the contributions of this thesis outlining future work. The work presented in this thesis makes several contributions to the fields of Software Engineering and Distributed Systems.

7.1 Summary

Distributed Systems and collaborative software require that the same data may be accessed and modified by several users geographically distant. Concurrent editing and consistency are an issue to take into consideration to achieve collaboration and conflicts resolution. It is important that all replicas eventually converge to the same state. We saw that several algorithms have been developed to achieve this goal such as Operational Transformation (OT) and Differential Synchronization, as seen in Section 2.3.2.4, and Section 2.3.2.3.

In our work, we first studied all the features required to build a collaborative environment (Chapter 3). This lead us to a complete feature model for collaborative environments that we used to select the most appropriate components for our CollabServer framework. As an example, the network architecture and conflict management are some key points and must be chosen with care. We hope this feature model may help developers to make decisions when designing collaborative software.

We described the implementation of our algorithm used by CollabServer for concurrency control in Chapter 4. It uses the CRDT principles so that no consensus, locking or synchronization is required to achieve SEC. The conflict resolution is done locally at each replica, by following a strict set of rules, which guarantee that all replicas eventually converge. Although the CRDT algorithm has a really powerful design that ensures SEC, it may be hard to design new and complex data. We presented our implementation for Register, Set, Map and Graph.

Building collaborative environments is a difficult task, specially when the collaborative data with concurrency control algorithm has to be built from scratch. We presented our CollabServer framework (in Chapter 5) that helps developers in this task. It has a modular architecture that separates the technical tasks according to the developers expertises. We provide the developer with a set of CRDT primitives, usable to create his own data type without having to understand the complexity of CRDT.

At last, we evaluated our framework to validate its usability and correctness in Chapter 6. Through the practical work of several developers, we evaluated their experience using CollabServer. The overall appreciation was rather positive and the integration of CollabServer has been achieved without major issues. The correctness of our algorithm has been validated by 250 unit tests that cover the most relevant collaborative scenarios.

7.2 Outlook

Currently, only four CRDT primitives are implemented: Register, Set, Map, Graph. Potential extensions are to integrate more implementations such as Tree, Array and, List. Moreover, although these primitives aim to ease creation of end-user data types, some knowledge of CRDT are still required. We implemented one example of end-user graph data called SimpleGraph. Another potential extensions are to integrate more common data such as plain text or XML document. Since CollabServer is designed for real-time collaboration, we also plan to make a performance benchmarking such as global speed, number of concurrent operations supported etc. Another possible improvement is to support offline collaboration using another paradigm than our current Last Writer Wins implementations.

BIBLIOGRAPHY

[1] Inria talk about strong eventual consistency and conflict-free replicated data types. https://www.youtube.com/watch?v=ebWVLVhiaiY, 2013.

[2] Subversion book. https://tortoisesvn.net/docs/release/TortoiseSVN_en/tsvn-basics-versioning.html, 2017.

[3] Kerstin Altmanninger, Martina Seidl, and Manuel Wimmer. A survey on model versioning approaches. *International Journal of Web Information Systems*, 5(3): 271–304, 2009.

[4] Francesco Basciani, Juri Di Rocco, Davide Di Ruscio, Amleto Di Salle, Ludovico Iovino, and Alfonso Pierantonio. MDEForge: an extensible Web-based modeling platform. *CloudMDE*, page 66, 2014.

[5] Tim Baumann. OT explained, 2015.

[6] Philip A. Bernstein and Nathan Goodman. Concurrency control in distributed database systems. *ACM Comput. Surv.*, 13(2):185–221, June 1981. URL http://doi.acm.org/10.1145/356842.356846.

[7] E. Brewer. Cap twelve years later: How the "rules" have changed. *Computer*, 45 (2):23–29, Feb 2012.

[8] Eric A. Brewer. Towards robust distributed systems (abstract). In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC ’00, pages 7–, New York, NY, USA, 2000. ACM. ISBN 1-58113-183-6. URL http://doi.acm.org/10.1145/343477.343502.

[9] Eric A. Brewer. Towards robust distributed systems. https://awoc.wolski.fi/dlib/big-data/Brewer_podc_keynote_2000.pdf, 2000.

[10] Mike Chapple. Abandoning acid in favor of base in database engineering. `https://www.lifewire.com/abandoning-acid-in-favor-of-base-1019674`, 2018.

[11] Yuan Cheng, Fazhi He, Shuxu Jing, and Zhiyong Huang. An multiuser undo/redo method for replicated collaborative modeling systems. *Computer Supported Cooperative Work in Design, 2009. CSCWD 2009 13th International Conference on Computer Supported Cooperative Work in Design*, 2009.

[12] Eugene Syriani Constantin Masson, Jonathan Corley. Collaborative modeling environment features. `https://www.remodd.org/v1/content/collaborative-modeling-environment-features`, 2017.

[13] Jonathan Corley, Eugene Syriani, and Hüseyin Ergin. Evaluating the cloud architecture of atompm. In *MODELSWARD*, pages 339–346, 2016.

[14] Jonathan Corley, Eugene Syriani, Huseyin Ergin, and Simon Van Mierlo. Cloud-based Multi-View Modeling Environments. In A M Cruz and Paiva S, editors, *Modern Software Engineering Methodologies for Mobile and Cloud Environments*. IGI Global, 2016.

[15] Davide Di Ruscio. Collaborative model driven software engineering: a Systematic Mapping Study. *COMMitMDE at MoDELS 2016*, 2016.

[16] Michel Dirix. blog.genmymodel.com/discover-the-revision-history-in-your-genmymodel-projects.html, 2017.

[17] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. *SIGMOD Rec.*, 18(2):399–407, June 1989. URL `http://doi.acm.org/10.1145/66926.66963`.

[18] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, SIGMOD ’89, pages 399–407, New York, NY, USA, 1989. ACM. ISBN 0-89791-317-5. URL `http://doi.acm.org/10.1145/67544.66963`.

[19] Neil Fraser. Diff strategies. https://neil.fraser.name/writing/diff/, 2006.

[20] Neil Fraser. Fuzzy patch algorithm. https://neil.fraser.name/writing/patch/, 2006.

[21] Neil Fraser. Differential synchronization. In *Proceedings of the 9th ACM Symposium on Document Engineering*, DocEng '09, pages 13–20, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-575-8. URL http://doi.acm.org/10.1145/1600193.1600198.

[22] Neil Fraser. Differential synchronization google tech talks. https://www.youtube.com/watch?v=S2Hp_1jqpY8, January 2009.

[23] Josh Fruhlinger. Why google wave failed: Too complicated, no fun. https://www.infoworld.com/article/2625908/saas/why-google-wave-failed--too-complicated--no-fun.html, 2010.

[24] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, and Grady Booch. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994. ISBN 0-201-63361-2.

[25] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. ISBN 0-201-63361-2.

[26] Google. Google realtime api. https://developers.google.com/google-apps/realtime/overview, 2017.

[27] Guido Van Rossum. How the Dropbox Datastore API Handles Conficts. https://blogs.dropbox.com/developers/2013/08/how-the-dropbox-datastore-api-handles-conflicts-part-two-resolvin
2013.

[28] Irisate. Real time collaboration technology roundup. `https://irisate.com/collaborative-editing-solutions-round-up/`, 2017.

[29] Kevin Jahns. Yjs framework website. `http://y-js.org/`, 2014.

[30] S Kelly. Collaborative modelling with version control. *Seidl M., Zschaler S. (eds) Software Technologies: Applications and Foundations. STAF 2017*, 2018.

[31] Steven Kelly. Smart model versioning. `https://modeling-languages.com/smart-model-versioning/`, 2017.

[32] Steven Kelly, Kalle Lyytinen, and Matti Rossi. MetaEdit+ A fully configurable multi-user and multi-tool CASE and CAME environment. In *Conference on Advanced Information Systems Engineering*, volume 1080 of *LNCS*, pages 1–21. Springer, 1996.

[33] Haim Kilov. From semantic to object-oriented data modeling. In *First International Conference on System Integration*, pages 385–393, 1990.

[34] Martin Kleppmann. A critique of the cap theorem. *eprint arXiv:1509.05393*, 2015.

[35] Dimitrios S. Kolovos, Louis M. Rose, Nicholas Matragkas, Richard F. Paige, Esther Guerra, Jesús Sánchez Cuadrado, Juan De Lara, István Ráth, Dániel Varró, Massimo Tisi, and Jordi Cabot. A research roadmap towards achieving scalability in model driven engineering. In *Proceedings of the Workshop on Scalability in Model Driven Engineering*, BigMDE '13, pages 2:1–2:10, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2165-5. URL `http://doi.acm.org/10.1145/2487766.2487768`.

[36] Leslie Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM 21, 7 (July 1978), 558-565. Reprinted in several collections, including Distributed Computing: Concepts and Implementations, McEntire et al., ed. IEEE Press, 1984.*, pages 558–565, July 1978. URL `https://www.microsoft.com/en-us/research/publication/time-clocks-ordering-events-distributed-system/`.

[37] Leslie Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):51–58, 2001.

[38] Graig Larman. *Applying UML and patterns: an introduction to object oriented analysis and design and interative development*. Pearson Education India, 2012. ISBN 0-131-48906-2.

[39] Mihai Letia, Nuno Preguiça, and Marc Shapiro. Consistency without concurrency control in large, dynamic systems. *ACM SIGOPS operating Systems Review*, 2010.

[40] Miklós Maróti, Tamás Kecskés, Róbert Kereskényi, Brian Broll, Péter Völgyesi, László Jurácz, Tihamér Levendovszky, and Ákos Lédeczi. Next Generation (Meta)Modeling: Web- and Cloud-based Collaborative Tool Infrastructure. In *Multi-Paradigm Modeling*, volume 1237, pages 41–60. CEUR-WS.org, oct 2014.

[41] M Maróti, Tamas Kecskes, R Kereskényi, Brian Broll, Péter Völgyesi, L Jurácz, T Levendoszky, and Akos Ledeczi. Next generation (meta)modeling: Web- and cloud-based collaborative tool infrastructure. *CEUR Workshop Proceedings*, 1237: 41–60, 01 2014.

[42] C Masson, J Corley, and E Syriani. Feature model for collaborative modeling environments. *MODELS 2017 Satellite Events - Workshop on Collaborative Modelling in MDE*, 2017.

[43] T. Mens. A state-of-the-art survey on software merging. *IEEE Transactions on Software Engineering*, 28(5):449–462, May 2002.

[44] Petru Nicolaescu, Kevin Jahns, Michael Derntl, and Ralf Klamma. Near real-time perr-to-peer shared editing on extensible data types. *Group 16 Proceedings of the 19th International Conference on Supporting Group Work*, 2016.

[45] OBEO. Collaborative features. `https://www.obeodesigner.com/en/collaborative-features`, 2017.

[46] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. 2014.

[47] Karin Petersen, Mike J SPreitzer, Douglas B Terry, Marvin M Theimer, and Alan J Demers. Flexible update propagation for weakly consistent replication. *SOSP '97 Proceedings of the sixteenth ACM symposium on Operating systems principles*, October 1997.

[48] Philip Langer. Version control for models: From Research to Industry and Back Again. In *Workshop on Models and Evolution*, volume 1706, page 1. CEUR-WS.org, 2016.

[49] Rachel Potvin and Josh Levenberg. Why google stores billions of lines of code in a single repository. *Communications of the ACM*, 2016.

[50] Nuno Preguiça, Joan Manuel Marques, Marc Shapiro, and Mihai Letia. A commutative replicated data type for cooperative editing. *IEEE International Conference on Distributed Computing Systems*, 2009.

[51] Martin Robert C. and Martin Micah. *Agile Principles, Patterns, and Practices in C#*. Prentice Hall, 2006. ISBN 0-131-85725-8.

[52] Yasushi Saito and Marc Shapiro. Optimistic replication. *ACM Computing Surveys*, March 2005.

[53] Douglas C. Schmidt. Model-Driven Engineering. *IEEE Computer*, 39(2):25–31, 2006.

[54] Gilbert Seth and Lynch Nancy. Brewer's conjecture and the feasibility of consistent, available, partition tolerant web services. *SIGACT News*, 2002.

[55] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In *Stabilization, Safety, and Security of Distributed Systems*. Springer, Berlin, Heidelberg, 2011.

[56] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Crdt github repository inria (delta-enabled-crdts). `https://github.com/CBaquero/delta-enabled-crdts`, 2016.

[57] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirsk. A comprehensive study of convergent and commutative replicated data types. Technical report, Inria, Centre Paris-Rocquencourt, 2011.

[58] Alexander Shvets, Gerhard Frey, and Marina Pavlova. Observer pattern. https://sourcemaking.com/design_patterns/observer, 2018.

[59] Alexander Shvets, Gerhard Frey, and Marina Pavlova. Observer pattern. https://sourcemaking.com/design_patterns/visitor, 2018.

[60] Daniel Spiewak. Understanding and applying operational transformation. https://www.codecommit.com/blog/java/understanding-and-applying-operational-transformation, 2010.

[61] David Sun and Chengzheng Sun. Operation context and context-based operational transformation. In *Proceedings of the 2006 20th Anniversary Conference on Computer Supported Cooperative Work*, CSCW '06, pages 279–288, New York, NY, USA, 2006. ACM. ISBN 1-59593-249-6. URL http://doi.acm.org/10.1145/1180875.1180918.

[62] E. Syriani. A multi-paradigm foundation for model transformation language engineering. doctoral dissertation, mcgill university. https://www-ens.iro.umontreal.ca/~syriani/files/dissertation.pdf, 2011.

[63] Gabriele Taentzer, Claudia Ermel, Philip Langer, and Manuel Wimmer. A fundamental approach to model versioning based on graph modifications: from theory to implementation. *Software & Systems Modeling*, 13(1):239–272, 2014.

[64] Simon Van Mierlo, Bruno Barroca, Hans Vangheluwe, Eugene Syriani, and Thomas Kühne. Multi-level modelling in the modelverse. *17th ACM/IEEE International Conference MODELS*, 2014.

[65] Yentl Van Tendeloo. Modelverse documentation. https://msdl.uantwerpen.be/documentation/modelverse/, 2018.

[66] Werner Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, January 2009. URL `http://doi.acm.org/10.1145/1435417.1435432`.

[67] Dustin Wüest, Norbert Seyff, and Martin Glinz. FlexiSketch: A Mobile Sketching Tool for Software Modeling. In *Mobile Computing, Applications, and Services*, Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, pages 225–244. Springer, Berlin, Heidelberg, 2012.

Credits for the icons we used from flaticon.com:

- Simple Icon (`https://www.flaticon.com/authors/simpleicon`)
- Gregor Cresnar (`https://www.flaticon.com/authors/gregor-cresnar`)
- Smash Icons (`https://www.flaticon.com/authors/smashicons`)
- Icon Works (`https://www.flaticon.com/authors/icon-works`)
- Google (`https://www.flaticon.com/authors/google`)
- Pixel Buddha (`https://www.flaticon.com/authors/pixel-buddha`)
- Freepik (`https://www.flaticon.com/authors/freepik`)
- Dave Gandy (`https://www.flaticon.com/authors/dave-gandy`)