Universidade do Estado do Rio de Janeiro

Centro de Tecnologia e Ciências

Faculdade de Engenharia

Fabio de Almeida Daltro Bosisio

Permissionless Peer-to-Peer JSON Datasets

Rio de Janeiro

2023

Fabio de Almeida Daltro Bosisio

Permissionless Peer-to-Peer JSON Datasets

Dissertação apresentada, como requisito parcial para obtenção do título de Mestre em Ciências, ao Programa de Pós-Graduação em Engenharia Eletrônica, da Universidade do Estado do Rio de Janeiro. Área de concentração: Redes de Telecomunicações.

Orientador: Prof. Dr. Francisco Sant'Anna

Rio de Janeiro

2023

CATALOGAÇÃO NA FONTE

UERJ / REDE SIRIUS / BIBLIOTECA CTC/B

B743 Bosisio, Fabio de Almeida Daltro.

Permissionless peer-to-peer JSON datasets / Fabio de Almeida Daltro Bosisio. – 2023.

121 f.

Orientador: Francisco Sant'Anna.

Dissertação (Mestrado) – Universidade do Estado do Rio de Janeiro, Faculdade de Engenharia.

1. Engenharia eletrônica - Teses. 2. Compartilhamento de arquivos de computador - Teses. 3. JavaScript (Linguagem de programação de computador) - Teses. 4. Banco de dados distribuído - Teses. I. Sant'Anna, Francisco. II. Universidade do Estado do Rio de Janeiro, Faculdade de Engenharia. III. Título.

CDU 004.65

Bibliotecária: Júlia Vieira – CRB7/6022

Fabio de Almeida Daltro Bosisio

Permissionless Peer-to-Peer JSON Datasets

Dissertação apresentada, como requisito parcial para obtenção do título de Mestre em Ciências, ao Programa de Pós-Graduação em Engenharia Eletrônica, da Universidade do Estado do Rio de Janeiro. Área de concentração: Redes de Computadores e Sistemas Distribuídos.

Aprovado em: 01 de Junho de 2023

Rio de Janeiro

2023

AGRADECIMENTO

Agradeço primeiramente a Deus por estarmos vivos e bem, pela nossa saúde e por todas as oportunidades e graças concedidas incessantemente ao longo da minha vida.

Agradeço ao apoio incondicional da minha esposa Cristiane, e das minhas filhas Giovanna e Gabriella, que suportaram os diversos períodos de ausência, abrindo mão em diversos momentos, sempre me motivando e incentivando com amor e esperaça todos os dias.

Agradeço a minha mãe Marlene pela vida, pelos sacrifícios, pelo apoio, pelo incentivo e pelo amor incondicional. Agradeço a meu pai Ney *(in memoriam)* que apesar de não estar mais presente entre nós, sinto seu apoio e incentivo a todo instante, principalmente nos momentos mais difíceis. Agradeço ao meu irmão Rafael por ser minha referência nos estudos, e pelo incentivo em sempre seguir em frente.

Agradeço ao meu orientador, professor Francisco Sant'Anna pelos ensinamentos, pela disponibilidade de tempo, pela paciência, por ser incansável em fazer com que eu após tão longo período afastado do meio acadêmico, conseguisse novamente seguir a trilha do conhecimento, da pesquisa e da produção científica.

Agradeço aos professores Alexandre Sztajnberg e Marcelo Rubinstein pela confiança em me aceitarem no ingresso desse mestrado e por todos os ensinamentos e incentivos recebidos durante o curso.

Agradeço aos professores Adriano Caminha e Vera Caminha pela carta de recomendação após tanto tempo de formado, pela confiança e incentivo.

Agradeço ao meu amigo Tiago Maessi, por ter me tirado da inércia acadêmica, me incentivando a ingressar no mestrado e retornar aos estudos.

Agradeço ao PEL, a UERJ por viabilizar e pelos recursos investidos nesta pesquisa.

Estar decidido, acima de qualquer coisa, é o segredo do êxito

Henry Ford

RESUMO

BOSISIO, Fabio de Almeida Daltro. *Permissionless Peer-to-Peer JSON Datasets.* 121 f. Dissertação (Mestrado em Engenharia Eletrônica) - Faculdade de Engenharia, Universidade do Estado do Rio de Janeiro (UERJ), Rio de Janeiro, 2023.

Aplicações colaborativas em rede, como Google Docs e Github, permitem que usuários remotos compartilhem projetos enquanto trabalham juntos simultaneamente. Essas aplicações dependem de conjuntos de dados (datasets) distribuídos que representam documentos, códigos e outros dados de aplicações, que os usuários esperam ver e compartilhar de maneira consistente. Atualmente, os sistemas colaborativos mais práticados dependem de servidores centrais para garantir que os datasets permaneçam consistentes em toda a rede. No entanto, autoridades centralizadas detém muito poder, pois controlam a propriedade dos dados e a disponibilidade dos serviços. Neste trabalho, propomos um sistema peer-to-peer (P2P) não permissionado para manipular datasets JSON descentralizados, no qual todos os usuários participam de um mecanismo de consenso para preservar a consistência e corretude dos dados. Nossa principal contribuição é conciliar o Automerge, um tipo de dados replicado livre de conflitos (CRDT) baseado em JSON com o Freechains, um protocolo P2P não permissionado que fornece um mecanismo de reputação que modera o conteúdo e fornece consenso na rede. Como prova de conceito, prototipamos uma Wikipedia não permissionada, na qual os artigos são estruturados como arquivos Automerge JSONs armazenados no Freechains. Cada artigo é uma cadeia separada que armazena uma lista de modificações no seu respectivo JSON. Essa lista é percorrida desde o início em ordem de consenso para recriar o artigo como um JSON completo. Nesse sentido, nosso protótipo se assemelha a um sistema de controle de versão distribuído (DVCS), mas com um mecanismo de consenso que mescla edições concorrentes automaticamente.

Palavras-chave: Aplicações colaborativas. Sistemas não permissionados. Protocolos peer to peer. JSON datasets.

ABSTRACT

BOSISIO, Fabio de Almeida Daltro. *Permissionless Peer-to-Peer JSON Datasets.* 121 f. Dissertação (Mestrado em Engenharia Eletrônica) - Faculdade de Engenharia, Universidade do Estado do Rio de Janeiro (UERJ), Rio de Janeiro, 2023.

Networked collaborative applications, such as Google Docs and Github, allow remote users to share projects while working together concurrently. These applications rely on distributed datasets that represent documents, code, and other application data, which users expect to see and share in a consistent way. Currently, most practical collaborative systems rely on central servers to ensure that datasets remain consistent across the network. However, centralized authorities concentrate too much power, since they control data ownership and service availability. In this work, we propose a permissionless peer-to-peer (P2P) system to manipulate decentralized JSON datasets, in which all users participate in a consensus mechanism to preserve data consistency and correctness. Our main contribution is to reconcile Automerge, a JSON-based conflict-free replicated data type (CRDT) with Freechains, a permissionless P2P protocol that provides a reputation mechanism that moderates content and delivers network consensus. As a proof of concept, we prototyped a permissionless Wikipedia, in which articles are structured as Automerge JSONs files stored on Freechains. Each article is a separate chain that stores a list of modifications to its respective JSON. This list is traversed from the beginning in consensus order to recreate the article as a complete JSON. In this sense, our prototype resembles a distributed version control system (DVCS), but with a consensus mechanism that merges concurrent editions automatically.

Keywords: collaborative applications. permissionless systems. CRDT. Peer to peer protocols. JSON datasets.

LIST OF FIGURES

Figure 1 - Consistency and its challenges 20
Figure 2 - Replication conflict 25
Figure 3 - CvRDT (converged replicated data type) - State Based 29
Figure 4 - CmRDT (commutative replicated data type) - Operations Based 31
Figure 5 - Wikipedia update operations 35
Figure 6 - Original post 36
Figure 7 - Automerge: Data encoding 37
Figure 8 - Automerge: Concurrent text edition 39
Figure 9 - Posts with creation-date element 40
Figure 10 - Automerge: Conflict resolution example 40
Figure 11 - Computation Systems 43
Figure 12 - P2P Pure x P2P Hybrid 45
Figure 13 - Overlay and Underlay Networks 46
Figure 14 - Unstructured P2P Network 47
Figure 15 - Structured P2P Network 49
Figure 16 - Basic concepts of Freechains: chains, blocks, authors and peers. 54
Figure 17 - Ecosystem proposed. 61
Figure 18 - Execution timeline. 69
Figure 19 - Execution timeline using Freechains-JSON. 74
Figure 20 - The branches in the DAG are ordered by reputation. Only the first patch is applied successfully (first write wins) 84

LIST OF TABLES

Table 1 - Comparison between unconverted vs binary converted JSON data files. . . 38

Table 2 - The 3 types of Freechains chains and arrays 53

LIST OF ACRONYMS

P2P	Peer to Peer
CRDT	Conflict-free replicated data type
CmRDT	Commutative replicated data types (Operation-based CRDTs)
CvRDT	Convergent replicated data types (State-based CRDTs)
DAG	Directed acyclic graph
JSON	JavaScript Object Notation
RTCE	Real-time collaborative editing platforms
VCS	Version control system
CVCS	Centralized Version Control System
DVCS	Distributed Version Control System
SC	Strong Consistency
EC	Eventual Consistency
SEC	Strong Eventual Consistency
PoS	Proof of Stake
PoW	Proof of Work
DPoS	Delegated Proof of Stake

SUMMARY

INTRODUCTION 12

1 **THEORETICAL FOUNDATIONS** 15

1.1 Collaborative Applications 15

1.1.1 Types of Collaborative Applications 15

1.1.1.1 RTCE: Real-Time Collaborative Editing Platforms 15

1.1.1.2 VCSs: Version Control Systems [1] 16

1.1.2 Wikipedia 18

1.2 Consistency models 19

1.2.1 Types of Consistency Models 21

1.2.2 Intent-Preserving Consistency 25

1.3 CRDTs: Conflict-Free Replicated Data Types 26

1.3.1 Decentralized Conflict Resolution 27

1.3.2 CRDTs Approaches: CvRDTs & CmRDTs 28

1.3.3 CmRDTs and CvRDTs: Pros and Cons 32

1.3.4 Automerge: a JSON CmRDT 34

1.4 Peer-to-Peer Systems 43

1.4.1 Overlay Networks 46

1.4.2 Unstructured vs. Structured P2P systems 47

1.4.3 Consensus in P2P Protocols 50

1.4.4 Freechains: Permissionless Reputation-Based Consensus 52

1.4.4.1 Internal Operations 53

1.4.4.2 Reputation System 55

2 **PEER-TO-PEER DATASET ECOSYSTEM** 59

2.1 Ecosystem of the Proposed Solution 60

2.2 AMRW: Automerge-JSON Command-Line Editor 62

2.3 AMDIFF: Automerge-JSON Command-Line Diff 66

2.4 AMPATCH: Automerge-JSON Command-Line Patch 67

2.5 Timeline Simulation 68

2.6 Freechains-JSON: Automerge-JSON Synchronization for Freechains 73

3 **PROOF OF CONCEPT: A PERMISSIONLESS WIKIPEDIA** 78

4 **RELATED WORK** 88

4.1 Decentralized Network Protocols 88

4.1.1 Federated Protocols 88

4.1.2 Peer-to-Peer Systems 91

4.2 Decentralized Wikis 94

4.3 CRDT-Based Collaborative Systems and Applications 97

CONCLUSION 101

REFERENCES 103

APPENDIX 103

INTRODUCTION

Networked collaborative applications allow remote users to share projects while working together concurrently [2]. Examples of collaborative applications include Google Docs [3], Github [4], and also Wikipedia [5]. These applications rely on distributed datasets that are read, written, stored, and synchronized through the network. By datasets, we mean any kind of data structures that can represent documents, code, or social media feeds, as found in collaborative applications. Users expect to see and share these datasets in a consistent way, even if their machines synchronize sporadically and at different times. In this work, we materialize the concept of datasets as generic JSON files, given that they are human readable and widely adopted as an interchange format.

Currently, most practical collaborative systems rely on a central server that acts as an authority to ensure that applications datasets remain consistent across the network. However, centralized authorities concentrate too much power, since they control data ownership and service availability [6]. As an emerging alternative, peer-to-peer (P2P) systems [7] aim to decentralize control, such that applications grow organically with new users [8], who contribute with storage, availability, and also, in our context, with moderation of the datasets.

However, P2P systems strive to ensure data consistency and correctness, particularly in the presence of malicious users or Sybils [9]. For instance, malicious users may abuse the system by vandalizing datasets with SPAM or hate speech. By consistency, we mean that all replicas reach the same state, while by correctness, we mean that such consistent state must also be immune to vandalism (integrity) and preserve the users intent (accuracy) [10]. In our context of human-centered collaborative systems, integrity and accuracy inevitably depend on the subjective judgement of users, which we refer as *subjective correctness*.

As a partial solution to preserve consistency and correctness, permissioned P2P systems opt for a centralized membership authority, which grants permission for new users to join the network. Examples of permissioned P2P networks include Ripple [11] and Hyperledger Fabric [12]. Nevertheless, one of our main goals is to preserve the desired organic growth of traditional P2P systems, such that no centralized authority is required.

In this work, we propose a permissionless P2P system to manipulate decentralized

JSON datasets, in which participation is not controlled by a central authority. All users can validate the datasets and participate in a consensus mechanism to preserve data consistency and correctness.

In a permissionless context, a major challenge is to support concurrent data edition and moderation, such that users share consistent and correct datasets even in the presence of Sybils. Bitcoin [13] is the first permissionless protocol to resist Sybils through consensus. The protocol is Sybil resistant because it is expensive to write to its unique timeline (either via proof-of-work or transaction fees). However, Bitcoin and cryptocurrencies in general are not suitable to store generic datasets found in typical collaborative applications, mainly because they impose economic costs to use the protocol. As an alternative, Freechains [14] is a permissionless Sybil-resistant P2P protocol that targets generic collaborative applications. Its main contribution is a reputation mechanism that moderates content through likes and dislikes and, at the same time, delivers network consensus. However, Freechains does not support structured datasets with attached semantics, since it represents content as a simple linked list of binary blobs stored in individual "chains", which each application must interpret appropriately.

A complementary approach to support concurrent data editions is to rely on conflict-free replicated data types (CRDTs) [15]. The main advantage of CRDTs is that they can represent typical data structures with attached semantics, such as arrays and dictionaries. As an example, Automerge [16] provides JSON data structures as CRDTs, such that they can be read and written concurrently without conflicts. Nevertheless, CRDTs are still subject to corner cases, such as simultaneous editions of the very same part of documents. In addition, CRDTs are not themselves immune to malicious users and Sybils, which might vandalize datasets with no restrictions. In such cases, a consensus mechanism might still be necessary to preserve data integrity and accuracy.

Our main contribution is to reconcile CRDTs and permissionless P2P systems, such that decentralized collaborative applications support consistent and correct structured datasets, even in the presence of Sybils. More specifically, we propose to reconcile Automerge and Freechains, extending JSON CRDTs with a consensus mechanism that can handle consistency corner cases and also moderate malicious editions: Automerge editions are ordered by the Freechains consensus mechanism providing a priority to solve conflicts, thus preserving accuracy. In addition, further moderation can revert priorities or even remove abusive or malicious editions, thus preserving integrity.

As a proof of concept, we prototyped a permissionless Wikipedia [5], in which articles are structured as Automerge JSONs files stored on Freechains. Each article is a separate chain that stores a list of modifications to the JSON. This list is traversed from the beginning in consensus order to recreate the article as a complete JSON. In this sense, our prototype resembles a distributed version control system (DVCS) [17], but with a consensus mechanism that merges concurrent editions automatically. For reading and editing articles, we developed specialized versions of *diff* and *patch* tools on top of Automerge, which we refer as *AMDIFF* and *AMPATCH*. *AMDIFF* obtains the difference between a new edition and the current version on the chain, such that it can be written back to the chain. *AMPATCH* is used to restore the complete JSON from individual modifications on the chain. Since patches are applied in consensus order, editions with higher priority are merged first, with further conflicting editions being rejected automatically. Finally, as mentioned above, further moderation can revert priorities or even remove abusive or malicious editions.

The rest of this dissertation is organized as follows: Chapter 1 presents the theoretical foundations of our work: collaborative applications, consistency models, CRDTs, and P2P protocols. Chapter 2 describes our proposed ecosystem for permissionless JSON datasets based on Automerge and Freechains. Chapter 3 prototypes a permissionless Wikipedia based on our proposed ecosystem. Chapter 4 discusses related work on decentralized content sharing. We close this dissertation with final considerations and conclusions about this work.

1 THEORETICAL FOUNDATIONS

This chapter presents the theoretical foundations supporting this work, including collaborative applications, consistency models, conflict-free replicated data types (CRDTs) [18], and peer-to-peer protocols [8].

1.1 Collaborative Applications

"Collaborative applications are designed to help people involved in a common task to achieve their goals. One of the earliest definitions of collaborative software is intentional group processes plus software to support them" [19].

1.1.1 Types of Collaborative Applications

"In terms of the level of interactions it provides, collaborative software may be divided into Real-time collaborative editing platforms (RTCE) and Version control platforms (VCS)" [20].

The fundamental difference between RTCE and VCS is the mode of communication and collaboration they provide. RTCE enables real-time synchronous communication and collaboration, while VCS provides asynchronous collaboration through the tracking and management of changes.

1.1.1.1 RTCE: Real-Time Collaborative Editing Platforms

RTCE involves synchronous communication, which means that participants interact with each other in real-time. "This allows multiple users to engage in live, simultaneous and reversible editing of the same digital document, computer file or cloud-stored data, with automatic and nearly instantaneous merging of their edits" [20]. Examples of RTCEs are online spreadsheets and word processing, such as Google Docs [3] and Collabora Online [21]. RTCEs automatically synchronize changes made by all online users to a document in real-time, periodically and instantly, as they edit it on their respective devices.

The challenge for real-time collaborative editing solutions is maintaining synchronization in an environment subject to communication latency.

Communication latency refers to the delay that occurs between sending and receiving data on a network. This delay can be caused by various factors such as physical distance between devices, network congestion, and connection quality.

When there is network latency, the speed of communication is affected because data transmission is delayed. For example, when a user sends a command to a remote system, the response may take a few seconds to arrive back to the user due to network latency. This can result in noticeable delays in system responsiveness and efficiency.

In the context of real-time collaboration, network latency is particularly important as it can impact users' ability to edit and view changes made to the document in real-time. If latency is too high, there can be significant delays in synchronizing changes made by multiple users, which can lead to conflicts and inconsistencies in the collaborative document.

As we discuss next, version control alternatives can handle these asynchronous and concurrent edits without relying on permanent and low-latency network connections.

1.1.1.2 VCSs: Version Control Systems [1]

"VCSs [1] are used in asynchronous collaboration environments, allowing users to make concurrent edits to a file, while preserving every individual edition by every user in multiple files that are small variants of the original file" [20].

In a VCS, each user can make edits to the file in their own local copy, which is essentially a separate branch of the original file. When the user is ready to save their changes, they can commit their changes to the VCS. The VCS then creates a new version of the file that incorporates the user's changes, while preserving the original version of the file. Other users can then pull the latest version of the file from the VCS, which includes the changes made by the original user. They can then make their own edits to the file in their local copy, and repeat the commit process to save their changes as a new version of the file.

By using a VCS, each user can work on their own branch of the file independently, without worrying about conflicts or overwriting changes made by other users. The VCS manages the merging of changes between different branches, ensuring that every saved edit is preserved as a separate version of the file.

There are two distinct categories of VCSs: centralized and distributed.

- **Centralized Version Control Systems (CVCS) [22]** contain just one global repository, which every user needs to commit to reflect changes.

  Two steps are required to make a user changes visible to others:

  - The user commit to the central repository.
  - The other users update from the central repository.

  In general, the most used VCSs are centralized, such as Apache Subversion [23] and Perforce Helix Core Version Control [24].

  The biggest disadvantage of CVCS is having a single point of failure within the centralized server. If the remote server is down, then no one can work on the file or commit changes. This condition is avoided if we go for a distributed approach.

- **Distributed Version Control Systems (DVCS) [22]** makes it so that each user has a complete local copy of the repository so they can commit, branch, and merge locally. The server does not need to store the complete files, only the differences between each commit performed. "Each user has his own replica of the repository, and while working and committing changes do not give other users access to them. This is because a commit will reflect changes only on his local replica of the repository and he needs to push the changes in order to make them visible to the other users' replicas. Similarly, when the user does updates, he does not get other users' changes, unless he first pulls those changes into his replica of repository" [17].

  Four steps are required to make a user changes visible to others:

  - The user commits changes locally
  - The user pushes the changes to a shared remote repository
  - Other users pull the changes from the shared remote repository
  - Other users update their local working copies with the changes

  Examples of DVCS are distributed source code management systems such as Git [25], Mercurial [26] and Bazaar that copy the repository as well as its history as a local copy on your users.

In summary, collaborative applications can be built on top of distributed systems to enable collaboration between users who may be geographically dispersed. By leveraging the power of distributed systems, these applications can provide high levels of reliability, scalability, and performance, while enabling real-time collaboration between users.

Asynchronous collaborative editing can happen by any user at any time, and even at the same time, so more than one user can simultaneously edit the same item, the same line, the same word causing editing conflicts. To maintain consistency, it is necessary for users to choose which edition should prevail, or it forces the system to perform merges or arbitrary choices on editions of several users without prior notice to them.

Our work relies on decentralized consensus to decide over simultaneous edits through a system of user reputations. This environment makes it possible to have collaborative applications based on distributed version control that also has an interesting solution for the network to deal with concurrent edition conflicts, based on the reputation of the users involved.

Next we discuss an example of a CVCS collaborative application, the Wikipedia [5], which we use as a proof of concept, changing its profile from CVCS to DVCS, in a permissionless P2P environment, through the proposal of this work.

1.1.2 Wikipedia

Wikipedia [5] is one example of a CVCS collaborative application in action. "It is a multilingual free online encyclopedia written and maintained by a community of volunteers through open collaboration and a wiki-based editing system" [27]. The site relies on volunteers from around the world to write, edit, and fact-check articles. Wikipedia's collaborative approach results in a vast, comprehensive, and ever-evolving encyclopedia. But as it is a platform open to the public with minimal controls in place, it becomes easier for individuals to submit unauthorized or inappropriate content, sometimes vandalism, which can negatively impact the quality and accuracy of the information available.

Wikipedia is a centralized system to which users send requests and one of its servers responds with articles. Let's imagine a use case, where a Wikipedia user types in the search term “First World War”, at that moment he is a client. This searched term will be forwarded to Wikipedia's servers, which will return articles based on the term. In other words, it is a client-server relationship.

Moderation on Wikipedia

Wikipedia is a a permissioned system, since the site is managed and moderated by experienced and trusted editors, called administrators who are committed to maintaining the correctness and quality of its content.

At first, all Wikipedia users are/have the capacity to act as moderators. "They can flag articles for possible deletion, they may file and participate in reports for misconduct and misuse of editing tools, and in some cases they may close discussions and deletion debates. However, only administrators, who are trusted and experienced editors of the site, may physically delete entire articles, protect articles from improper editing and disruption, and block users (a physical block implemented by the site software). Occasionally some contributors and/or commentators are banned (forbidden to post on certain pages or topics), and some may be blocked. Editors are banned as an outcome of the dispute resolution process, violation of a ban may result in a block. Obvious and/or serious cases of violation of policy such as disruptive editing, copyright infringement, vandalism, libel, etc, may be unilaterally blocked by an administrator" [28].

In summary, Wikipedia [5] was built based on the wiki system, using rich structures, and it is a permissioned environment, using owned administrators to guarantee the integrity of the posted content avoiding vandalism and SPAM. However, we want the network growth and the content control not to be controlled by predefined entities. We are proposing a permissionless collaborative system, where there are no centralized administration granting users to participate or moderate content. Our goal is to use a pemissionless reputation system in which users spend their reputation to moderate published content in a decentralized way, both from the point of view of integrity, SPAM, abusive behavior and from the point of view of accuracy, keeping the user intention on possible conflicts of simultaneous editions.

1.2 Consistency models

"In real-time collaborative editing applications, users create a shared document by issuing insert, delete, and undo operations on their local replica anytime and anywhere" [29]. "In a single computer, we can guarantee that the system stores the most recently updated data, however, in a distributed system, data is shared and replicated across many computing nodes" [29]. Therefore, concurrent editing conflicts cause data consistency problems.

A **consistency model** is a "contract between a distributed system and the applications that run on it, and is composed of a set of guarantees made by the distributed system" [30]. Under these guarantees, distributed systems can implement various consistency models. Like for example strong consistency, eventual consistency and strong eventual consistency.

By **consistency** we mean the "property of the distributed system which says that every node/replica has the same view of data at a given point in time irrespective of whichever replica has updated the data" [31]. "For instance, if we perform a read operation on a consistent system, all the nodes return the value of the most recent write operation" [31].

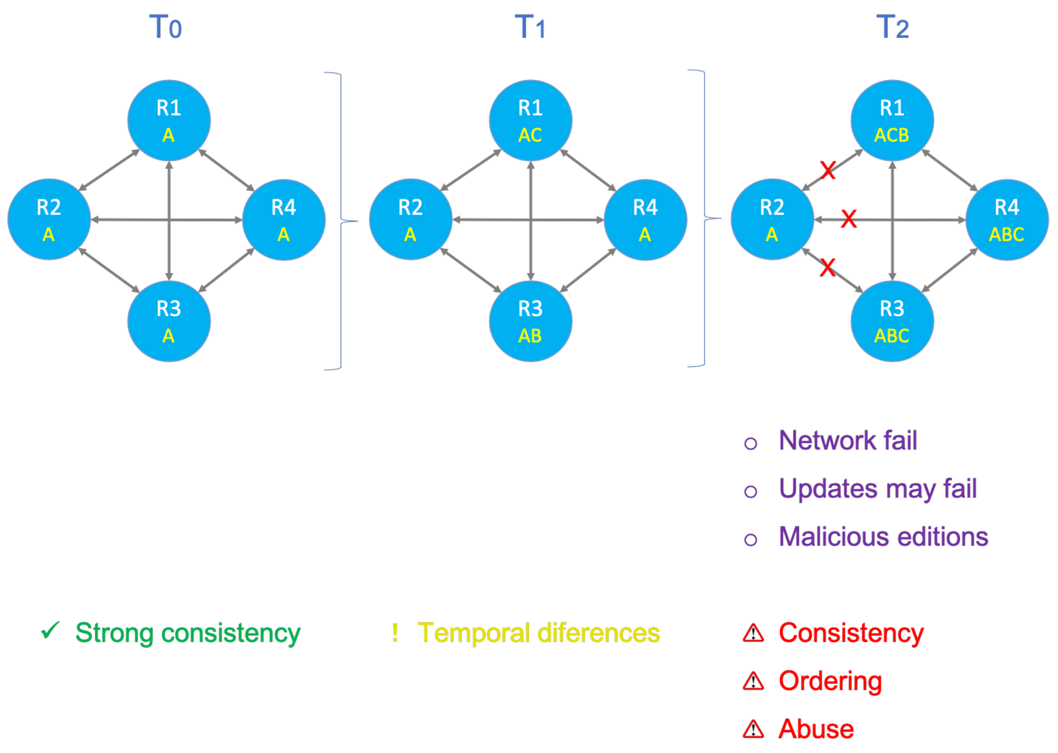


Figure 1 - Consistency and its challenges

Figure 1 illustrates the challenges of consistency in decentralized collaborative editions:

- **T0 - Initial**: At first, the 4 replicas are in synchronicity, that is, they have the same version of the content (A).

- **T1 - Local edition**: Then, two replicas, R1 and R3, edit their local copies (AC and AB), leading to differences in the states of each distributed replica. Therefore, the system is not consistent at this moment.
- **T2 - Synchronization**: Then, we expect that after the replicas synchronize, they all reach a unique consistent state, which is either the values ABC or the values ACB on all replicas. However there are some problems that can occur in this process:
  - **Network or system failures**, in which some replicas may never receive updates (R2).
  - **Ordering failures**, in which messages may be received in arbitrary orders (R1 vs R4).
  - **Malicious edition failures**, in which users make a malicious editions that are propagated through the network.

1.2.1 Types of Consistency Models

Strong consistency (SC) "can be understood as linearisability, serialisability, or a combination of the two (one-copy serialisability). Informally, the goal of strong consistency is to make a system behave like a single sequentially executing node, even when it is replicated and concurrent" [30]. In the example of Figure 1, strong consistency requires that all replicas have one of the options, ABC or ACB.

"Most systems implement strong consistency by designating a single node as the leader, which arbitrates a total order of operations and prevents concurrent access from causing conflicts" [30]. However, "strong consistency may be unwarranted or unnecessary depending on the application: it may impose a performance degradation on the system, or it may simply be unfeasible to implement, especially in large distributed systems" [30]. "Relying on a single leader or central server limits the use and deployment of these systems: the server may become a bottleneck that limits scalability, and it makes the system vulnerable to disruption by network outages, denial-of-service attacks, censorship, and server failures" [30]. "Nodes must constantly communicate with the leader in order to perform operations, if a node cannot reach the leader due to a network fault, its execution is stalled, and we will have an infinite loading in the request of the node" [30].

Summary of strong consistency:

- All nodes will always return the most current value.
- Replicas always converge.
- There is a total order of operations that prevents concurrent access from causing conflicts.
- There's no replication conflict.

There is a difficulty in maintaining strong consistency. For instance, in our proposal of completely decentralized Wikipedia, in which each peer can make and receive updates, it is not possible to guarantee that each replica receives the same sequence of updates. However decentralized architectures, or peer to peer, that use weaker consistency, such as eventual consistency, can offer greater fault tolerance, scalability and performance.

Eventual consistency (EC) "guarantees that if no new updates are made to the shared state, all nodes will eventually have the same data" [32]. In this model, updates to the data may take some time to propagate across all nodes in the system. The key characteristic of eventual consistency is that, given a sufficiently long period of time without updates, all nodes in the system will eventually converge to the same consistent state. In an eventually consistent system, when a write operation is performed on a node, that node will update its own local copy of the data, but may not immediately propagate that update to all other nodes in the system. As a result, different nodes may have slightly different versions of the data at any given time. However, over time, as updates propagate through the system, all nodes will eventually come to see the same version of the data.

Eventual consistency is often used in large-scale distributed systems, where strong consistency may be difficult or impossible to achieve due to factors such as network latency, node failures, or high update rates. "While eventual consistency can result in temporary inconsistencies in the system, it can be a practical and effective way to ensure that all nodes eventually converge to a consistent state, even in the face of highly dynamic and unpredictable conditions. Since this model allows conflicting updates to be made concurrently, it requires a mechanism for resolving such conflicts. For example, version control systems such as Git require the user to resolve merge conflicts manually, and some NoSQL distributed database systems such as Cassandra adopt a last-writer-wins policy, under which one update is chosen as the winner, and concurrent updates are discarded" [33]. Eventual consistency offers weak guarantees: it does not constrain the system behaviour when updates never cease, or the values that read operations may return prior to convergence.

Summary of eventual consistency:

- If no further updates are made to a given data item, all reads for that item will eventually return the same value.
- There are no guarantees on when replicas will converge.
- Conflicts arising from merges are resolved asynchronously, in the background.
- In case of replication conflict: arbitrate or revert.

Considering that our decentralized Wikipedia uses eventual consistency, there is no determined order of operations, and we may face some problems, such as inconsistencies during updates: Because updates are not immediately propagated to all users, some users may see an outdated version of an article while others see a more recent version. This can lead to confusion and inconsistencies in the information presented.

Strong eventual consistency (SEC) is a "model that strikes a compromise between strong and eventual consistency" [15]. Like eventual consistency, SEC allows for some delay in updates propagating across the system, but like strong consistency, it guarantees that all nodes will eventually see the same data in the same order.

In a system that provides SEC, when a node performs a write operation, it will first update its local copy of the data, and then send a message to all other nodes in the system informing them of the update. When a node receives this message, it will apply the update to its own local copy of the data, and then forward the message to all other nodes in the system. This process continues until all nodes have received and applied the update, ensuring that all nodes see the same data in the same order.

The key difference between SEC and eventual consistency is the level of guarantees provided. Eventual consistency guarantees that all nodes will eventually converge to a consistent state, but makes no guarantees about the order in which updates will be seen. In contrast, SEC guarantees both eventual convergence and strict ordering of updates. This makes SEC a stronger consistency model than eventual consistency.

In addition, SEC also provides liveness, safety, and monotonicity properties.

Liveness in SEC refers to the guarantee that updates will eventually be applied to all replicas, even in the presence of failures or network partitions. In other words, the system will eventually make progress, and all updates will eventually be applied to all replicas.

Safety in SEC refers to the guarantee that all replicas will eventually converge to the same state, regardless of the order in which updates were made. This means that replicas will not diverge, and all users will eventually see the same content.

Monotonicity in SEC refers to the guarantee that once a user observes a particular state of the system, they will never see that state again. In other words, the system's state will always move forward and never regress to a previous state.

Together, these properties ensure that the system is reliable, consistent, and continuously makes progress. Users can trust that their updates will eventually be applied to all replicas, and all users will eventually see the same content. Additionally, users will never see a previous state of the system again, and the system will always move forward, even in the presence of failures and network partitions.

Large-scale deployments of SEC algorithms include datacenter-based applications using Riak [34]. "SEC is also typically achieved using techniques such as conflict-free replicated data types (CRDTs), which are designed to support operations that can be safely applied in any order" [15].

Summary of strong eventual consistency:

- Updates will replicate eventually (**liveness**).
- Any two or more nodes that have received the (unordered) set of updates will be in the same state (**safety**).
- The application will never suffer rollbacks, the order is preserved (**monotonicity**).

In our proposal, we rely on Freechains [14], which is a peer-to-peer content dissemination protocol that provides strong eventual consistency, as the basis of our permissionless environment. We also rely on Conflict-free Replicated Datatypes (CRDTs), which ensure strong eventual consistency [15] to manage concurrent dataset operations between peers. Based on this scenario, we will build our proof of concept, the decentralized and non-permissioned Wikipedia, with ordered converged updates across all nodes, but without the need for a leader or central server.

1.2.2 Intent-Preserving Consistency

"When it is unclear how to reconcile changes to achieve data consistency, we have a replication conflict" [35]. As an example, in a decentralized system without a shared clock, it is virtually impossible to determine which update happens first.

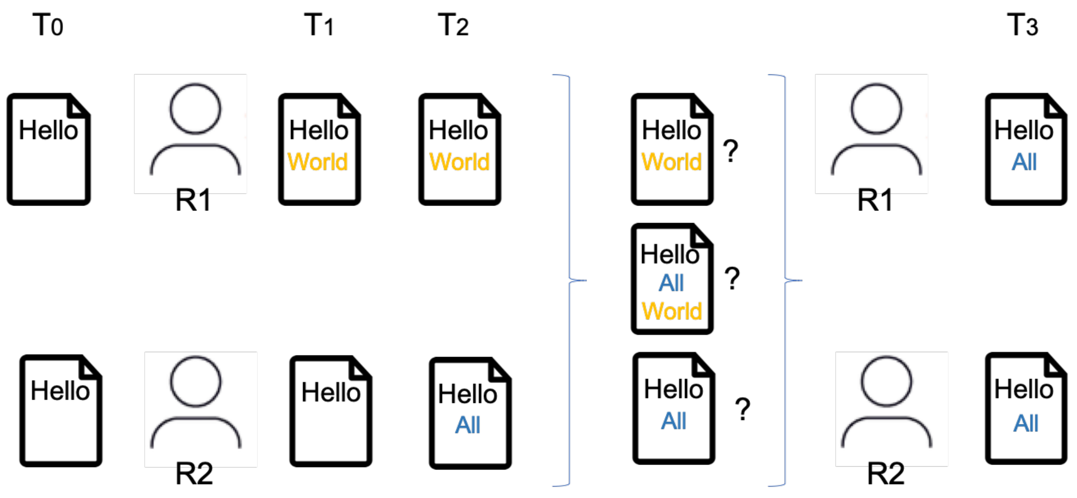


Figure 2 - Replication conflict

Back to our Wikipedia use case, in Figure 2 we see two replicas, R1 and R2 from users who are editing the same portion of a document:

T0: We have a consistent document containing the word "Hello" for all users.

T1: The replica R1 completes the existing text "Hello" with the word "World", replica R1 becomes "Hello World". Replica R2 does not make any changes.

T2: R1 does not make any changes, while R2 completes the original document containing the word "Hello" with the word "All", leaving R2 as "Hello All".

Considering this scenario, what should be the result of T3 when the replicas converge? "Hello World", "Hello All" or a combination of the two, such as "Hello All World"?

The most common solution is to have a single arbitration policy that resolves these conflicts. As an example, "last write wins" uses the timestamp of each update to apply the changes in the order they occurred. For instance, if R2 made his edits after R1, then R2 changes will overtake R1, and the and the converged document becomes "Hello All".

However, such arbitration raises some questions, such as, is this the desired behavior? Is the intent of the users being preserved? And what happens if the replication conflict cannot be resolved, in case of simultaneous editions of the same item?

When timestamps are equivalent and cannot be used to resolve the conflicts, the resolution policy can be a rule like "Which account has the lowest identification number" or, “Which username comes first in the alphabet”. Nevertheless, these policies are also arbitrary as long as they are applied consistently by all peers.

Occasionally, a radical approach is to abandon the efforts to reconcile conflicting changes, and do not arbitrarily combine updates and neither select one. In the case of replicas R1 and R2, a radical approach might be to discard changes, reverting to the original document, back to the word "Hello". This may not be the best decision for the network, and it certainly isn’t preserving the users’ intent when they make their edits, but ensures data consistency, because, the end result converges for all users.

Our proposal in this work is to use a consensus mechanism with SEC to ensure consistency. However we cannot guarantee that operations are applied in order, so we will use conflict-free replicated data types (CRDTs), which are designed to support operations that can be safely applied in any order. We also reconcile CRDTs with a reputation system in which users spend reputation credits to evaluate other user changes, thereby guaranteeing that the final convergence preserves the intent of users, that is, an intent-preserving consistency.

Next, we explore the concept of CRDTs, its possibilities and our application of this technique.

1.3 CRDTs: Conflict-Free Replicated Data Types

A conflict-free replicated data type (CRDT) [15] is an abstract data type designed to be replicated at multiple processes. "CRDTs exhibit the following properties: (i) a replica can be modified without coordinating with other replicas; (ii) when any two replicas have received the same set of updates, they reach the same state, ensuring a convergent state". [18]

1.3.1 Decentralized Conflict Resolution

Let's take a step back and look at decentralized ways to resolve replication conflict issues, and why we understand that CRDT may be the best way to go about our work. Basically we are going to talk about two forms of conflict resolution, Operational transformations (OT) [36] and CRDTs [15].

"Operational transformations (OT) is a optimistic consistency control algorithm based on a client-server model" [37]. Google Docs [3] is an example that uses OT: each user is a client that owns a replica of the document, and multiples clients make simultaneous changes, so that each will be in a different state. Google docs acts as a central server to mediate changes. Clients changes are sent to the server, which accommodates different states, deciding the order in witch the set of pending operations should be applied (using rules such as last-write win) and forward them to the others clients. This technique ensures SEC, and preserves the intentions of the clients, but depends intrinsically on a central server to decide the order and manage operations.

However, our goal is to use a permissionless P2P environment, therefore we cannot rely on a central server. Unlike Operational transformations (OT) [36], CRDTs [15] can adopt any network topology like P2P and is resilient to network partitions, which makes it decentralized. Therefore, CRDTs [15] offer the potential to resolve replication conflicts when concurrent operations occur on replicas in a decentralized environment.

Before we detail CRDTs and our proposal for its use, let's revisit three related mathematical properties: commutativity, associativity, and idempotence. These properties are important to ensure the correct operation of CRDTs in distributed environments, as we discuss next.

Commutativity is the property that the order of operations does not affect the final result. In other words, two operations can be executed in any order and the result will be the same. An example of a commutative operation is the addition of real numbers. For example, if we have two numbers, 2 and 3, the sum of $2+3$ is equal to the sum of $3+2$.

Associativity is the property that the order of operations does not matter, as long as the operands only operate with their neighbours. In other words, when multiple operations are performed in sequence, the result is the same regardless of the order in which the operations are grouped. An example of an associative operation is the multiplication of real numbers. For example, if we have three numbers, 2, 3, and 4, the result of (2x3)x4 is the same as the result of 2x(3x4).

Idempotence is the property that applying the same operation multiple times has the same effect as applying it once. In other words, if we apply an operation several times, the final result will be the same as if the operation were applied only once. An example of an idempotent operation is the insertion of an element into a set. If we add an element to a set twice, the resulting set will be the same as if we added the element only once.

A CRDT is a type of data structure that allows multiple users to modify the same data concurrently without conflicts or inconsistencies. The basic idea behind CRDTs is to use data structures that are designed to merge multiple updates to the same data in a way that ensures consistency across all replicas, even when updates are made concurrently. Therefore, even though all replicas may diverge temporarily, the data will eventually become consistent (strong eventual consistency).

There are two main types of CRDTs: operation-based CRDTs (CmRDTs) and state-based CRDTs (CvRDTs).

Operation-based CRDTs (CmRDTs) represent the data as a sequence of operations that can be applied to reconstruct the data. When multiple users concurrently update the data, each user generates a set of operations, and these operations can be merged together to produce a consistent state across all replicas.

State-based CRDTs (CvRDTs), on the other hand, represent the data as a state, and updates to the data are made by merging the current state with the updates generated by other replicas.

Next, we detail the fundamental differences between these two approaches and the connections with our proposal.

1.3.2 CRDTs Approaches: CvRDTs & CmRDTs

State-based CRDTs are also known as convergent replicated data types (CvRDTs), while operation based CRDTs, as commutative replicated data types (CmRDTs). The two alternatives are theoretically equivalent, since each one can emulate the other. However, there are practical differences, which we detail next.

CvRDTs: In state-based CRDTs, the client sends the complete new state of data to all other clients. Users can merge remote changes with their own changes, and the CRDT ensures a consistent policy for resolving conflicts and converge.

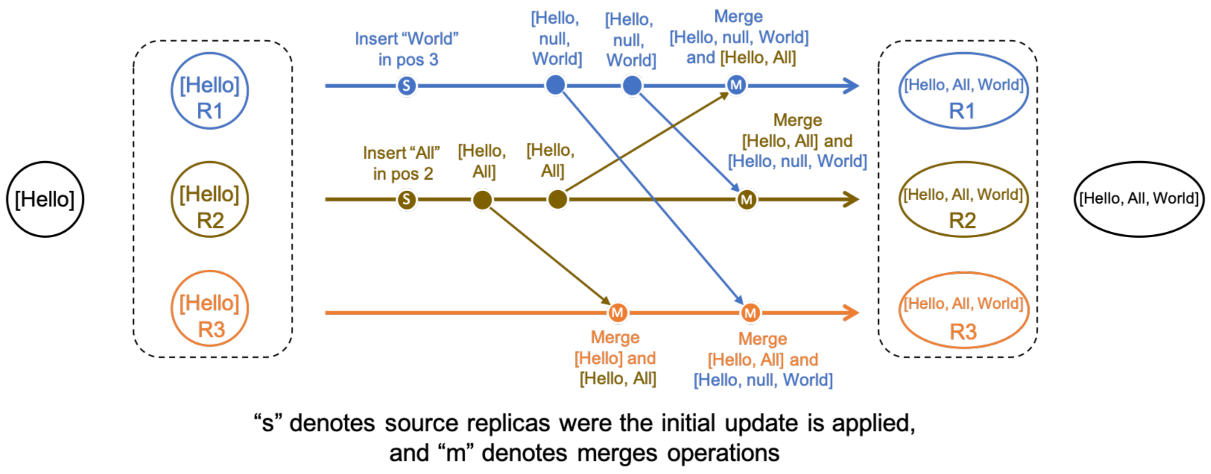

[Hello]
[Hello] R1
[Hello] R2
[Hello] R3
Insert "World" in pos 3
[Hello, null, World]
[Hello, null, World]
Merge [Hello, null, World] and [Hello, All]
Insert "All" in pos 2
[Hello, All]
[Hello, All]
Merge [Hello, All] and [Hello, null, World]
Merge [Hello] and [Hello, All]
Merge [Hello, All] and [Hello, null, World]
[Hello, All, World] R1
[Hello, All, World] R2
[Hello, All, World] R3
[Hello, All, World]
"s" denotes source replicas were the initial update is applied, and "m" denotes merges operations

Figure 3 - CvRDT (converged replicated data type) - State Based

Considering Figure 3, when a replica receives an update from a client, it first updates its local state and some time later sends its full state to another replica, as we can see in replicas R1 and R2. Therefore, occasionally, each replica is sending its complete state to some other replica in the system. When a replica (e.g. R3) receives a state from another replica (e.g. R2), it applies a merge function to combine its local state with the state it just received. Likewise, that replica could also occasionally send its state to another replica, so that each updates eventually reach all replicas in the system.

"One of the key underlying concepts of CvRDTs is the semi-lattice. A semi-lattice is a partially ordered set in which all elements have a supremum (i.e., a least upper bound). In the context of CvRDTs, the Semi-Lattice is used to model the precedence relationship between updates" [38]. "Each update in a CvRDT can be viewed as an element of the semi-lattice. Replicas can receive updates in any order but will always converge to the same consistent state because the semi-lattice ensures that updates are applied in the correct order, respecting the precedence relationship" [38]. "For the set of all possible system states to be a semi-lattice, the merge operation has to be idempotent, associative, and commutative" [15].

Furthermore, "CvRDTs are monotonic. This means that once an update is applied, it is never undone or reversed. All updates are cumulative, which allows replicas to be updated independently without needing to explicitly coordinate with each other" [38].

Back to the example in the Figure 3, when an insert operation is performed on a replica, a state update is created that is propagated to other replicas. If two replicas enter different elements at the same time, such as R1 entering "World" and R2 entering "All", they will create two independent state updates that can be applied in any order.

In summary, a CVRDT is a type of CRDT that automatically converges to a single consistent state after multiple updates in independent replicas, using the concept of semi-lattices to model the precedence relationship between updates and being monotonic to allow for independent updates.

The predominant use of CvRDTs is in file systems like NFS, AFS and Coda [15] as well as key-value storage systems like Dynamo [39] and Riak [34].

CvRDT properties:

- The merge function must be:
  - Commutative
  - Associative
  - Idempotent
- The merge function provides a join for any peer of replica states, so the set of all states forms a semilattice.
- The update function must monotonically increase the internal state. There is a partial order in the states, and merge operations and updates increase their states in that partial order.

CmRDTs: In operation-based CRDTs, a replica does not send its complete state to another replica (which can be huge). Instead, a replica just sends the update operations to all other replicas, and expects them to replay those updates. Going back to our example, considering an initial vector containing the word "Hello" from a post, as we can see in the example of Figure 4.

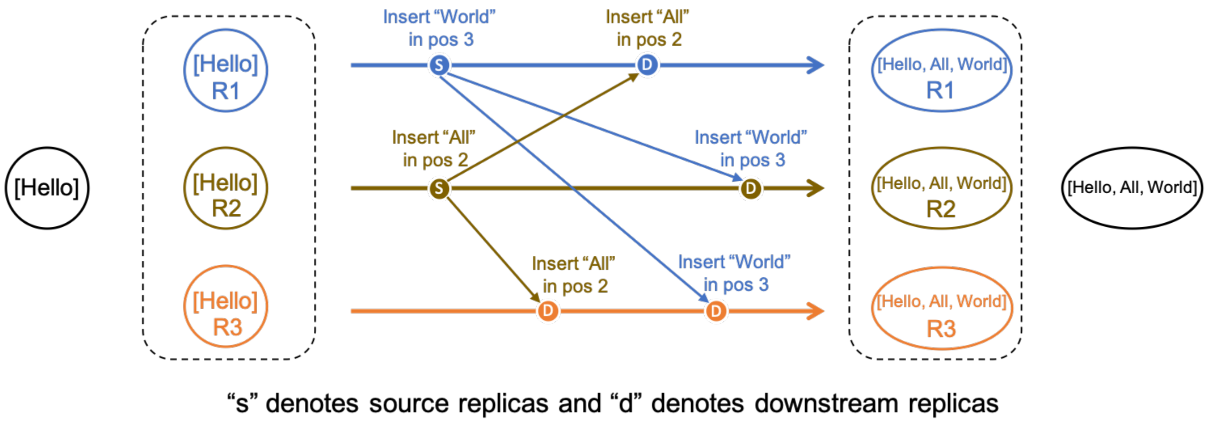


Figure 4 - CmRDT (commutative replicated data type) - Operations Based

Considering Figure 4, we initially have a vector containing the word "Hello" and this information is consistent in the 3 replies of 3 Wikipedia users. In the next moment, two replicas perform update operations, replica R1 inserts the word "World" into position 3 of the array, replica R2 inserts the word "All" into position 2 of the array.

Replica R1 transmits its insertion operation to replicas R2 and R3, replica R2 transmits its insertion operation to replicas R1 and R3, and these updates arrive at the destination replicas in different orders in time. How do these replicas converge then? They can converge if these updates operations are commutative: no matter which order these updates are applied at a replica the resulting state will be the same.

There are some delivery requirements to ensure that each update operation is transmitted only once to a replica because, in some formulations of CRDTs, the operations are not always idempotent, and once the operations arrive at the replica they can be applied in any order, asynchronously.

CmRDT properties:

- The concurrent operations must be commutative, however they are not necessarily idempotent.
- Requires a single (without duplication), but not necessarily ordered semantics.

Some examples of using CmRDTs are the cooperative systems, Bayou [40] and IceCube [41], and Automerge [16], which is an open source JavaScript library for implementing CRDTs.

1.3.3 CmRDTs and CvRDTs: Pros and Cons

The two approaches are theoretically equivalent, as each one can emulate the other, but there are practical differences:

CvRDT - State-based CRDTs

- One advantage: They are generally simpler to design and implement, considering that no additional replication mechanisms are needed, your only communication channel requirement is some kind of gossip protocol to propagate the complete state, much more simple compared to other type.

- One disadvantage: There is considerable communication overhead imposed by full state propagation. We can see that the size of the data objects that represent the state grows significantly over time, and when a modification occurs, it would be more efficient just propagate the modifications instead of the complete state.

CmRDT – Operation-based CRDTs

- One advantage: This model exposes a greater efficiency, by only propagating the information about the operation performed, instead of the complete state which is usually smaller than the full state.

- One disadvantage: In "CmRDTs, update operations are propagated in a way that ensures that all nodes eventually converge to the same state. This requires that updates are applied in causal order, which reflects the order in which updates were made by the different nodes"" [38].

  "To maintain the causal order, it is necessary to ensure that updates are transmitted with causality guarantees, meaning that they are delivered to other nodes in the same order in which they were made. This is typically achieved using a messaging system that provides ordering guarantees, such as a total order broadcast or a reliable multicast protocol" [38].

  Ensuring causality guarantees can add complexity to the implementation of CmRDTs, as it requires careful management of the order in which updates are applied, and it can also impact the performance of the system if the messaging system used to provide ordering guarantees is slow or unreliable.

In summary, "CRDTs serve as a robust foundation to model concurrent updates in distributed collaborative local-first applications" [42]. However, since by design, CRDTs may have corner cases when they face conditions of specific conflicts, such as concurrent editions of the same part of the document, the decision arbitrated by the CRDT may not be the best decision for the network and may not preserve the intention of the users.

To exemplify this corner case of concurrent editions, let's discuss the example of Martin Kleppmann's paper: "Alice and Charlie are concurrently typing into the same document at once. The data already reads "Hello!". Alice decides to insert her name in the text, to make the data read "Hello Alice!". Meanwhile, Charlie has the same idea and inserts his name, so the text reads, "Hello Charlie!". "Each individual character they add is treated as a separate insertion into the document, and they are simply added, at the same insertion point, ordered by timestamp" [43].

Adding the typed characters in insertion order means that everyone sees the same copy of the data, but the jumble of edits that results (H e l l o A l C i h a r c l i e e !) isn't what either user intended. In these cases, often require human intervention to solve these specific conflicts (quasi-CRDTs) and make the best decision.

In our proposal, the operations will be ordered by a consensus algorithm. We will use the Freechains [14] consensus algorithm as a way to resolve CRDT conflicts automatically, which we will discuss in more detail further, in the P2P systems section. Our observation is that, since the Freechains [14] consensus algorithm already relies on human interactions, we can use it to resolve conflicts automatically, while preserving the user's intent.

Freechains itself is composed by a three-layered CRDT scheme to build decentralized collaborative applications. The first layer of Freechains uses state-based CRDTs (CvRDTs) at the transport layer. This layer ensures that data is replicated across multiple nodes without conflicts or inconsistencies. State-based CRDTs achieve this by maintaining a single state value, which is then replicated across all nodes in the network. The second layer of Freechains uses operation-based CRDTs (CmRDTs) at the application layer. CmRDTs enable users to modify the replicated data in a manner that is consistent with the overall state of the system. CmRDTs achieve this by applying commutative and associative operations that can be merged without causing conflicts. Finally, after consensus is applied, Freechains utilizes quasi-CRDTs with arbitrary operations. This layer allows users to perform arbitrary operations on the data once consensus has been established. These operations may not be commutative or associative, so they require consensus to avoid conflicts.

Freechains will provide us with a permissioless P2P environment, for the other part of our work, the consistency of the datasets encapsulated in JSON, we will use a CRDT tool called Automerge [16], which we detail next.

1.3.4 Automerge: a JSON CmRDT

Automerge [16] is an open-source library for building collaborative applications that provides JSON datasets designed as a CmRDT, and relying on commutative operations [44]. This approach allows arbitrary concurrent changes to a shared dataset or a document, which are merged automatically in a principled manner, without requiring any centralized coordination.

Automerge's JSON data model allows to represent complex data structures, such as nested objects and arrays, and supports a wide range of primitive types, including strings, numbers, booleans, and dates.

Automerge is network agnostic, meaning that it does not depend on any specific network architecture or communication protocol. Instead, it allows users to read and modify data even while their device is offline, and then synchronize their changes with instances of the application on other devices when a network connection is available.

In addition, Automerge does not require data to be sent via a centralized server, but rather allows local and peer-to-peer networks to be used for synchronization. This makes it ideal for building decentralized applications that can operate without relying on a centralized authority or single point of failure.

Automerge documents are represented as directed acyclic graphs (DAGs) of changes, in which each change consists of an actor ID, a list of parent changes, and a list of operations

The use of CmRDTs provides strong eventual consistency [30], meaning that all replicas eventually converge to the same state, even in the presence of temporary failures. As discussed in Section 1.3, this is achieved by ensuring that all operations on the shared data are commutative, meaning that they can be executed in any order without affecting the final result.

Consider the example of Figure 5, where two Wikipedia users edit a term with the word "Helo". User 1 realizes that there is a grammatical error in the word and corrects it to "Hello" by inserting the letter "l". User 2 does not notice this error, and decides to emphasize the word by including the exclamation mark "!" at the end, resulting in "Helo!". The intent is for the system to achieve consistency and arrive at a result with the two adjustments made by users to all user replicas present (Hello!).

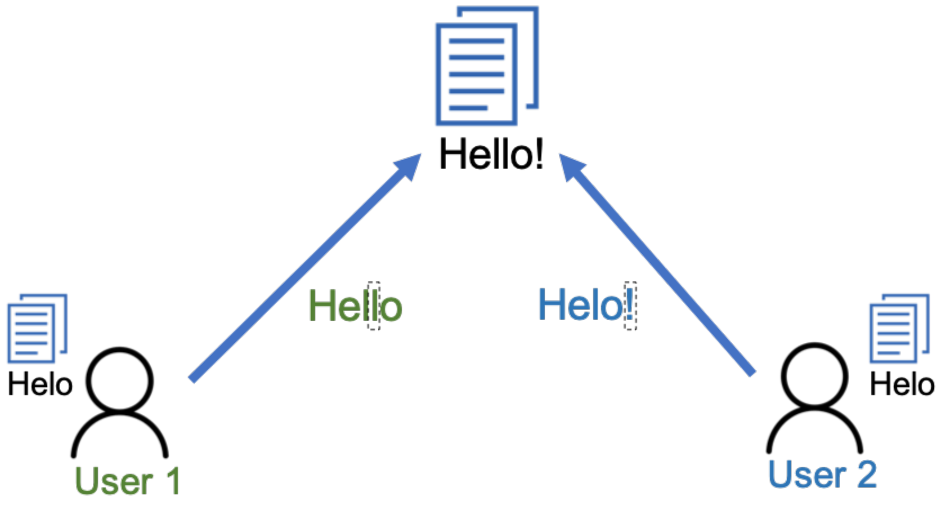
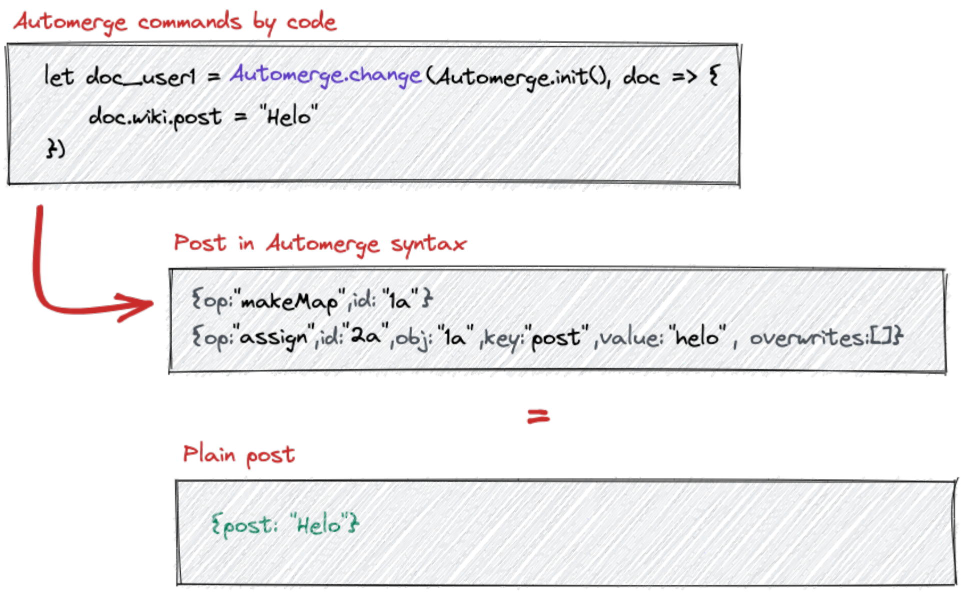


Figure 5 - Wikipedia update operations

From Automerge, to create the initial structure of the example above (post: "Helo"), we use the command "Change" via code: including an element called "Post" containing the word "Helo" (doc.wiki.post = "Helo"), as we can see in the Figure 6. This document should then be replicated to the other replicas/users of the system, through an external mechanism, as Automerge [16] itself does not do this replication.

Still in Figure 6 we have the post ("helo") in Automerge syntax and just below the same post in plain.



```
let doc_user1 = Automerge.change(Automerge.init(), doc => {
    doc.wiki.post = "Helo"
})
```

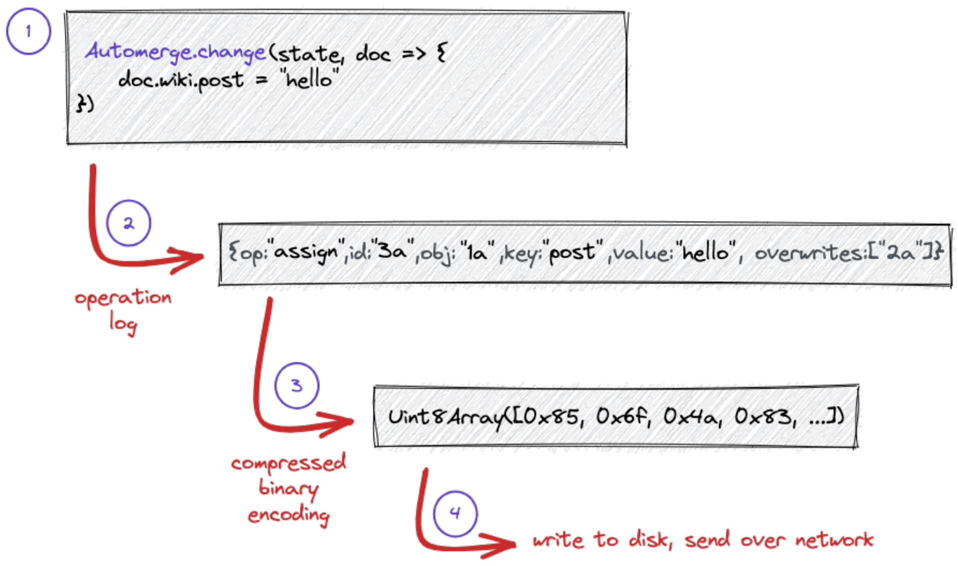
```
{op:"makeMap",id: "1a"}
{op:"assign",id:"2a",obj: "1a",key:"post",value: "helo", overwrites:[]}
```

```
{post: "Helo"}
```

Figure 6 - Original post

At first, the Automerge [16] syntax may appear larger than the plain text post, but this quickly reverses as the post gets more content, since only the update operations corresponding to the changes between the current state of the post in the chain and the local-first edition by the user will be persisted.

Also through the "Change" command of Automerge, User 1 of our Wikipedia will perform its update operation by changing the word "Helo" to "Hello". In the Figure 7 below we show a step by step of this edition and in the sequence the explanation of each of these steps.



```
Automerge.change(state, doc => {
    doc.wiki.post = "hello"
})
```

```
{op:"assign",id:"3a",obj:"1a",key:"post",value:"hello", overwrites:["2a"]}
```

```
Uint8Array([0x85, 0x6f, 0x4a, 0x83, ...])
```

Figure 7 - Automerge: Data encoding

In Figure 7, we observe the following steps:

1. User 1 uses the Automerge.change command to change the word "helo" to "hello";
2. Operation log: the command of step 1 causes these operations to be recorded in Automerge syntax format:
    - Operation type: assign
    - Id of this operation: 3a
    - Object id: 1a
    - command key: post
    - value to be applied: "hello"
    - operations to be overwritten: 2a
3. Compressed binary encoding: These operations will be compressed encoding in binary [45] to reduce the size of the operations package;
4. Write to disk (local first) and afterwards send over network to get consistency among others replicas.

This same walkthrough will be done by User 2 when he performs his edit to "Helo!" or "Hello!" depending on its initial state being "Helo" or already consistent with the edit made by User 1 "Hello".

Still in Figure 7, more specifically in step 4, highlighting an interesting feature of Automerge witch is, after generating an operations log table, it applies a binary compressed encoding [45] to this data, reducing the size of the operations package that will be stored and later shared/transmitted.

Table 1 - Comparison between unconverted vs binary converted JSON data files.

File format	**File size**
JSON (uncompressed)	146.406.415 bytes
JSON + gzip	6.132.895 bytes
Custom binary format	302.067 bytes

Table 1 demonstrates a comparison between an uncompressed JSON file, a gzip-compressed JSON file, and the file containing the same data in binary format. We can see a big reduction in the size of the file that will be stored on the replica, and afterwards replicated among the other replicas.

Managing Conflicts

Automerge allows different nodes to independently make arbitrary changes to their respective copies of a document. In most cases, those changes can be combined without any trouble. For example, if users modify two different objects, or two different properties in the same object, Automerge creates branches for each of them, then it merges those branches to combine those changes (Figure 8). Even through it is the same object, due to the fact that there are no conflicting editions.

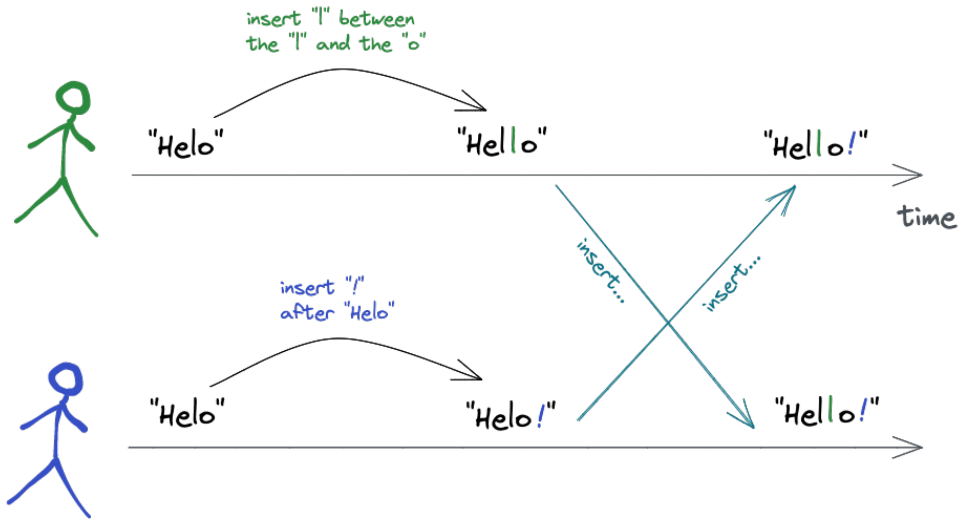
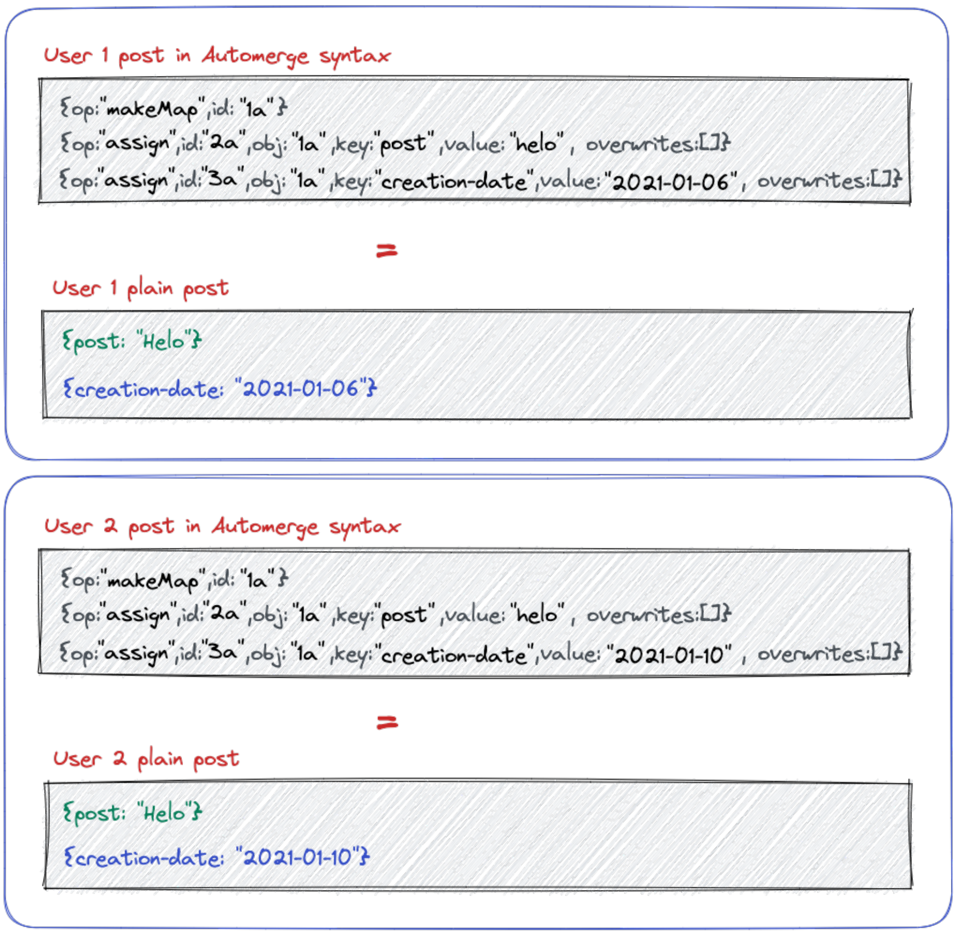


Figure 8 - Automerge: Concurrent text edition

However, Automerge [16], as a CRDT tool, has trouble dealing with conflicts of simultaneous edits of the very same part of the document. Let's consider our previous example, but let's add one more element called "creation-date" for both 1 and 2 replicas (Figure 9).


User 1 post in Automerge syntax
=
User 1 plain post
User 2 post in Automerge syntax
=
User 2 plain post

```
{op:"makeMap",id: "1a"}
{op:"assign",id:"2a",obj: "1a",key:"post",value: "helo", overwrites:[]}
{op:"assign",id:"3a",obj: "1a",key:"creation-date",value:"2021-01-06", overwrites:[]}
```

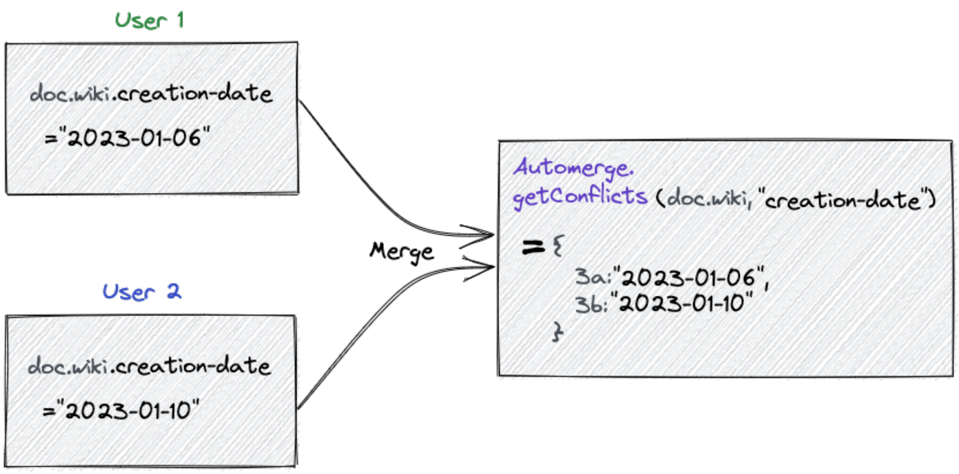
```
{post: "Helo"}
{creation-date: "2021-01-06"}
```

```
{op:"makeMap",id: "1a"}
{op:"assign",id:"2a",obj: "1a",key:"post",value: "helo", overwrites:[]}
{op:"assign",id:"3a",obj: "1a",key:"creation-date",value: "2021-01-10" , overwrites:[]}
```

```
{post: "Helo"}
{creation-date: "2021-01-10"}
```

Figure 9 - Posts with creation-date element

Let's now imagine an edit conflict situation, in which both users 1 and 2 edit the value of the same element "creation-date" (Figure 10).



```
doc.wiki.creation-date
="2023-01-06"
```

```
doc.wiki.creation-date
="2023-01-10"
```

```
Automerge.
getConflicts (doc.wiki, "creation-date")
= {
    3a:"2023-01-06",
    3b:"2023-01-10"
  }
```

Figure 10 - Automerge: Conflict resolution example

In this case (Figure 9), there is no clear and fair definition of which modification should prevail. This is the only case Automerge cannot handle automatically, because there is no well-defined resolution when users concurrently update the same property in the same object. Therefore, usually in conflict situations, Automerge preserves all the insertions, deletions or editions. If two or more users concurrently insert at the same position, Automerge will ensure that on all nodes the inserted items are placed in the same order. Every operation has a unique operation ID that is the combination of a counter and the actorId that generated it. Conflicts are ordered based on the counter first (using the actorId only to break ties when operations have the same counter value).

In these cases of concurrent editing of the same element, Automerge picks arbitrarily one of the concurrently written values as the "winner" (e.g. highest local time, or alphabetical order), and it ensures that this winner is the same on all nodes.

But this arbitrary decision does not preserve user intent and may not be the most interesting decision for the community or for the system, precisely because it is not based on a consensus among the participants. Let's expand this example of the Figure 10 at the code level to better understand how it proceeds.

```
// Create two different documents
let doc_user1 = Automerge.change(Automerge.init(), doc => {
  doc.wiki.creation-date = "2023-01-06"
})
let doc_user2 = Automerge.change(Automerge.init(), doc => {
  doc.wiki.creation-date = "2023-01-10"
})
doc_user1 = Automerge.merge(doc_user1, doc_user2)
doc_user2 = Automerge.merge(doc_user2, doc_user1)
```

Note that merge operations occur local-first, regardless of any connectivity - net agnostic behavior.

Now, doc_user1 might be either creation-date: "2023-01-06" or creation-date: "2023-01-10" – the choice is arbitrary. However, doc_user2 will be the same, whichever value is chosen as winner.

```
assert.deepEqual(doc_user1, doc_user2)
```

Although only one of the concurrently written values shows up in the object, the other values are not lost. They are stored in a conflicts object. Suppose doc.wiki.creation-date = "2023-01-10" is chosen as the "winning" value:

```
doc_user1 // {creation-date: "2023-01-10"}
doc_user2 // {creation-date: "2023-01-10"}
Automerge.getConflicts(doc_user1, 'creation-date') //
{'1@01234567': "2023-01-06", '1@89abcdef': "2023-01-10"}
Automerge.getConflicts(doc_user2, 'creation-date') //
{'1@01234567': "2023-01-06", '1@89abcdef': "2023-01-10"}
```

In the example there is a conflict on property *creation-date*. The object returned by *getConflicts* contains the conflicting values, both the "winner" and the "loser". The keys in the conflicts object are the internal IDs of the operations that updated the property *creation-date*. The next time that happens a conflicting property, the conflict is automatically considered to be resolved, and the conflict disappears from the object returned by *Automerge.getConflicts()*.

This is a consensus problem, there is no trivial solution. As discussed, arbitrary criteria, such as alphabetical order, highest local time or odd/even number order, may not preserve user intent. Therefore, our purpose of this work is to use the consensus mechanism of Freechains based on user reputation to determine which user edition will prevail over the others. As it is a system where users spend their reputation credits to positive (like) or negative (dislike) updates, and therefore the best evaluated will prevail, maintaining the consistency and quality of the network content and preserving the users' intention, since their likes (or dislikes) that will make an edition prevail (or not) among the other replicas.

We chose to use Automerge [16] as the CRDT tool of our work, especially for the support of a JSON object that makes the datasets to be used flexible and generic. We will use the Automerge data types, for example array, dictionary and primitive types (string, numbers, boolean and null). Another point that helped us decide for Automerge is the feature of reducing the size of the operations packet using binary compression, which optimizes our sharing of editing operations over the P2P network.

However, a drawback is that Automerge only can be operated through code, making it difficult to edit files for a standard user, not a developer. For that we have developed a tool called AMRW (Automerge-JSON command line editor) that encapsulates these commands in the code and allows the user to make edits into Automerge [16] files via the command line.

Automerge, based on CRDT operations, will deal with local-first conflicts, however it does not replicate this data among the other replicas spread over the P2P network. To carry out this replication, we will use Freechains as a P2P protocol along with its reputation mechanism to deal with malicious users and competing editions of the same item. We discuss more about this in Section 1.4.

1.4 Peer-to-Peer Systems

Computation systems can be classified in three broad categories: centralized, federated, and peer-to-peer (P2P) systems [46], as Figure 11 illustrates:

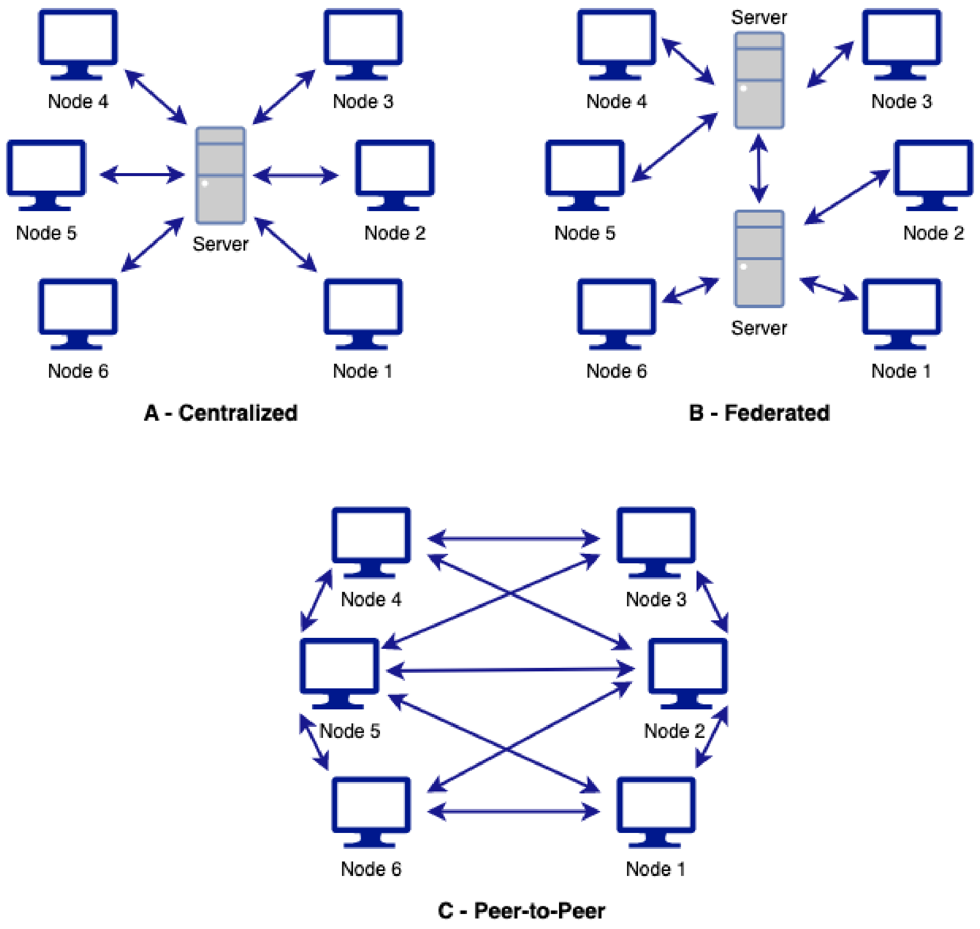


Figure 11 - Computation Systems

Figure 11.A is a representation of a centralized system. "In these systems, users are connected to a central service owner or “server”. The server stores data, which other users can access, controlling all devices connected to the network" [47]. An example of a centralized system is Wikipedia [5], which exposes a massive (logical) server to which users send requests and get the articles they requested.

Figure 11.B is a federated system, which is a middle ground between a centralized and peer-to-peer systems. "This type of system let users pick a server to sign up with, which gives them access to the entire network spread out across many different servers. This gives users more choices for applications, policies, and community cultures" [46]. "Email is an example of a federated protocol that everyone on the internet uses. Gmail is a popular email application, but if you use a different provider you can still communicate with anyone with an email address. In our context of content sharing, Mastodon is a federated social network that allows users to create their own instances and communicate with users on other instances. Each instance has its own rules and policies, and users can choose which instance to join based on their preferences" [46].

Figure 11.C is a peer-to-peer system. "P2P is a form of distributed system in which each node or computer is equally responsible for providing resources and performing tasks for other nodes in the network by direct exchange, since nodes are indistinguishable. Rather than requiring the intermediation or support of a centralized authority, note that, unlike client-server models, P2P consumers and providers typically run exactly the same software" [7]. This type of system is commonly used in file-sharing applications and real-time communication, such as video conferencing and online gaming.

The P2P models can be classified as pure or hybrid (Figure 12).

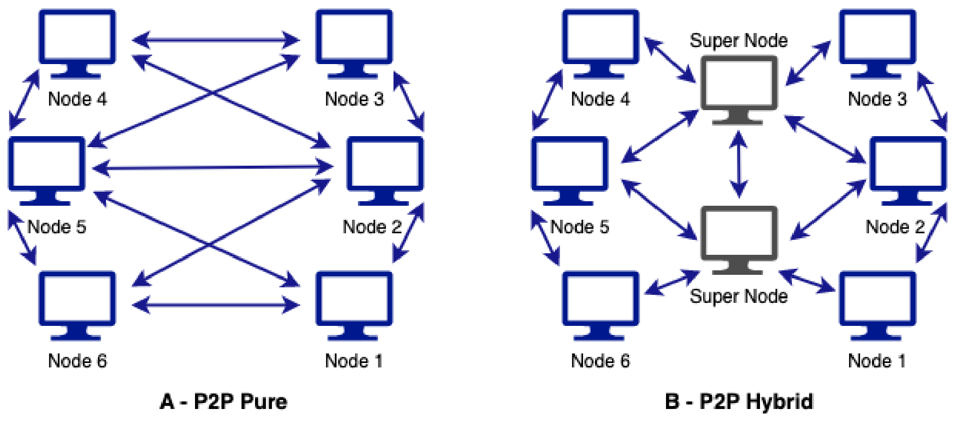


Figure 12 - P2P Pure x P2P Hybrid

Pure P2P, illustrated in Figure 12.A, refers to "totally distributed systems, in which all nodes are completely equivalent in terms of functionality and tasks they perform" [7]. "In pure P2P any single, arbitrary chosen terminal entity can be removed from the network without having the network suffering any loss of service" [47]. Freechains [14] is an example of pure P2P.

"Hybrid P2P, illustrated in Figure 12.B, includes "supernodes", which are nodes that function dynamically assigned, facilitating the interaction between peers by maintaining directories of metadata, or describing the shared files stored by the peer/nodes. The terms “peer-through-peer” or “broker mediated” are sometimes used for such systems. Although the end-to-end interaction and file exchanges may take place directly between two peer nodes, the "supernodes" facilitate this interaction by performing the lookups and identifying the nodes storing the files" [48].

At first sight, federated systems in Figure 11.B and P2P Hybrid systems in Figure 12.B seem very similar. However, the possibility of nodes interacting with each other, instead of only through the supernodes, is a differential between the hybrid P2P and the federated system. Also, in P2P nodes can connect to any supernode whereas, in the federated the node has to connect to the server providing the service it uses.

However, our goal is a system that doesn’t depend on central authorities to grant access, or to moderate content. Hence, conforming to Figure 11.C, we intend to transform a centralized Wikipedia [5] into a permissionless P2P alternative. In particular, we want the growth of the network not to be tied to a boundary controlled by predefined entities, but instead to be an organic growth, and that users themselves can define the content that should or should not be present on the network, preserving their intentions. So our work is based on a "pure P2P system, without any centralized control authority. In such systems all nodes are equivalent in functionality, and they are named as ”servant” (SERver+cliENT), which represents the capability of nodes to act as server as well as a client at the same time" [47].

1.4.1 Overlay Networks

A peer-to-peer (P2P) network is an overlay network. "An overlay network is a computer network that is built on top of other networks called underlay networks, in other words, overlay networks are supported by the infrastructure of underlay networks" (Figure 13) [49]. The underlay networks is basically the physical network.

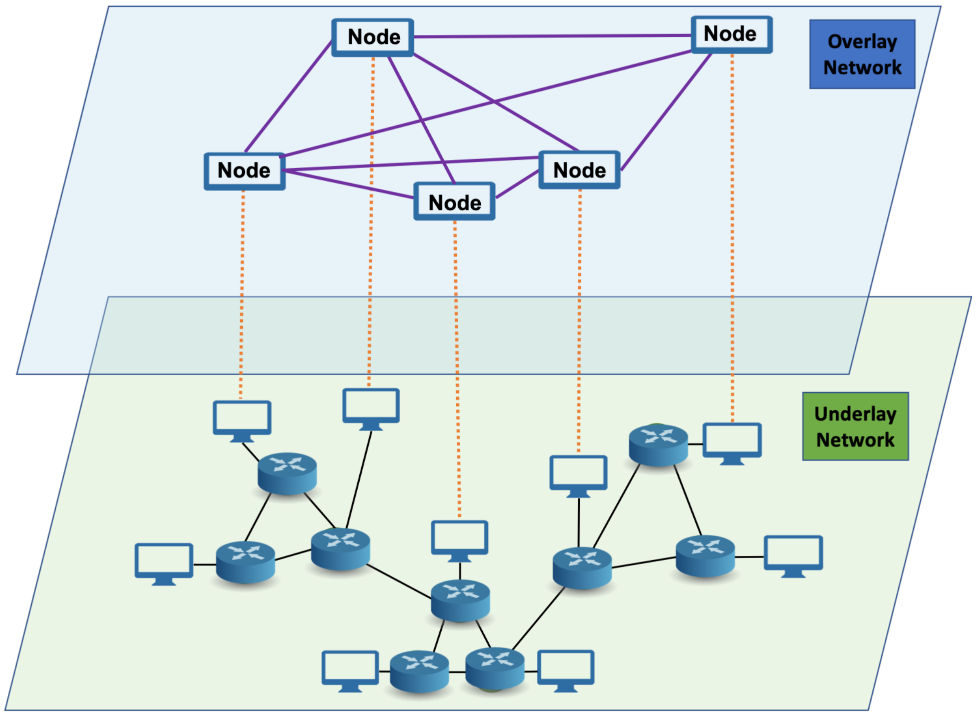


Figure 13 - Overlay and Underlay Networks

"In an overlay network, network nodes are connected with each other through a logical or virtual link which corresponds to a path in the underlying network. Multiples overlay networks can exist on top of the same underlying network, where each implements its own specific services" [50].

Considering our P2P Wikipedia, logical neighbors works collaboratively in a Wikipedia article, and new users need the addresses of other peers, which become logical neighbours (Figure 11.C), not necessarily linked to their physical location.

"P2P networks can be classified by the degree to which these overlay networks contain some structure or are created ad-hoc. Structure refer to the way in which the content of the network is located with respect to the network topology" [7]. Therefore, based on how the nodes in the overlay network are linked to each other, we can classify the P2P networks as unstructured or structured.

1.4.2 Unstructured vs. Structured P2P systems

Unstructured Peer-to-Peer (P2P) networks

"An unstructured peer-to-peer (P2P) network (Figure 14) is formed when the overlay connections are established arbitrarily. Here the peers join the network without any specific rules (loose rules), and network nodes can connect to any other node in the network, these characteristics make the growth of the network organic. Also there is no particular hierarchy or organization to the network's topology" [51].

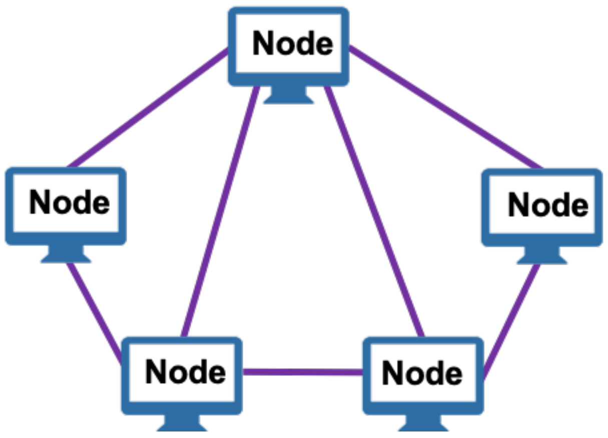


Figure 14 - Unstructured P2P Network

"One advantage of unstructured P2P networks is that can be easily constructed, a new peer that wants to join the network can copy existing links of another node and then form its own links over time. However, if a peer wants to find a desired piece of data in the network, the query has to be flooded through the network in order to find as many peers as possible that share the data " [52]. "One consideration with such networks is that the queries may not always be resolved, but to us this is not a problem since each replica maintains a local-first copy of all content. A popular content is likely to be available at several peers and any peer searching for it is likely to find the same, but, if a peer is looking for a rare or not so popular data shared by only a few other peers, then it is highly unlikely that search be successful. Since there is no correlation between a peer and the content managed by it, there is no guarantee that flooding will find a peer that has the desired data. Flooding also causes a high amount of signalling traffic in the network and hence such networks typically have a very poor search efficiency" [52].

An example of an unstructured P2P network is the BitTorrent protocol, which is commonly used for sharing large files such as movies or software. "In the BitTorrent network, nodes (or "peers") connect to each other and share pieces of a file with each other. When a new node joins the network and wants to download a file, it contacts other nodes in the network and requests pieces of the file. Over time, the new node builds up a complete copy of the file by downloading pieces from multiple peers" [53].

Characteristics of unstructured networks:

- No specific structure
- Easy design
- Easy to implement and deploy since nodes can connect in a random or ad-hoc manner.
- Well-suited for applications with unpredictable or dynamic resource requirements, such as file-sharing
- Offers more privacy and anonymity since queries and requests are broadcasted to all nodes, making it harder to track user activity.
- Higher churn rate tolerant
- Inefficient searching
- Flooding-based searching method

Examples: Napster [54], Gnutella [55], BitTorrent [56], Freenet [57], KaZAA [58] and Freechains [14].

Structured Peer-to-Peer (P2P) networks

In structured P2P networks (Figure 14), "the topology is tightly controlled and the contents are placed only at specified locations but not at random peers. These systems provide a mapping between the data identifier and it location on the network, in the form of a Distributed Hash Table (DHT). This allows each peer to be responsible for a specific part of the content in the network, so that queries can be efficiently routed to the node with the desired data " [47].

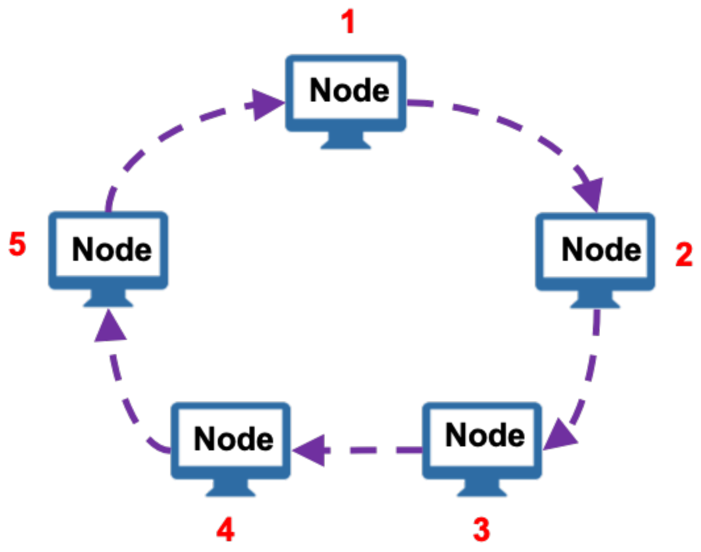


Figure 15 - Structured P2P Network

These networks use hash functions and assign values to every content and every peer in the network and then follow a global protocol in determining which peer is responsible for which content. This way, whenever a peer wants to search for some data, it uses the global protocol to determine the peer(s) responsible for the data and then directs the search towards the responsible peer(s).

Characteristics of structured networks:

- Specific structure
- Complex design
- Well-organized structure allows for efficient resource discovery and retrieval, with minimal network traffic.

- Well-suited for applications that require reliable and scalable resource location, such as content distribution or online marketplaces.
- Resistant to malicious nodes since the protocol can detect and exclude nodes that provide fake or incorrect information.
- Lower churn rate
- Efficient searching
- Hash-based searching method

Examples: Chord [59], CAN [60], Pastry [61] and Kademila [62].

In our work, all peers have a local-first copy of all content, so it is not necessary to use DHT, and also for that reason the search is not a problem either. In addition to these reasons, we chose to use an unstructured P2P system because we want to ensure that the solution is more flexible (there are no restrictions on the number or location of nodes in the network), scalable (there is no hierarchical structure to organize nodes and content, the network can grow organically as more nodes join the network, without incurring a management overhead), resilient to censorship (they do not have a single point of failure or centralized control that can be attacked or taken down, content is distributed across the network and can be accessed from multiple sources) and with greater privacy (there is no central authority that can monitor or control content or traffic on the network, content is shared directly between nodes without intermediaries). As we detail further, we use Freechains as our underlying permissionless P2P protocol.

1.4.3 Consensus in P2P Protocols

For a P2P system to have data consistency and reliable data, a permissioned approach is typically necessary. "In this approach the membership is controlled by a central authority that grants permission to users join the network and also participate in the consensus of conflicts situations of simultaneous editions" [56].

Consensus is key to eradicate Sybil attacks [9], which are the major threats to decentralized applications in general: without consensus, it is not possible, at the protocol level, to distinguish between correct and malicious users, because there is no way to know who is believable or not.

However, we want the growth of our network not to be subject to the approval of a central authority, or a predefined set of users, we want the entry of new users and consequently the expansion of the network to be organic. We also want the user's intentions to be preserved in the application of update operations performed by users on the P2P Wikipedia, without this being approved by some entity or set of predefined users.

Therefore, we are proposing a permissionless P2P system, where the participation is not controlled by an central administrator, or a predefined set of users. Anyone can participate in the consensus and validate the data. And also there are no central administrators allowing the users to participate or giving them the permission and rights to make the changes.

But we have challenges in this scenario, such as guaranteeing the delivery of information to all network peers and guaranteeing the consistency, accuracy and integrity of the information transmitted in the network. A major problem related to these challenges and inherent of a P2P permissionless scenario is data edition concurrency, and there are approaches to dealing with data concurrency and having consensus about what should be persisted.

Blockchains [63] are one form of permissionless network implementation, where network participants use consensus to confirm transactions and maintain data integrity on the network. Some of the most common consensus mechanisms used on permissionless blockchains [63] include Proof of Work (PoW) [13], Proof of Stake (PoS) [64] and Delegated Proof of Stake (DPoS) [65]. The overall objective is to ensure the reliability and integrity of the network. But permissionless blockchains have challenges and limitations, for examples: scalability (high computational demands required to validate transactions) and data integrity (vulnerable to attacks, for example Sybil [9]).

Bitcoin [13] is the first permissionless protocol to resist Sybils attacks [9] through consensus. Its key insight is to rely on a scarce resource the proof-of-work to establish consensus. The protocol is Sybil [9] resistant because it is expensive to write to its unique timeline (either via proof-of-work or transaction fees). However, Bitcoin and cryptocurrencies in general are not suitable for content sharing: (i) they enforce a unique timeline to preserve value and immunity to attacks; (ii) they lean towards concentration of power due to scaling effects; and (iii) they impose an external economic cost to use the protocol. These issues threaten our idea of free decentralization.

In our proposal, to prevent data edition concurrency problems, we are using conflict-free replicated data types (CRDTs) [15] through Automerge [16]. However in case of concurrency conflicts of simultaneous editions of the same item, Automerge gives us an arbitrary decision, which is not based on the users' intention. A decision based on the users' intention would be a consensual decision.

In order to reach a consensual decision, we will use the Freechains reputation system, in which users use their reputation credits to evaluate the content and agree on what should or should not follow on the network. This same reputation system will also serve to deal with problems of attacks (i.e. Sybil [9]), malicious behavior, SPAM and haters.

Next, we will talk about the solution we adopted at this work with Freechains, its characteristics and our motivations for using it as a non-permissionless P2P environment for our proposal.

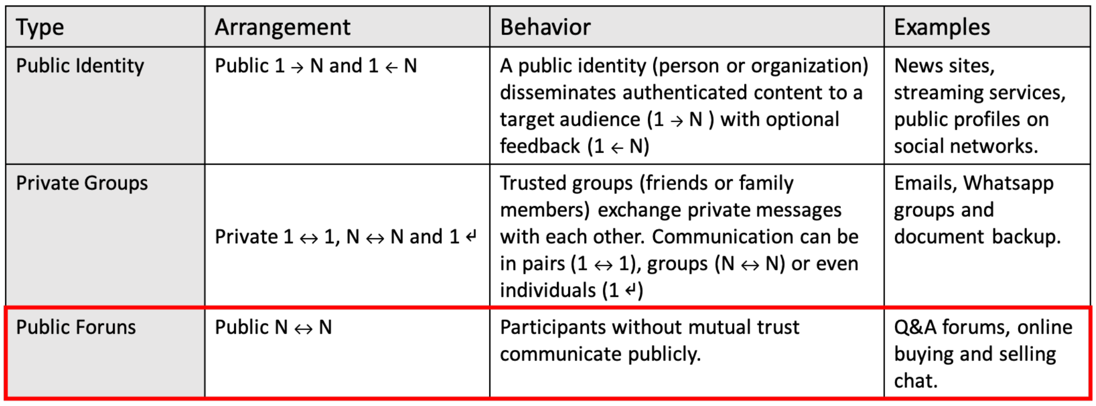
1.4.4 Freechains: Permissionless Reputation-Based Consensus

Freechains [14] is an unstructured (according to Section 1.4.2) permissionless (according to Section 1.4.3), Sybil-resistant peer-to-peer protocol for content sharing. It is used to disseminate content in a peer-to-peer network, without intermediation or centralized access control, and also without the need for trust between its users. In Freechains, a user posts a message to a topic, also called a chain, and all other reachable subscribers on the peer-to-peer network eventually receive the message.

Freechains main contribution is a reputation system that moderates content, at the same time, delivers network consensus. This is a unique feature, which brought us the interest in this specific tool, since we want to share data in a permissionless environment maintaining the users' intent. Another advantage is that the use of a reputation system together with a consensus algorithm, make Freechains resist to Sybil attacks [9], which are the major threats to P2P applications in general.

Freechains has two main contributions:

1. A minimal protocol for multiple content dissemination arrangements (as detailed in Table 2).



Type	Arrangement	Behavior	Examples
Public Identity	Public 1 → N and 1 ← N	A public identity (person or organization) disseminates authenticated content to a target audience (1 → N) with optional feedback (1 ← N)	News sites, streaming services, public profiles on social networks.
Private Groups	Private 1 ↔ 1, N ↔ N and 1 ↵	Trusted groups (friends or family members) exchange private messages with each other. Communication can be in pairs (1 ↔ 1), groups (N ↔ N) or even individuals (1 ↵)	Emails, Whatsapp groups and document backup.
Public Foruns	Public N ↔ N	Participants without mutual trust communicate publicly.	Q&A forums, online buying and selling chat.

Table 2 - The 3 types of Freechains chains and arrays

As indicated in Table 2, our focus is on pubic forums, which adheres to the Wikipedia P2P format, where all users are public and unknown at first.

2. An autonomous and decentralized reputation system. The Freechains reputation system governs the quality of posts and authors within public chains with the following objectives:

    - Highlighting quality content with a mechanism of likes and dislikes.
    - Fighting SPAM, fake news and illicit content by demanding reputation of authors and removing posts with a very low proportion between likes and dislikes.
    - And in our work, it will work as a mechanism to reach a consensual decision in concurrent editions of the same item.

1.4.4.1 Internal Operations

The Figure 16 illustrates the four basic concepts of Freechains: chains, blocks, authors and peers.

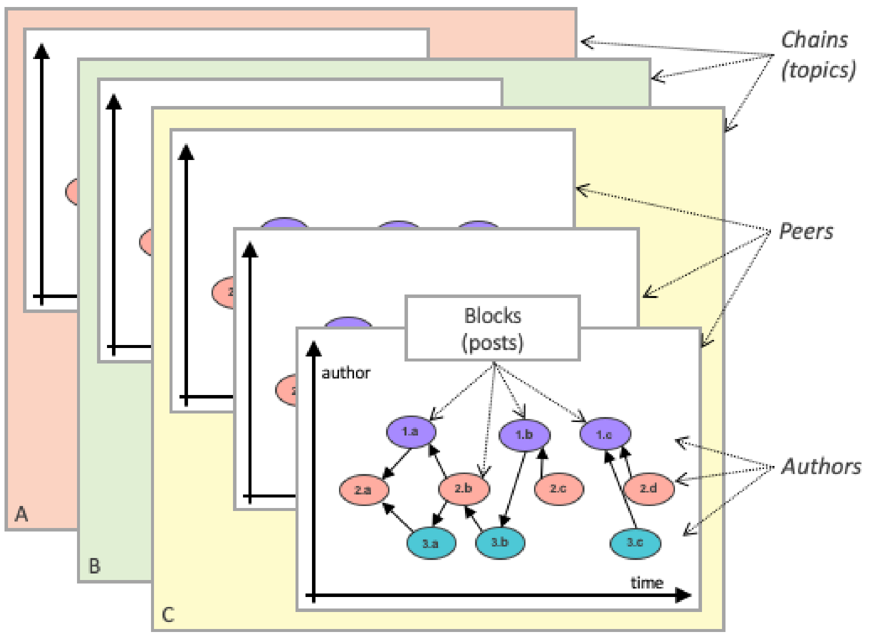


Figure 16 - Basic concepts of Freechains: chains, blocks, authors and peers.

"The chains, highlighted in colored squares (red/A, green/B and yellow/C), are independent of each other, and can represent, for example, a unique Wikipedia article with all of its update operations" [14].

"Each chain has a set of blocks with their posts, which are highlighted as colored ellipses (purple/1.x, red/2.x and green/3.x), where each color represents an author. The blocks of the chain form a causality graph indicating the temporal relationship between them. For example, the first message from the red author (2.a) occurred before the first messages from the purple and green authors (1.a and 3.a). The graph is a DAG and expresses the partial order among all blocks in the chain. Every chain has a preexisting genesis block (not shown in the image) which, by construction, is reachable by all blocks in the chain" [14]. The blocks would represent each edit operation of an article.

"The chain graph is fully replicated in all peers. In the figure, the yellow chain/C is replicated in three peers. The dissemination of the graph over the network is done by gossip, that is, the peers connect two by two to synchronize their graphs. In this way, the receipt of messages on the peer is eventual, as it depends on a peer-to-peer routing from source to destination" [14].

"The chaining of the blocks forms an acyclic directed graph immune to modifications (Merkle DAG). When creating a block, the hash of its metadata is calculated, which in turn includes the hashes of the message itself and of the previous blocks. That new hash uniquely identifies the block and can be recalculated at any time to check if the data has been modified. As the previous links are also identified in the same way, the graph can be checked up to its genesis (whose hash depends only on the parameters of the chain)" [14].

In the context of our work proposal, the idea is that users post Automerge objects, which contains only the difference in operations between the previous object stored in the Freechains chain and the local-first object edited by that user. This chain will be fed by all users since the first creation of the JSON object and all its subsequent editing operations, each edit post in a block of this chain. Posts are structured in a cryptographic acyclic directed graph that is immune to modifications (Merkle DAG). The graph is disseminated peer to peer in the network by gossip in an unstructured manner.

1.4.4.2 Reputation System

"The pemissionless P2P scenario we are proposing in this work is analogous to decentralized public forums, and this kind of arrangement is an invitation to abuse by malicious users with SPAM, fake news and illicit content. To mitigate abuse, Freechains' reputation system counts the number of likes and dislikes given to authors and posts and allows only users with a previous reputation to post new content. Posts from unreputable authors are withheld and need approval from reputable users" [14].

In our scenario, in addition to avoiding all the abusive behaviors mentioned above, which include Sybil attacks, we also rely on the reputation system to prevail the updates operations with the best reputation on the network, preserving the intent of the users in conflict concurrent editions operations. We understand that the updates operations stored in blocks that are the best evaluated, are the ones that most presents the intention of the network users. In the sequence, we will see how this reputation system works in depth.

In the context of our work, let's understand how an update operation performed local-first by a user persists in the Freechains chain as a block.

The unit of reputation, known as rep, works as follows:

1. Generation:

   (a) The first post of a chain adds +30 reps to the author (the pioneer).
   (b) Any post older than 24h counts +1 rep to the author, but limited to one per day: If the author has 10 posts in the last 7 days, he receives only +7 reps.

2. Consumption:
   (a) Any post younger than 24h counts -1 rep to the author.
3. Transfer:
   (a) A like from author A to post P from author B counts -1 rep for A and +1 rep for B.
   (b) A dislike from author A to post P from author B counts -1 rep for A and -1 rep for B. If a post reaches twice the number of dislikes in relation to the number of likes, and 5 dislikes at minimum, then its contents are blocked on the network.
4. Additional Rules:
   (a) Posts from unreputable users are held until they receive a like, and are neither chained nor relayed.
   (b) Users are limited to +30 reps.
   (c) Only posts newer than 90 days are considered.
   (d) In private chains, all users have infinite reputation.
   (e) In public identity chains, the owner has infinite reputation.

The first rule to generate reps (rule 1.a) is essential, since it would be impossible to post if nobody has any reputation (rule 4.a). Thus, the author of the first post shapes the initial culture of the chain by transferring his reputation to other authors, who in turn transfer it to other authors, expanding the community.

Going back to our P2P Wikipedia example, we can hold an article "wikitopic" in a chain of the same name, and posts containing update operations (e.g., the change from "Helo" to "Hello", shown in Figure 5) is stored in the chain to be sent to the network, more specifically, e.g., in the block with the id "2_12AB5C...". Users are be able to like or dislike this post, making the changes more believable or not.

The like and dislike commands act on an existing post:

```
$ freechains chain "#wikitopic" like 2_12AB5C... --sign=96700A...
```

In this case, the user who signed the like (96700A...), transfers 1 rep to the author of the referenced post (containing update operations) (rule 3.a). The reps command checks the reputation by passing the author's public key or hash identifier of the post to be consulted:

```
$ freechains chain "#wikitopic" reps 2_12AB5C...
1 <-- post reputation
```

The quality of posts is subjective and it is up to users to judge them with likes, dislikes or simple abstentions. On the one hand, as reps are finite, users must consider and avoid their indiscriminate spending. On the other hand, reps also expire after 3 months (rule 4.c), so users have incentives to cooperate with the quality of the chains. Considering that the replies are scarce, the banning of posts (rule 3.b) is only intended to prevent malicious users from acting, or undo a concurrent edition conflict by eliminating competing editions with lower reputations.

A block with banned content has to be kept in the chain graph, as the integrity of Freechains depends on the immutability of the graph. Thus, what is not retransmitted are the contents of the block with the message itself. Note that nothing prevents a block with banned content from being retransmitted if it receives later likes. For example, a part of the network with several likes was disconnected for a while and when it returned, it balanced the block's reputation. With this, update operations performed concurrently that have more likes remain visible on the network, which preserves the intent of network users, and also that operations that are not offensive, harmful or hateful remain visible, in addition to resisting attacks such as Sybil.

The Freechains reputation system seeks to offer minimally fair opportunities to participate in chains. Therefore, it restricts the number of posts by a given author in two ways: (1) it penalizes posts with less than 24 hours (rule 2.a); and (2) limits reputation gain by new posts to 1 rep per day (rule 1.b). The first rule prevents the same author from posting many messages in sequence, under penalty of consuming his own reputation too quickly. The second avoids accumulating reputation simply by posting too often.

The size of a chain's "economy" is its number of consolidated posts (rule 1.b), given that posts older than 24 hours are the only way to generate reps. Note that likes and dislikes only transfer reputation and the initial reputation of the first author becomes insignificant over time. The economy also depends a lot on the number of active authors, since the generation of reps per author is limited to 1 per day. This mechanism encourages the welcome of new authors, while discouraging dislikes as they drain 2 reps from the economy (rule 3.b). On the one hand, this disincentive contributes to discussions with a reasonable level of disagreement, as it prevents the chain from collapsing with an outbreak of dislikes. On the other hand, clearly unwanted content such as SPAM from users who contributed little, and competing editions with lesser reputation can be quickly removed from the chain with few dislikes (rule 3.b).

We chose to use Freechains in our work, especially for the reputation system combined with consensus, which will serve as a resource to prevent abusive behavior and malicious users, and to solve CRDT corner cases (e.g., simultaneous concurrent editing of the same item), maintaining consistency through a consensus that preserves the intention of the users. Another point is that Freechains DAG model ensures that the operations are published in consensual order from the Automerge CmRDT.

In this chapter we explained the theoretical foundations that underlie our proposal for a Permissionless Peer-to-Peer JSON Datasets system, going through each technology used, and detailing each decision we made along the way in order to make this scenario feasible and achievable.

2 PEER-TO-PEER DATASET ECOSYSTEM

This chapter presents the ecosystem of our proposal, a concurrent JSON dataset in a decentralized and permissionless environment. We now explain how to combine CRDTs, P2P networks and reputation techniques to solve the 3 challenges as follows:

1. How to avoid duplication and ensure the delivery of this data to all peers.
2. How to reach consensus on concurrent edits while keeping the users intent, ensuring the accuracy of the content.
3. How to prevent Sybil attacks, abusive behavior, malicious users and haters, keeping the integrity of the content.

To meet these challenges, our work enforces the following guarantees:

1. The data structures are consistent across all participants, even if there are concurrent edits, in the same parts of the dataset, by the use of a CRDT together with a P2P dissemination system.
2. The system works without the need for a central authority, that is, it is permissionless, by the use of a human reputation system.
3. The system data remains accurate, considering the users intent in case of concurrent editions. This is a subjective property and therefore without an automatic solution, hence the need for a human reputation system.
4. The system data remains integrate, regarding haters, SPAM, fake news and illicit content, also using the human reputation system.

We rely on CRDTs to ensure that concurrent editions do not generate inconsistencies, and that all participants view and access the same content. As introduced in section 1.3, we adopt Automerge [16] as a JSON CRDT.

However, as explained in Section 1.3, CRDTs are still subject to corner cases when editing the very same part of the document, Automerge handles these cases by arbitrarily deciding which edition prevails, which might not preserve the users' intentions, and may not be the most interesting for the network as a whole.

As introduced in Section 1.4, we rely on the reputation system of Freechains [14] to achieve consensus on competing edits of the same item while preserving user intent, and also for the network to auto-moderate abusive, hateful content, and Sybil attacks. Users use their reputation credits (reps) to like or dislike posts, which in turn contain the users' update operations. Therefore, the network, through the most credible posts (update operations) and most active users, self-moderate and determine which edition preserves the intent of most of the user community.

Furthermore the same reputation system also guarantees the integrity of the data from the point of view of haters, SPAM, fake news, illicit content and Sybil attacks [9], since the most active, and credible users can remove reputation from malicious editions that could harm the quality of the content, preventing them from being spread on the network. In summary, we eliminate a central authority, and posts are moderated by the network participants themselves.

2.1 Ecosystem of the Proposed Solution

To demonstrate the theoretical proposal of this article, we conceptualize an ecosystem using Freechains [14], Automerge [16] and 3 tools we have developed: AMRW, AMDIFF and AMPATCH.

AMRW is a command line tool with a predefined syntax for editing Automerge-JSON files, since local-first JSON files cannot be edited manually to generate Auromerge-CRDT update operations. AMDIFF is a command line tool based on Automerge's `getChanges` API to get the differences between two Automerge files. AMPATCH is a command line tool based on Automerge's `applyChanges` API, which consolidates the differences obtained between two Automerge files through AMDIFF into another Automerge file, updating it. Freechains provides permissionless decentralized P2P functionalities and an integrated reputation system, and Automerge provides the CRDT functionalities to manipulate our datasets. Figure 17 illustrates how these tools interact to form the ecosystem.

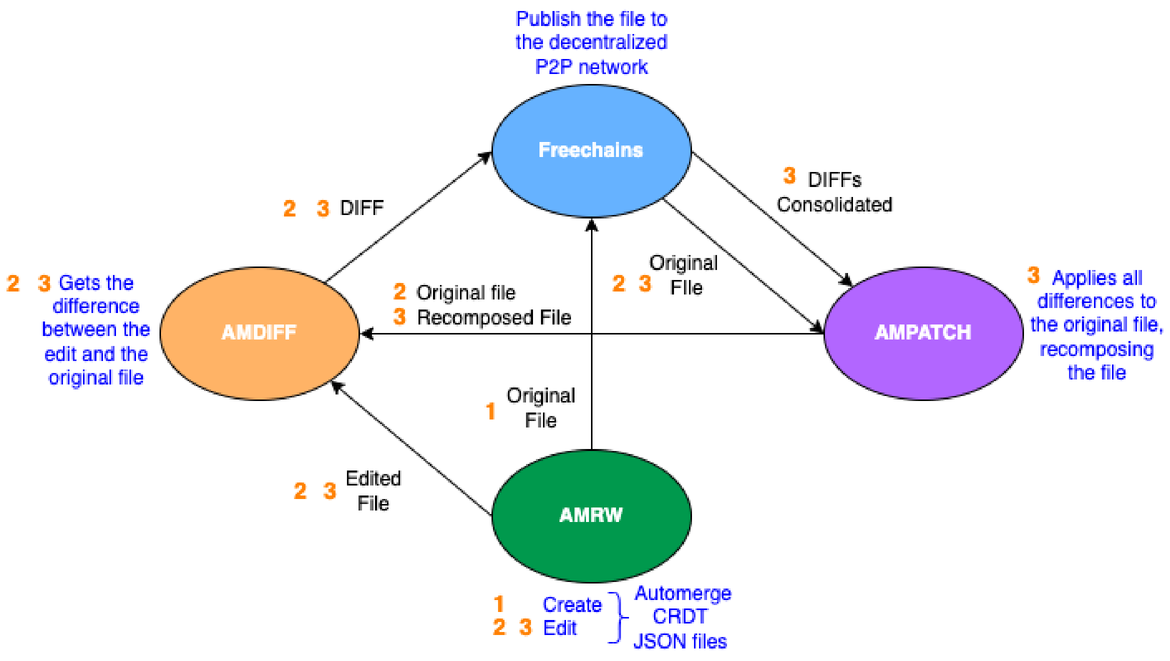

Publish the file to the decentralized P2P network
Freechains
2 3 DIFF
3 DIFFs Consolidated
2 3 Gets the difference between the edit and the original file
AMDIFF
2 Original file
3 Recomposed File
2 3 Original FIle
AMPATCH
3 Applies all differences to the original file, recomposing the file
1 Original File
2 3 Edited File
AMRW
1 Create
2 3 Edit
Automerge CRDT JSON files

Figure 17 - Ecosystem proposed.

Figure 17 illustrates the ecosystem of our proposed solution.

Let's assume that a user wants to create a new page on our P2P Wikipedia, and only he is online.

1. The user starts using AMRW to create a local CRDT JSON Automerge file to represent the Wikipedia article, and in the sequence posts this file for the first time in the corresponding Freechains chain.

2. In the sequence, the same user edits his local replica file with AMRW, and decides to post his edited file again. At that moment AMDIFF and AMPATCH come into action. First AMPATCH obtains the original file first posted in the Freechains chain. Since no differences were posted later, he makes this file available for AMDIFF to obtain the difference between the local replica edited by the user and the original file that was on the network. This difference is available for posting on the Freechains chain.

3. The user makes a new edit using AMRW. At that moment AMPATCH gets the original file first posted the Freechains chain, and in a second round it gets the post containing the difference from the user's first edit posted in the previous step. It applies this difference by recomposing the file identical to the one the user had before

making this new edition. This file is accessed by AMDIFF, which will obtain the difference between the local replica with the new edition made by the user, and the recomposed file with what was on the network. This new difference is is available for posting on a new post on the Freechains chain.

This loop will repeat over and over again as new users edit this page, making their local edits and network posts. Automerge's CRDT system will be in charge of maintaining consistency by merging concurrent editions, and users through the Freechains reputation system will be able to solve the CRDT corner case, which are simultaneous concurrent editions of the same item, element, word, etc. So that the edition that prevails, which has the most likes, maintains the intention of the majority of the network, guaranteeing the accuracy of the information. Through that same reputation system, users will block through dislikes malicious edits, hate speech, SPAM and Sybil attacks, maintaining data integrity,

Next, we detail the construction and operation of each of these 3 tools.

2.2 AMRW: Automerge-JSON Command-Line Editor

As mentioned before, Automerge stores data structures in a JSON-like CRDT through operations. These operations are recorded in a file in binary format. However, Automerge is operated via code and, more importantly, does not provide a mental model to navigate through the JSON elements and operate them individually. Therefore, we developed a tool to perform operate files via the command line with a clear semantics, which allows us to navigate through elements, and read and write to them. In summary, we provide a mental model to edit an Automerge-JSON as if we were editing a common JSON file.

AMRW saves a local binary file with Automerge operations that corresponds to the created/edited JSON. It also allows converting the Automerge-JSON binary files to human-readable JSON, that supports a wide range of datasets, including strings, numbers, booleans, and dates, as we can see on Appendix, for easy viewing the final results.

The semantic construction of AMRW took into account the possibility of completely navigating through the elements of the CRDT JSON Automerge through the command line by indicating a path as a parameter. This path is built item by item from the root element of the JSON to the elements you want to reach, always signaling its type and, if necessary, its index. For example, let's assume the JSON below:

```
{
"Sections": {
    "Introduction": "Distributed systems are",
    "History": "...",
    "Applications": ["Ethereum", "IPFS", "Ripple"]
    }
}
```

Suppose we want to read the second element "IPFS" of the "Applications" array. To do that we use the following command:

```
$ amrw <file> field Sections field Applications index 1 read
```

All the commands after the file name <file> and before the "read" mode are part of the navigation path. We enter the "Section" object and inside it we access the "Applications" array and display the element that is at index 1 of this array. The answer will be: **"IPFS"**.

To insert an element, we replace the operating mode "read" with "write", and define which field and values we want to insert, edit or delete. For example, suppose we want to insert a new field "Date of publication" as "June 01, 2023". In the command below we notice that the path navigates inside the Sections field, enables the writing mode, informs that it is going to write in an object "Sections" the element "Publication date" of type string, with the content "June 01, 2023".

```
$ amrw <file> field Sections write object ins "Publication date" string
"June 01, 2023"
```

The result is the JSON below:

```
{
"Sections": {
    "Introduction": "Distributed systems are",
    "History": "...",
    "Applications": ["Ethereum", "IPFS", "Ripple"],
    "Publication date": "June 01, 2023"
    }
}
```

An important consideration is that the creation, edition and deletion operations of the local-first JSON file must be done through explicit commands, which this tool implements, so that these update operations are actually recorded in the internal history of the Automerge saved file. This file data we use to carry out the DIFFs and PATCHs and travel in the Freechains P2P network blocks.

The manual with the complete syntax of AMRW commands, are available in the appendix of this work.

AMRW is written in NodeJS, and is available at https://github.com/fabiobosisio/amrw.

AMRW operation example

Suppose we are going to create a local file to publish later on P2P Wikipedia. The first step is to initialize the file:

- Initializing an Automerge-JSON file:

  We initialize the file "p2p" with the "init" command:

  ```
  $ amrw p2p init
  ```

  At this moment the resulting file only contains an empty object:

  ```
  {}
  ```

- Creating an object (a JSON hash table):

  We now want to include an initial empty object called "Sections" with the "write" command. We only need to navigate to the root "object" and through the operation "ins", we insert "Sections", which is also an "object":

  ```
  $ amrw p2p write object ins Sections object
  ```

  As a result, the "Sections" field contains an empty object.

```
{"Sections":{}}
```

- Creating a field inside the object:

  We now want to insert a field "Introduction" into "Sections", containing the string "Distributed systems are".

  Starting from the root "object", we now need to navigate to its field "Sections", and then through the "write" command, we take it as an "object", and use the operation "ins" to insert the "Introduction" field, which is the string passed as argument:

  ```
  $ amrw p2p field Sections write object ins Introduction \
      string "Distributed systems are"
  ```

  As a result, the "Sections" field, which is an object, now contains a field "Introduction" with the given string:

  ```
  {"Sections":{Introduction: "Distributed systems are"}}
  ```

- Creating one more field inside the object:

  We now want to insert a new field "History" into "Sections", containing the string "...":

  ```
  $ amrw p2p field Sections write object ins History string "..."
  ```

  As a result, the "Sections" field now contains two fields:

  ```
  {"Sections":{"Introduction":"Distributed systems are","History":"..."}}
  ```

- Creating an array inside the object:

  Finally, we want to insert an "Applications" field, which is an empty array:

  ```
  $ amrw p2p field Sections write object ins Applications array
  ```

  As a result, the "Sections" field contains another "Application" field with an empty array.

  At this point we have the following object:

```
{
"Sections": {
    "Introduction": "Distributed systems are",
    "History": "...",
    "Applications": [ ]
    }
}
```

- Erase the contents of a given object:

  We now decide to delete the "History" field.

  Likewise, we need to navigate to the "Sections" field and user the "write" command to an object, but now we use the operation "del" to remove the field "History" and its contents:

  ```
  $ amrw p2p field Sections write object del History
  ```

  Therefore, we have the following object:

  ```
  {
  "Sections": {
      "Introduction": "P2P networking is ..."
      "Applications": []
  }
  }
  ```

Through these examples we demonstrate AMRW, which is a tool for local use, through the command line to edit a JSON file in Automerge format. More examples and AMRW commands can be found in the Appendix.

2.3 AMDIFF: Automerge-JSON Command-Line Diff

AMDIFF is a comand line tool to get changes between two versions of an Automerge-JSON file. In our work, only the differences between the users' local replicas and the chain's consolidated content will be transmitted and posted on the network. the AMDIFF is essential to obtain this difference. The result is encoded in the binary format of Automerge and is persisted as a local file.

Our implementation relies on Automerge's `getChanges` API to get the difference between two files. AMDIFF is written in NodeJS, and is available at https://github.com/fabiobosisio/amdiff.

AMDIFF operation example

- Getting the difference between two Automerge files and saving to disk in a binary format file:

```
$ amdiff old.am new.am
```

A file will be saved on disk with the name "old-new.diff" which is a composite of the names of the two compared files. This file will contain the differences between the two files compared, saved in binary format.

The manual with the complete syntax of AMDIFF commands, are available in the appendix of this work.

2.4 AMPATCH: Automerge-JSON Command-Line Patch

AMPATCH is a command line tool to apply (patch) changes obtained through AMDIFF to a different version of the document. AMPATCH pairs with AMDIFF by applying the differences obtained to a local replica, updating it. AMPATCH is also the tool that is used to recompose the contents of the chain, applying to the original file in the chain all the differences stored in the subsequent posts. AMPATCH receives the original file, the patch file to apply, and generates a third file with the patch applied. This result is encoded in the binary format of Automerge and stored locally.

Our implementation relies on Automerge's `applyChanges` API to apply the differences to a given file.

AMPATCH is written in NodeJS, and is available at https://github.com/fabiobosisio/ampatch.

AMPATCH operation example

- Applying (patching) the difference

  Applies (patches) the contents of the "old-new.diff" file generated by AMDIFF to the "old.am" file. The result is saved in a new file called "new.am".

  ```
  $ ampatch old.am old-new.diff new.am
  ```

A file named new.am is saved to disk, and its contents consists of the data contained in old.am with the content of old-new.diff (differences obtained through AMDIFF) applied.

The manual with the complete syntax of AMPATCH commands, are available in the appendix of this work.

2.5 Timeline Simulation

Figure 18 represents a simulation of a timeline to demonstrate the proposed ecosystem, but now with the interaction of 3 users A, B and C. The users interact with the system asynchronously, using AMRW to edit their local replica file, AMDIFF, AMPATCH, and Freechains publish commands for posting the differences and getting content from the network. In this simulation frame we can follow the edited files as well as their content and the corresponding commands that the tools are using internally. As a goal, we want to demonstrate the maintenance of data consistency, where users will eventually have locally the same replica of the content posted on the network, even with concurrent editions of multiple users. Next, we explain in detail each step of the simulation.

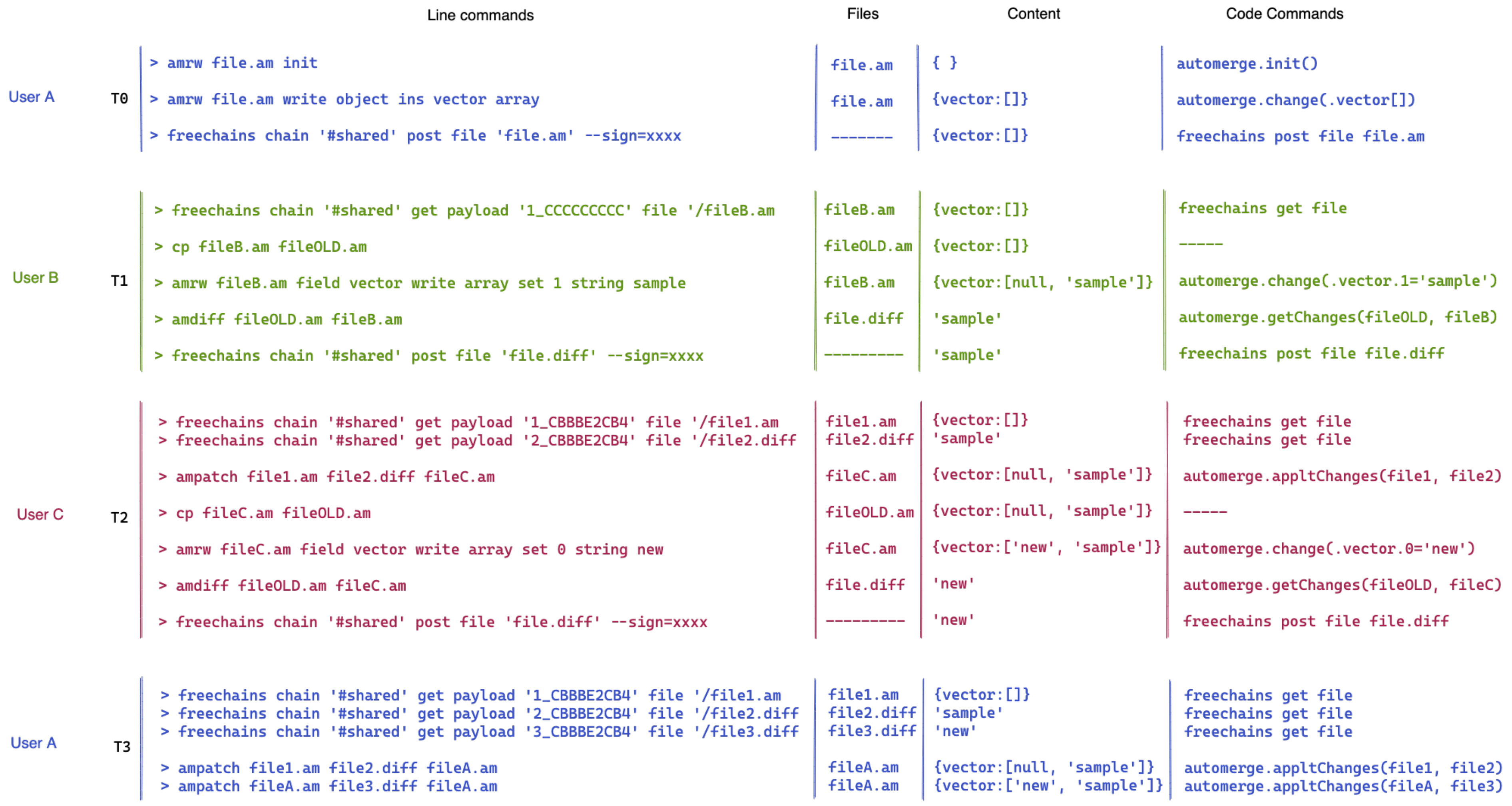
Next, we reworked the timeline, but this time using a fourth tool we developed called Freechains-JSON, which encompasses the AMDIFF, AMPATCH, and Freechains publish commands. In this new timeline we can see that this new tool reduces the number of steps and makes the operation of our solution simpler.

T0 - User A:

1. Initializes an Automerge-JSON file

   The Automerge file will store the data set that will subsequently be distributed in a decentralized and permissionless P2P environment. This file stores only operations, since Automerge is a CmRDT, and allows one to recompose the JSON at any time.

   And since it is a CRDT, most simultaneous editions will reach consensus, except when there are CRDTs corners cases like concurrent editions of the same item,



		Line commands	Files	Content	Code Commands
User A	T0	`> amrw file.am init`	`file.am`	`{ }`	`automerge.init()`
		`> amrw file.am write object ins vector array`	`file.am`	`{vector:[]}`	`automerge.change(.vector[])`
		`> freechains chain '#shared' post file 'file.am' --sign=xxxx`	`-------`	`{vector:[]}`	`freechains post file file.am`
User B	T1	`> freechains chain '#shared' get payload '1_CCCCCCCCC' file '/fileB.am`	`fileB.am`	`{vector:[]}`	`freechains get file`
		`> cp fileB.am fileOLD.am`	`fileOLD.am`	`{vector:[]}`	`-----`
		`> amrw fileB.am field vector write array set 1 string sample`	`fileB.am`	`{vector:[null, 'sample']}`	`automerge.change(.vector.1='sample')`
		`> amdiff fileOLD.am fileB.am`	`file.diff`	`'sample'`	`automerge.getChanges(fileOLD, fileB)`
		`> freechains chain '#shared' post file 'file.diff' --sign=xxxx`	`---------`	`'sample'`	`freechains post file file.diff`

User C	T2	`> freechains chain '#shared' get payload '1_CBBBE2CB4' file '/file1.am`
		`> freechains chain '#shared' get payload '2_CBBBE2CB4' file '/file2.diff`
			`file1.am`
			`file2.diff`
				`{vector:[]}`
				`'sample'`
					`freechains get file`
					`freechains get file`

		`> ampatch file1.am file2.diff fileC.am`	`fileC.am`	`{vector:[null, 'sample']}`	`automerge.appltChanges(file1, file2)`
		`> cp fileC.am fileOLD.am`	`fileOLD.am`	`{vector:[null, 'sample']}`	`-----`
		`> amrw fileC.am field vector write array set 0 string new`	`fileC.am`	`{vector:['new', 'sample']}`	`automerge.change(.vector.0='new')`
		`> amdiff fileOLD.am fileC.am`	`file.diff`	`'new'`	`automerge.getChanges(fileOLD, fileC)`
		`> freechains chain '#shared' post file 'file.diff' --sign=xxxx`	`---------`	`'new'`	`freechains post file file.diff`

User A	T3	`> freechains chain '#shared' get payload '1_CBBBE2CB4' file '/file1.am`
		`> freechains chain '#shared' get payload '2_CBBBE2CB4' file '/file2.diff`
		`> freechains chain '#shared' get payload '3_CBBBE2CB4' file '/file3.diff`
			`file1.am`
			`file2.diff`
			`file3.diff`
				`{vector:[]}`
				`'sample'`
				`'new'`
					`freechains get file`
					`freechains get file`
					`freechains get file`

		`> ampatch file1.am file2.diff fileA.am`
		`> ampatch fileA.am file3.diff fileA.am`
			`fileA.am`
			`fileA.am`
				`{vector:[null, 'sample']}`
				`{vector:['new', 'sample']}`
					`automerge.appltChanges(file1, file2)`
					`automerge.appltChanges(fileA, file3)`

Figure 18 - Execution timeline.

element, word and etc. In these cases, users will define which edition should prevail, maintaining the accuracy of the data.

2. Includes an array named vector in the created vector.

   Edits in the file will be made through the Automerge Change command implemented in AMRW, all recorded in the result file.

3. Post the created file on a Freechains public chain.

   Through the post command, user A uploads the data from the local CmRDT file to the Freechains chain. The Freechains reputation system will solve CRDT corner cases, where two users edit the same part of the document, in which case the block with the edition with the highest reputation will prevail, therefore, maintaining the intention of the network users, and will also deal with abusive and hating behavior, that will have their edits blocked due to low reputation.

T1 - User B:

1. Gets the first item in the public chain.

   Through the Freechains Get command, it obtains the content of the chain (at this moment only the edition of user A) and saves it in a local Automerge file.

2. Makes a copy of the downloaded file.

   Duplicates the file using the Linux cp command.

3. Edits the array created by user A and adds a string value to the one index of that array.

   Edits in the file will be made through AMRW, all recorded in the result file.

4. Gets the difference between the file downloaded from the network and the newly edited file, and saves in a diff file on disk.

   Using the AMDIFF, user B obtains the difference of operations between the original file and the copy he edited. This difference is stored in a third file.

5. It uploads to Freechains chain the diff file containing the differences between the the original file and the copy he edited.

Through the Freechains Post command, user B upload the data from the local diff file (diff of update operations between the downloaded file and the edited copy) to the Freechains chain.

T2 - User C:

1. Gets the first item in the chain.

   Through the Freechains Get command, it obtains the first block of content of the chain (the edition of user A) and saves it in a local Automerge file.

2. Gets the second item in the public chain.

   Also through the command Freechains Get, get the second block of content (the diff edition of user B) and saves it in a local Automerge file.

3. Recompose the original file using the first file downloaded from the chain and inserting the differences present in the second downloaded file.

   Through AMPATCH, user C applies the diff file of user B's operations (obtained in the second block of the chain) in the file of user A (obtained in the first block of the chain), starting to have a complete file, with all the content from chain.

4. Makes a copy of the recomposed file.

   Duplicates the file using the Linux cp command.

5. Edits the array created by user A and adds a string value to the zero index of that array.

   Edits in the file will be made through AMRW, all recorded in the result file.

6. Gets the difference between the original file recomposed and the newly edited file.

   Using AMDIFF, user C obtains the difference of operations between the file with the recomposed contents of the chain, and the copy he edited. This difference is stored in a third file.

7. It uploads the new diff file containing the differences to a block on chain.

   Through the Freechains Post command, user C upload the data from the local diff file (diff of Automerge operations between the downloaded file and the edited copy) to the Freechains chain.

T3 - User A:

1. Gets the first block in the public chain.

   Through the Freechains Get command, it obtains the first block of content of the decentralized and permissionless chain (the edition of user A) and saves it in a local Automerge file.

2. Gets the second item in the public chain.

   Through the command Freechains Get, get the second block of content (the diff edition of user B) and saves it in a local Automerge file.

3. Gets the third block in the public chain.

   Also through the command Freechains Get, get the third block of content (the diff edition of user C) and saves it in a local Automerge file.

4. Recompose the original file using the first file downloaded from the chain and inserting the differences present in the second and the third downloaded file.

   Through AMPATCH, user A applies the diff file of user B's operations (obtained in the second block of the chain) in the file of user A (obtained in the first block of the chain), and in sequence applies the diff file of user C's operations (obtained in the third block of the chain) in the file with the data of users A and B, starting to have a complete file, with all the content from chain.

At this point User A has an updated file with all the changes made by other users present on the network. This scenario demonstrates the functioning of our work's proposal (a dataset distributed in a decentralized and permissionless P2P environment) with 3 active users.

Any editing conflict situation, Automerge [16] will handle. Automerge corner cases, like concurrency simultaneous editions, Freechains [14] reputation system will make the blocks containing the edits that have the highest reputation among users prevail.

The update operations (edits) will each be posted in a block on the Freechains chain, these posts are structured in a cryptographic acyclic directed graph that is immune to modifications (Merkle DAG). this will ensure the causal order of the update operations.

Any problem with malicious users, Sybill attacks [9], SPAM and haters, the reputation system of Freechains [14] will cause the blocks containing editions that bring this type of content to be discredited by system users through dislike. And therefore be removed from view for users.

To make the ecosystem proposal more user friendly and facilitate the realization of the proof of concept in the next chapter, we unify the AMDIFF and AMPATCH tools as well as the Freechains [14] post and get commands in a single tool called Freechains-JSON, which we will explain below.

2.6 Freechains-JSON: Automerge-JSON Synchronization for Freechains

The Freechains-JSON relies on AMDIFF, AMPATCH, and the main Freechains commands for publishing and downloading from the network.

This tool allows to do uploads of Automerge file to the Freechains network, enabling to put into practice the concept of permissionless decentralized network-stored data structures with arbitrary datasets.

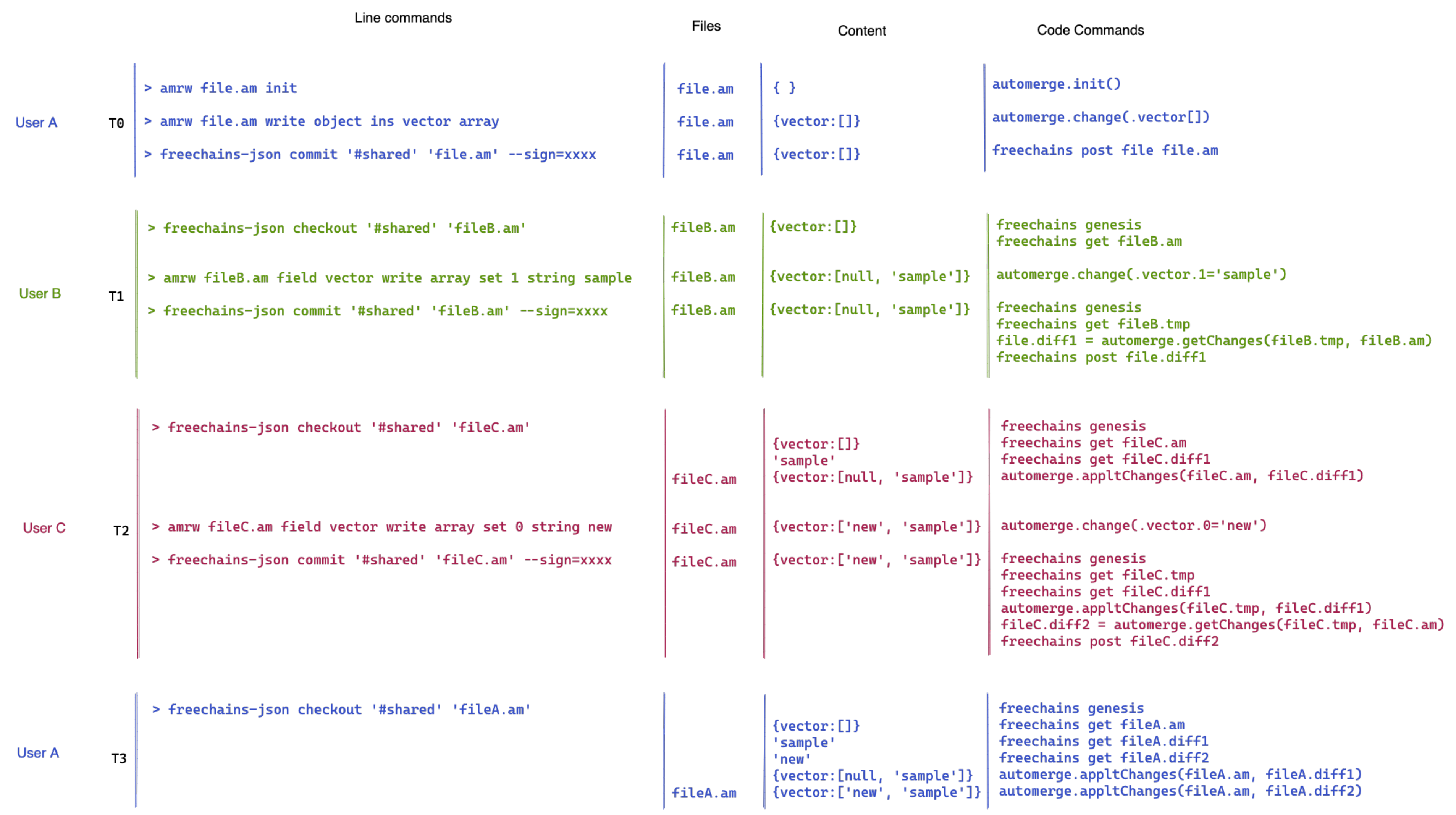
When a user joins the network, it downloads the most up-to-date data structure at that time. At that moment, Freechains-JSON recomposes the entire data structure by joining all the operations that are stored incrementally in the blocks of the Freechains chain.

When that same user makes an edit to this local file, the Freechains-JSON extracts the difference between the user's locally stored data structure and the last network update, and posts only that difference.

Freechains-JSON is written in NodeJS, and is available at
https://github.com/fabiobosisio/freechains-json.

The manual with the complete syntax of Freechains-JSON commands, are available in the appendix of this work.

Going back to the example in Figure 18, with the 3 users, the steps will be much simpler using Freechains-JSON, as shown in Figure 19.



		Line commands	Files	Content	Code Commands
User A	T0	`> amrw file.am init`	`file.am`	`{ }`	`automerge.init()`
		`> amrw file.am write object ins vector array`	`file.am`	`{vector:[]}`	`automerge.change(.vector[])`
		`> freechains-json commit '#shared' 'file.am' --sign=xxxx`	`file.am`	`{vector:[]}`	`freechains post file file.am`

User B	T1	`> freechains-json checkout '#shared' 'fileB.am'`	`fileB.am`	`{vector:[]}`	`freechains genesis`
					`freechains get fileB.am`

		`> amrw fileB.am field vector write array set 1 string sample`	`fileB.am`	`{vector:[null, 'sample']}`	`automerge.change(.vector.1='sample')`

		`> freechains-json commit '#shared' 'fileB.am' --sign=xxxx`	`fileB.am`	`{vector:[null, 'sample']}`	`freechains genesis`
					`freechains get fileB.tmp`
					`file.diff1 = automerge.getChanges(fileB.tmp, fileB.am)`
					`freechains post file.diff1`

User C	T2	`> freechains-json checkout '#shared' 'fileC.am'`	`fileC.am`	`{vector:[]}`
				`'sample'`
				`{vector:[null, 'sample']}`
					`freechains genesis`
					`freechains get fileC.am`
					`freechains get fileC.diff1`
					`automerge.appltChanges(fileC.am, fileC.diff1)`

		`> amrw fileC.am field vector write array set 0 string new`	`fileC.am`	`{vector:['new', 'sample']}`	`automerge.change(.vector.0='new')`

		`> freechains-json commit '#shared' 'fileC.am' --sign=xxxx`	`fileC.am`	`{vector:['new', 'sample']}`	`freechains genesis`
					`freechains get fileC.tmp`
					`freechains get fileC.diff1`
					`automerge.appltChanges(fileC.tmp, fileC.diff1)`
					`fileC.diff2 = automerge.getChanges(fileC.tmp, fileC.am)`
					`freechains post fileC.diff2`

User A	T3	`> freechains-json checkout '#shared' 'fileA.am'`	`fileA.am`	`{vector:[]}`
				`'sample'`
				`'new'`
				`{vector:[null, 'sample']}`
				`{vector:['new', 'sample']}`
					`freechains genesis`
					`freechains get fileA.am`
					`freechains get fileA.diff1`
					`freechains get fileA.diff2`
					`automerge.appltChanges(fileA.am, fileA.diff1)`
					`automerge.appltChanges(fileA.am, fileA.diff2)`

Figure 19 - Execution timeline using Freechains-JSON.

T0 - User A:

1. Initializes an Automerge-JSON file.

   Initializes the CRDT Automerge file that will contain the dataset using the Init command.

2. Includes an array named vector in the created vector.

   Edits in the file will be made through AMRW, all recorded in the result file.

3. Post the created file on a Freechains public chain using Freechains-JSON with commit command.

   The Freechains-JSON Commit command, through the Genesis and Get Freechains commands identifies that the chain is empty and through the Freechains post command, posts the content of the edited file.

T1 - User B:

1. Recompose the file posted by user A using Freechains-JSON with the checkout command.

   The Freechains-JSON Checkout command, through the Genesis and Get Freechains commands recompose the file posted by user A.

2. Edits the array created by user A and adds a string value to the one index of that array.

   Edits in the file will be made through AMRW, all recorded in the result file.

3. Use Freechains-JSON commit to post the edited file on the Freechains chain, at that moment the difference between the original post of user A and the new edited post of user B will be obtained, and only this difference will be posted.

   the Freechains-JSON Commit command, through the Genesis and Get Freechains commands recomposes the file posted by user A, and through AMDIFF obtains the difference between the file edited by user B and the file recomposed with the content that was posted in the chain, this difference is posted in the chain through the command Freechains post.

T2 - User C:

1. Recompose the file posted by user A and applies the difference of user B edition using Freechains-JSON with the checkout command.

   The Freechains-JSON Checkout command, through the Genesis and Get Freechains commands gets the first file posted (by user A), and subsequently gets (also via the Freechains Get command) the difference from user B's edition that was posted later. Using AMPATCH, it recomposes the file containing everything published on the network up to that point.

2. Edits the array created by user A and adds a string value to the 0 index of that array.

   Edits in the file will be made through AMRW, all recorded in the result file.

3. Use Freechains-JSON to post the edited file in the Freechains chain, at that moment the difference between the post recomposed with the editions of users A and B and the new edited post of user C will be obtained and only this difference will be posted.

   The Freechains-JSON Commit command, through the Genesis and Get Freechains commands, gets the first file posted (by user A), and subsequently gets (also via the Freechains Get command) the difference from user B's edition that was posted later. Using AMPATCH, it recomposes the file containing everything published on the network up to that point. In sequence through AMDIFF obtains the difference between the file edited by user C and the file recomposed with the content that was posted in the chain, this difference is posted in the chain through the command Freechains post.

T3 - User A:

1. Recompose the file posted by user A and applies the difference of user B and user C editions using Freechains-JSON with the checkout command.

   The Freechains-JSON Checkout command, through the Genesis and Get Freechains commands, gets the first file posted (by user A), and subsequently gets (also via the Freechains Get command) the difference from user B's edition that was posted later. Using AMPATCH, it recomposes the file containing users A and B content. In sequence gets (also via the Freechains Get command) the difference from user C's edition that was last posted. Using AMPATCH, it recomposes the file containing everything published on the network up to that point

At this point User A has an updated file with all the changes made by other users present on the network.

In this chapter we put into practice the techniques and theories detailed in the previous chapter through the development of practical tools. We also detail the interaction between these tools and the permissionless P2P network Freechains, developing an ecosystem that demonstrates a permissionless peer-to-peer scenario with JSON Datasets.

3 PROOF OF CONCEPT: A PERMISSIONLESS WIKIPEDIA

This chapter presents a proof of concept of the ecosystem proposed in Chapter 2. We prototype a permissionless P2P Wikipedia, in which fictional characters interact with it. We use a step-by-step interactive approach to show a concrete use of our ecosystem.

The Lamport space on a permissionless P2P Wikipedia:

Leslie Lamport is a distinguished computer scientist, who decided to create an article about distributed systems open for his students and anyone else interested in the topic. However, he wants an article that grows organically, that is, without a central authority, allowing new users to join and edit the article.

At first, Lamport provides a brief introduction of Distributed systems, and in sequence, a list of typical distributed applications. His goal is to open a collaborative debate on these uses. Initially, he intends to share a JSON to represent the article as follows:

```
{
    Sections: {
        Introduction: 'Distributed systems are ...',
        Applications: []
    }
}
```

- The first step is to create an identity on Freechains:

  ```
  // Create an identity:
  $ freechains keys pubpvt "lamport secret password"
  96700ACD1... A392D7FA1...   # (public/private keys)
  ```

  This command creates a pair of public/private keys to sign and encrypt posts. Note that this identity is personal, and is not linked to the device he is using. Therefore, he can edit on any device he has access to using his personal identity.

- He then joins a public forum representing the article as a pioneer:

  ```
  $ freechains chains join '#distributed-systems' '96700ACD1...'
  ```

At the end of the command, Lamport uses his public key as the forum pioneer (96700ACD1...), generated in the previous step.

As discussed in Section 1.4.4, the public forums in Freechains allow users to join a chain without the need for centralized approval. This enables the growth of the forum to be organic, and also allows all participants to have equal possibilities within the forum, such as for posting, liking, disliking, or acting as a content moderator.

- Lamport first writes the intro for the article:

  He creates the local file with a minimum JSON structure, inserts an empty object called Sections and finally writes the introduction.

  ```
  $ amrw distributed-systems init
  $ amrw distributed-systems write object ins "Sections" object
  $ amrw distributed-systems field "Sections" write object ins
      "Introduction" string "Distributed systems are..."
  ```

  AMRW is used to initialize and edit the local-first CmRDT file that created by Lamport. Note that only the update operations are registered in this file.

- Lamport decides to see the current state of his local page:

  ```
  // Check the local-first CmRDT file content
  $ amrw distributed-systems read
  {
      Sections: {
          Introduction: 'Distributed systems are ...'
      }
  }
  ```

  Note that the content of the local file only contains the introduction, included so far. Following the initial objective, in the next step Lamport will include the list of applications.

- Continuing, Lamport now creates the empty applications list:

```
// Create an empty list called Applications inside the object Section
$ amrw distributed-systems field "Sections" write object ins
"Applications" array
```

At this moment he has in his local file the object he initially intended:

```
{
    Sections: {
        Introduction: "Distributed systems are ...",
        Applications: []
    }
}
```

- Therefore, Lamport decides to pause editing, and commits his local changes on the public forum, and also synchronizes its local contents with the network:

```
// Post what he did locally on the public forum.
$ freechains-json --host=serverhost:8330 commit #distributed-systems
    distributed-systems --sign=003030E0D03030D...

// Synchronizes a chain with a given peer.
freechains peer 'remotehost' send '#distributed-systems'
```

As he is the first user to post, at this point we have exactly the same content he has in his local CmRDT file on the public forum.

- After synchronizing its local content with the network, Lamport decides to make a new change in its local replica, including the first application in the "Applications" array:

```
// Inserts the word "Ethereum" in the Application list
$ amrw distributed-systems field "Sections" field "Applications"
write array ins 0 string "Ethereum"
```

So far, Lamport has the following local JSON object inside the CmRDT local file distributed-systems.am:

```
{
    "Sections": {
        "Introduction": "Distributed systems are ...",
        "Applications": [ {"Ethereum"} ]
    }
}
```

And the public forum has the following object stored:

```
{
    "Sections": {
        "Introduction": "Distributed systems are ...",
        "Applications": []
    },
}
```

When the news of Lamport's collaborative page became known in the scientific and academic circles, several researchers on the topic and students were interested in participating. To encourage initial membership Lamport distributes reputation to a few closest students, Dave and Jake in this group. Dave is a master student who decided to collaborate with the page:

```
// Dave starts a host:
$ freechains-host start /home/pi/servers/distsys

// Dave creates an identity:
$ freechains keys pubpvt "Dave secret password" # creates
a pair of public/private keys to sign and encrypt posts

// Dave creates and joins the public forum #distributed-systems:
$ freechains chains join '#distributed-systems' '96700ACD1...'
#Lamport's pioneer key
```

- Dave do the checkout of the actual content of the public forum:

```
// Checking out public forum #distributed-systems to local
distributed-systems.am
$ freechains-json --host=serverhost:8330 checkout #distributed-systems
distributed-systems
```

A checkout recreates in Dave's local disk the latest updated version of the file, by applying all patches since the genesis block. The genesis block has the first element created, and each patch represents update operations from users, to step by step rebuild the final state of the JSON object.

- Dave writes his first contribution to an application in the Applications list, by coincidence the same choice as Lamport, but he unknowingly misspells the application name:

```
// Creates an string with the word "Blokchain", on the index
(position) 0 of the Application list
$ amrw distributed-systems field "Sections" field "Applications"
write array ins 0 string "Blokchain"
```

So far, Dave has the following local JSON object inside the CmRDT local file distributed-systems.am:

```
{
    "Sections": {
        "Introduction": "Distributed systems are ..."
        "Applications": [ {"Blokchain"} ]
    }
}
```

- Dave (less reputation) persists (commits) his changes made locally on the public forum, and synchronizing its local content with the network:

```
// Post what he did locally on the public forum.
$ freechains-json --host=serverhost:8330 commit #distributed-systems
distributed-systems --sign=93875E0D04056D...
```

```
// Synchronizes a chain with a given peer.
freechains peer 'remotehost' send '#distributed-systems'
```

- Lamport (more reputation) persists (commits) his new changes made locally on the public forum, and also synchronizing its local content with the network:

```
// Post what he did locally on the public forum.
$ freechains-json --host=serverhost:8330 commit #distributed-systems
distributed-systems --sign=003030E0D03030D...

// Synchronizes a chain with a given peer.
freechains peer 'remotehost' send '#distributed-systems'
```

A freechains-json commit command first makes a checkout from the repository into a local temporary file. Then, it compares this version against Dave's local changes, saving the diffs into file "changes.am. This changes file contains the set of update operations which correspond to the edits made by Dave. Finally, the commit command posts the "changes.am" file to the chain.

At this moment, we have a conflicting situation in which two authors edit and commit the same item of the file concurrently. After they commit the conflicting changes, the users synchronize in both directions. If we checkout the file, the changes are applied respecting the Freechains consensus order. As a result, we see that the Lamport change is applied, but not the Daves one, leaving the file in the longest possible consistent state. This happens because of the break in the checkout operation: once a conflict is found, no further patches apply in any of the remaining branches. Freechains chose to adopt a first write wins resolution to favor work in the changes of the user with more reputation, as shown in Figure 20

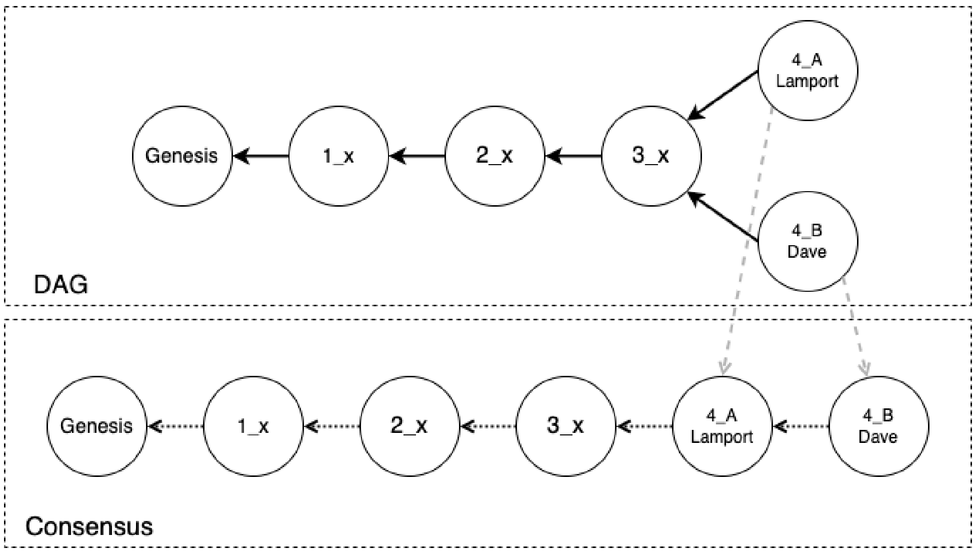


Figure 20 - The branches in the DAG are ordered by reputation. Only the first patch is applied successfully (first write wins).

Nevertheless, the failing patch branch is not totally ignored, since the checkout saves the conflict file and indicates the block causing it. We believe this optimistic choice that does not reject both patches is the most advantageous, since it keeps the file in an usable state and warns about the conflict to resolve. For instance, the authors can later decide to dislike one of the two commits to revoke it and remove the warning.

So at this moment we have the follow object in the public forum:

```
{
    "Sections": {
        "Introduction": "Distributed systems are ..."
        "Applications": [ {"Ethereum"} ]
    },
}
```

Jake is another student in the group who initially received reputation, but he is not a fan of distributed systems, and seeing the topic, decides to make a malicious post:

```
// Jake starts a host:
$ freechains-host start /home/pi/servers/distsys
```

```
// Jake creates an identity:
$ freechains keys pubpvt "Jake secret password" # creates
a pair of public/private keys to sign and encrypt posts

// Jake creates and joins the public forum #distributed-systems:
$ freechains chains join '#distributed-systems' '96700ACD1...' # type the
pioneer public key above
```

- Jake do the checkout of the actual content of the public forum:

```
// Checking out the content of public forum #distributed-systems
to his local distributed-systems.am file
$ freechains-json --host=serverhost:8330 checkout #distributed-systems
distributed-systems
```

- Jake writes his hate post:

```
// Writing the post
$ amrw distributed-systems field "Sections" write object ins "RealTrue"
string "Distributed systems don't scale!!!"
```

- Jake persists (commits) his changes made locally on the public forum, and synchronizing its local content with the network:

```
// Post what he did locally on the public forum.
$ freechains-json --host=serverhost:8330 commit #distributed-systems
distributed-systems --sign=60905E8D90734D...

// Synchronizes a chain with a given peer.
freechains peer 'remotehost' send '#distributed-systems'
```

At this point, we have the follow object in the public forum:

```
{
    "Sections": {
        "Introduction": "Distributed systems are ..."
        "RealTrue": "Distributed systems don’t scale!!!"
        "Applications": [ {"Ethereum"} ]
    },
}
```

- Lamport and Dave and other network users, when checking out the chains to follow interacting, detect the attack and immediately dislike the post, until the block of the post becomes revoked in the chain of the public forum:

```
// Lamport deslike the post
$ freechains chain "#distributed-systems" dislike 3_AE3A1B...
--sign=003030E0D...

// Dave dislike the post
$ freechains chain "#distributed-systems" dislike 3_AE3A1B...
--sign=93875E0D0...
```

And both of them commit their dislikes on the public forum, and synchronizing its local content with the network.

Doing at this moment the checkout operation, its will apply an empty patch associated with the revoked block of the post, effectively removing the line of the hate post from the file. This mechanism illustrates how the Freechains reputation system enables collaborative permissionless edition and curation:

```
{
    "Sections": {
        "Introduction": "Distributed systems are ..."
        "Applications": [ {"Ethereum"} ]
    },
}
```

This proof of concept demonstrated the functioning of a public forum, stored in a permissionless P2P network with data stored in datasets encapsulated in a CmRDT

JSON file. Conflict situations were demonstrated, which will be handled by the Freechains consensus system, which privileges users with the highest reputation in order to maintain the network's intention, since their reputation comes from the likes of others, but which still allows this decision to be reversed manually. We also demonstrate a possibility of malicious use, where a user posts hate, and other users can moderate this content by blocking it from the network. With this we show our proposal of this work in practical operation.

4 RELATED WORK

Many other solutions have been proposed for collaborative networked applications. In this chapter, we discuss federated and peer-to-peer protocols, decentralized wikis, and CRDT-based applications. We also compare them with our proposal in terms of level of decentralization, and data consistency, accuracy and integrity.

4.1 Decentralized Network Protocols

As discussed in Section 1.4, computation systems can be classified into three broad categories: centralized, federated, and peer-to-peer (P2P) systems. Now, we choose to compare two types of protocols: Federated and Peer-to-Peer. We chose to direct our comparisons only to these two models, because they have some level of decentralization, which is the focus of our work.

4.1.1 Federated Protocols

Federated protocols, such as e-mail, allow users from one domain to exchange messages with users of other domains seamlessly. More recently, other examples of federations emerged, such as Diaspora [66] and ActivityPub [67] for social media and Matrix [68] for chats.

As discussed in Section 1.4, federated protocols have some level of decentralization but still follow the client server architecture, in which multiple servers that are responsible for different services or resources, and in which clients make requests to servers, which respond to those requests and provide the necessary services or resources.

- **Diaspora [66]** is a decentralized social network, based on decentralized nodes or "pods", which are servers operated by individuals or organizations. Each node is responsible for storing and managing its users' data, and can communicate with other nodes to allow its users to interact with users on other nodes. The decentralization of the Diaspora network reduces reliance on a single controlling entity, giving users more control over their own data. Furthermore, the Diaspora network is more resistant to attacks and outages, as there is no central point of failure. Finally, the

Diaspora network allows users to connect with people in different "pods", which increases the diversity and plurality of points of view on the network.

- **ActivityPub [67]** is a decentralized social networking protocol that allows users to create and share content across different social networking platforms. It is designed to enable federation between different instances of the protocol, allowing users to connect and interact with each other regardless of the specific platform they are using.

  ActivityPub allows for a distributed network of servers and platforms that are owned and operated by different individuals and organizations. In the ActivityPub network, each server is called an "instance", and users can choose to create accounts on any instance that they wish. These instances can then communicate with each other through the ActivityPub protocol, allowing users to share content, follow other users, and participate in discussions across different platforms. Overall, the decentralization features of ActivityPub make it a powerful tool for creating a more open, democratic, and resilient online ecosystem.

- **Matrix [68]** is a decentralized communication protocol that emphasizes user control and interoperability. It allows users to communicate with each other through different messaging services and devices, while maintaining ownership and control of their data and content. The Matrix protocol achieves decentralization by using a distributed network of homeservers, which can be operated by individuals, organizations, or service providers. Users can create accounts on any homeserver, and these homeservers can communicate with each other using the Matrix protocol. This enables users to communicate with each other, regardless of the service or device they are using.

  One of the key advantages of the Matrix protocol's decentralized architecture is that it provides users with greater control over their data and content. Since each user has their own account on their preferred homeserver, they have full ownership and control over their data and content. This also makes it more difficult for centralized authorities to monitor or censor communication on the network. Another advantage of the Matrix protocol is its emphasis on interoperability. Since the protocol can be used by different messaging services and devices, it helps to prevent the fragmentation of the messaging ecosystem. This means that users are not locked into a single messaging service or device, and can communicate with each other regardless of their preferences.

Considerations

We can verify in all the examples that the federated protocols have some level of decentralization, with power distributed between servers. However, they still enforce some hierarchy, with some servers playing more important roles than others users. Our proposal for a decentralized network has a flatter and more distributed structure, without servers.

One of the challenges with federated protocols is maintaining data consistency across different nodes or servers in the network. Since each node or server is responsible for its own data and content, there is a risk that data may become inconsistent or out of sync as it is replicated or shared across different nodes.

To address this challenge, federated protocols use a variety of strategies to ensure data consistency. For example, some federated protocols use a "federation" model [69], where data is replicated across different nodes in the network. Each node retains a copy of the data and periodically synchronizes with other nodes to ensure that the data is consistent and up-to-date. Other federated protocols use a "sharding" model [69], where data is divided into smaller subsets that are stored and processed separately on different nodes in the network. This approach helps to distribute the workload and reduce the risk of data inconsistencies, since each node is responsible for a smaller subset of the data. Despite these strategies, ensuring data consistency in federated protocols remains a complex and ongoing challenge. As the number of nodes and users in the network grows, the risk of data inconsistencies and conflicts increases. Without coordinated consensus the system is subject to inconsistencies derived from competing editions, harming the consistency and accuracy of data. We propose in our work the use of CRDT to maintain data consistency and a reputation system based on likes and dislikes to resolve CRDT corner cases, such as concurrent editions of the same item, while maintaining data accuracy.

Another potential challenge of federated systems is the data integrity. In a federated system, each instance is independently operated and managed, and there is no centralized authority that is responsible for moderation or enforcing rules across all instances. This means that if a user engages in malicious behavior, such as spamming or hate speech, on one instance, they may not necessarily be identified as a threat in other instances. The lack of unified moderation can make it more difficult to identify and address these types of issues across the entire federated network.

To address this challenge, many federated systems implement various forms of moderation and community standards within each instance. For example, Mastodon [70] allows individual instances to set their own community guidelines and moderation policies, which can help to promote responsible behavior and prevent abuse within that instance.

However, this approach can also lead to fragmentation and inconsistency in moderation policies and practices across different instances, which can make it more difficult to maintain overall network integrity. As such, maintaining network integrity and preventing abuse in federated systems can be a complex and ongoing challenge that requires ongoing coordination and communication between instance operators and users. In our work, users through the reputation system when identifying some malicious behavior, SPAM, haters or anything that hurts the integrity of the data, can block this content through the collective dislike, maintaining the integrity and protection of the network

4.1.2 Peer-to-Peer Systems

As discussed in section 1.4, peer-to-peer(P2P) protocols have gained popularity as a mechanism for users to share content without the need for centralized servers [71]. In a P2P network, each device is equally capable of sharing resources and information directly with other devices. This allows for a variety of applications beyond simple file sharing [71]. The P2P protocol is more decentralized and less susceptible to a single point of failure as can occur with a federated protocol.

Bitcoin [13] is probably the most successful permissionless network, but serves specifically for electronic cash. Other examples of permissioned systems are IPFS [72] and Dat [73], that are data-centric protocols for hosting large files and applications, respectively, Aether [46] and Scuttlebutt [74] for public communities. Some works consider Scuttlebutt a hybrid, because some communities may have their own rules and guidelines for participation, which could be considered permissioned in a sense, but the overall architecture of Scuttlebutt is still designed to be decentralized and open, with no central authority or gatekeepers controlling access to the network, therefore the system as a whole is still largely permissionless.

- **Bitcoin [13]**. The consensus is achieved through a proof-of-work algorithm that requires nodes in the network to perform computationally intensive calculations in order to validate transactions and add them to the blockchain. This solution keeps data integrity, but does not solve the centralization issue entirely, given the high costs of equipment and energy. Proof-of-stake is a prominent alternative [75] that acknowledges that centralization is inevitable, and thus uses a function of time and wealth to elect peers to mint new blocks. As an advantage, these proof mechanisms are generic and apply to multiple domains, since they depend on an extrinsic scarce resource.

- **IPFS [72]** is centered around immutable content addressed data, while Dat [73] around mutable pubkey addressed data. IPFS is more suitable to share large and stable content such as movies and archives, while Dat is more suitable for dynamic content such as web apps. Both IPFS and Dat use DHTs as their underlying architectures, in both cases, users need to know in advance what they want, such as the exact link to a movie or a particular identity in the network. In our work, all nodes have a complete replica of the network, therefore we don't rely on DHTs.

- **Aether [46]** provides peer-to-peer public communities aligned with public forums that we are proposing in our P2P Wikipedia. A fundamental difference is that Aether is designed for ephemeral, mutable posts with no intention to enforce global consensus across peers, or keep the accuracy of the posts, since it does not considers the users intent. About data integrity, Aether employs a very pragmatic approach to mitigate abuse in forums. It uses established techniques, such as proof-of-work to combat SPAM, and an innovative voting system to moderate forums, but which affects local instances only.

- **Scuttlebutt [74]** is designed around public identities that follow each other to form a graph of connections. This graph is replicated in the network topology as well as in data storage. For instance, if identity A follows identity B, it means that the computer of A connects to B's in a few hops and also that it stores all of his posts locally. For public forum communications, like our P2P Wikipedia, Scuttlebutt uses the concept of channels, which are in fact nothing more than hash tags (e.g. #sports). Authors can tag posts, which appear not only in their feeds but also in local virtual feeds representing these channels. However, users only see channel posts from authors they already follow. In practice, channels simply merge friends posts and filter them by tags. In theory, to read all posts of a channel, a user would need to follow all users in the network (which also implies storing their feeds). A limitation of this model is that new users struggle to integrate in channel communities because their posts have no visibility at all. As a counterpoint, channels are safe places that do not suffer from abuse, maintaining data integrity. Scuttlebutt offers single-user/multi-node consensus, in which a single user has full authority over its own content across machines, but multiple users cannot reach consensus even in a local machine. Our work goal is to provide multi-user/multi-node consensus while keeping the consistency in the context of decentralized content sharing. Consensus together with the reputation system based on proof-of-authoring maintains the accuracy of the data.

Considerations

To ensure data consistency and accuracy in P2P networks, consensus may be achieved through a variety of mechanisms, depending on the specific architecture and goals of the network. Some P2P networks use proof-of-work (PoW) [76] or proof-of-stake (PoS) [76] mechanisms to achieve consensus, while others use proof-of-activity (PoA) [76], proof-of-elapsed time (PoET) [76] , practical byzantine fault tolerance (PBFT) [76] and delegated proof-of-stake (DPoS) [76] .

Maintaining data integrity is also a key point and can be hard to manage precisely because of the lack of centralized control. Identifying malicious users in an anonymous network, the difficulties of enforcing policies uniformly across a decentralized network, and the resilience of P2P networks to censorship and disruption, are some of the challenges faced.

To deal with these challenges, in our work in a non-permissioned P2P environment using JSON datasets, materialized in a P2P Wikipedia, we used Automerge CRDTs to achieve consensus between user editions, and for CRDT corner cases, such as simultaneous editions of the same item, we use the Freechains reputation system based on proof-of-authoring (using likes and dislikes to keep or block a certain post with update operations) to maintain the accuracy of the data, considering the intent of the users. This same reputation system will allow users to block malicious posts, haters, SPAM and Sybil attacks, maintaining the integrity of the network.

4.2 Decentralized Wikis

There are some examples of decentralized wikis. We present 3 examples in this section and discuss how they deal with the challenges mentioned at the beginning of this chapter.

- **Everipedia [77]** was a decentralized, wiki-based online encyclopedia that allows anyone to create and edit articles, but with some conditions. To contribute to Everipedia, a user must provide the sources for every piece of information they include in an article, and the article must be approved through voting from the community of users.

  One of the key features of Everipedia is that it uses blockchain technology to ensure the accuracy and transparency of its content. Every edit or change made to an article is recorded on the blockchain, which allows users to trace the history of an article and verify its accuracy. In addition, the platform uses a system of "IQ" tokens rewarded to incentives users to contribute high-quality content and reward them for their contributions.

  The Everipedia allows everyone to participate in the process of approving/rejecting items. If someone has even the smallest amount of IQ tokens, he can directly influence the events within Everipedia. When voting, it is necessary to invest an amount of IQ tokens. Investing a token amount proportional to the degree of support for a particular item is recommended. With each vote, it is possible to leave a comment in which it is possible to explain the position on accepting or rejecting the item.

  "The open-source nature of the protocol means that almost anyone can have a say. As a result, content on the platform may sometimes be biased. Critics have pointed out that the platform may sometimes fail to keep the users' intent and promote content from an unbalanced perspective" [78].

  One example of this criticism was in in October 2017, an Everipedia editor misinformed the public about a mass shooting in Las Vegas [79]. The editor identified the perpetrator of the shooting as Geary Danley, while Geary was innocent. While this got to the notice of Everipedia on time, and the post was brought down within 10 minutes from the time it was shared, some damage had already been done. Numerous social media users had spread false information, tarnishing the name of Geary.

The protocol has since then exercised even more caution before approving content, counting with a group of editors who review activity on the site and content they deem to be sketchy is deleted [80]. That puts the propose of collective moderation at risk. While the content itself cannot be removed from the blockchain, it can be removed from the user interface, which means that it is no longer visible to users. This is achieved through a process called "soft deletion", where the content is removed from the user interface, but remains in the blockchain for historical record-keeping purposes. While the information will still technically be on the blockchain, it will no longer be visible to users on the Everipedia platform, and it will not be included in search results or used to generate suggestions for related content.

Therefore, we realize that more and more to maintain data integrity, Everipedia seeks a permissioned approach, through a group of content moderators that, according to its founders "scrolls through every activity on the site, and if something is sketchy , we take it down" [80], through the "soft delete" feature maintaining the integrity of the blockchain. Our proposal is based on a permissionless user-based reputation mechanism to self moderate the network, without the need for specific moderation groups, and preserving data accuracy and integrity.

- **DSMW [81] [82] Distributed Semantic MediaWiki**

  This semantic wiki implements a multi-synchronous collaboration model [83], closely related to the DVCS idea. Participating peers host a full wiki, and subscribe to feeds of updates from other peers. The feeds are then automatically merged with the Logoot algorithm [84], that ensures consistency and data accuracy with intention preservation. However, the consistency guarantees restrict the possibilities of divergence. The multi-synchronous workflow is intended to lead to eventual consistency, and although a participant can choose not to integrate a set of changes from another participant, this will prevent it from integrating any changes made later on, virtually disconnecting the two versions.

Our work, a permissionless distributed wiki proposes, the accuracy of the data is achieved through consistency based on CRDT techniques, wheres the decision between concurrent editions is done automatically, except in cases of competing editions of the same item, in which case we use a user-based moderation system to maintain data accuracy. Even if the others versions are not absolutely discarded and are available in case users' intentions change.

To maintain data integrity, i.e. prevent malicious users, haters and spam, the DSMW recommends permissioned approaches such as access control, capcha verification, user access restriction, data moderation through moderators and community guidelines. Our proposal is a permissionless system, in which, through the user-based reputation system, the users themselves blocks any harmful content.

- **DistriWiki [85], a Distributed Peer-to-Peer Wiki**

  In this solution, each user's computer acts as a peer that stores redundant copies of wiki pages. In this way, it can reduce the number of failures due to hardware and configuration errors, and avoid centralized organizational control of the wiki pages. In order to have reliability in case a peer goes down, there are redundant data across peers, therefore no single network or hardware failure cause the system to become unavailable. Network access is easy and reliable, and the failure of a peer not affect the network. Unlike in a client-server system, the search function must span different hosts, since, in the case of this system, no single host will have a complete set of pages.

  Because documents can be simultaneously modified, there is a method of uniquely identifying documents. If two people produce version X of the same document, how does the network distinguish between the two? There is no central server, there is no way to prevent concurrent versioning. As a result, all documents have a unique, global identifier and the software notify the user of such conflicts when possible.

  They are dealing with a decentralized system, so synchronization issues may arise. Once different documents may have the same title or version, each document is given its own unique ID (randomly generated), and every document is given a time stamp. It is then up to the user to handle any conflicts. That's an important point, when it comes to data accuracy, the management of direct conflicts is handled manually by

DistriWiki users, therefore it is subject to individual desires, which do not always respect the collective intent of the network. It does not have a mechanism where all users participate in the conflict management decision-making process. In the same way that it does not have any treatment to prevent attacks, malicious users and haters, to guaranteeing the integrity of the data.

Our proposal is based on consensus, where all concurrences converge in a consistent and accurate version. As previously mentioned, this is done through the use of CRDT techniques in addition to a user-based reputation system. This same relay system is used to protect data integrity from the point of view of malicious users, SPAM, haters and Sybil [9] attacks.

Considerations

The challenges faced by decentralized wikis include the need to ensure consensus in a distributed network, the management of editing conflicts and content moderation (and some options decided, by concept, to give up uniting competing and divergent contents). In addition, decentralization can also make detecting and removing inappropriate content more difficult, as there is no single control point.

In our proposal for a permissionless P2P Wikipedia, we use Automerge CRDTs to manage conflicting editions and we use the consensus system together with the Freechains user-based reputation system, so that the chain participants themselves handle cases of concurrent edition of the same item, keeping the data accuracy and the users intention. Through this same reputation mechanism, the users themselves block editions of malicious users, haters and abuse, ensuring the integrity of data. All this without the need for a central authority. In addition, still about data integrity, we also uses the Freechains consensus mechanism to prevent SPAM and abusive behavior from byzantine nodes, which could otherwise generate very large graphs that take forever to synchronize.

4.3 CRDT-Based Collaborative Systems and Applications

In this section we talk about content systems and collaborative applications that use CRDT, and how they deal with the challenges mentioned at the beginning of this chapter.

- **Trello [86]**

Is a project management tool that uses CRDTs to allow multiple people to work together on a project. The system keeps a real-time copy of the project on each machine, and changes are synchronized in real-time between users. Trello uses CRDTs to maintain the consistecy of the data and ensuring that changes are being done in a decentralized manner.

Trello uses CRDTs to enable real-time collaboration among multiple users while maintaining data accuracy. But as mentioned in section 1.3, CRDTs have corner cases, when changes are performed concurrently on the same item, it sometimes makes arbitrary decisions (depending on the implemented algorithm) that do not always preserve the users intention.

In our work, we use the Freechains user-based reputation system to decides witch changes will prevail, where this decision is considering the intention of users, keeping the data acurracy.

To maintain data integrity, Trello uses permissioned features such as user authentication (requires users to authenticate before they can access a task board), authorization verification (verifies whether users have permission to access and modify the content of a task board before allowing changes to be made), and activity monitoring (monitors activity on a task board to detect suspicious or malicious activities). So, it is a permissioned system. This goes against our proposal of a permissionless self-moderated system.

- **Figma [87]**

  Figma is a collaborative graphic design application that allows multiple users to work on the same design file simultaneously, by the creation and editing of graphic elements such as icons, page layouts, and charts, as well as allowing comments and feedback. For that, Figma uses CRDTs to ensure that all changes made by each user are properly synchronized, keeping the data consistency. The synchronization is done in a decentralized manner, and CRDTs handle concurrent changes made by multiple users by creating a conflict-free history of all changes, which can then be merged together in a deterministic way. This ensures that all users see the same version of the design file, regardless of the order in which changes were made.

  As Figma uses CRDTs, they have corners cases with simultaneous editing of the same item, to deal with this, the strategy adopted is allows users to comment on specific parts of the design file, to help to manually resolve conflicts and reach consensus among team members.

About data integrity, Figma has features and policies in place to help prevent malicious use of its platform and promote a collaborative and positive environment. As an example, Figma has a code of conduct that all users must follow, which includes guidelines for appropriate behavior and prohibits hate speech, harassment, and other forms of malicious or inappropriate behavior. Figma allows team administrators to set permissions for each user and restrict access to sensitive information as needed. Has a reporting system that allows users to report any inappropriate behavior or misuse of the platform.

In summary, the corner cases of simultaneous editing conflict are resolved manually, without a policy or mechanism that automates this and keeps the data accuracy. From the point of view of data integrity it is a permissioned system, where the administrator defines permissions and restrict access to sensitive content. This also goes against our proposal of a permissionless self-moderated system.

- **Quip [88]**

  Quip is a collaborative application that allows you to create documents, spreadsheets, and presentations as a team. It uses CRDTs to allow multiple people to work on the same document simultaneously. Synchronization is done in a decentralized manner, and CRDTs ensure data consistency.

  Quip uses a version control system based on CRDTs, which allows multiple users to work on the same document simultaneously without conflicts or data loss. This means that all users are working on the same version of the document at all times, and changes are automatically synchronized in real-time.

  To ensure the data integrity, prevent malicious behavior and hate speech, Quip has community guidelines and policies in place that outline acceptable behavior and content. Users who violate these guidelines may be subject to warnings, temporary suspensions, or permanent bans. In addition, Quip's moderation team actively monitors content and takes action against users who violate the guidelines.

  Just like the previous example, Quip uses a permissioned approach, through community guidelines and policies controlled by an administrator to deal with malicious users and harmful content, in order to maintain the integrity of the platform's data.

Regarding the accuracy of the data and maintenance of the users' intention, Quip only uses the CRDT as resources and will probably have cases of corner cases, where simultaneous editions of the same item occur, which will be treated in an arbitrary way, which will certainly not guarantee the intention most network users, and the accuracy of data.

Considerations

In all of these examples, synchronization is done in a decentralized way, without relying on a central server to coordinate changes. Each user keeps a copy of the document on their own machine, and changes are propagated to other users in real-time. To ensure data accuracy, specific synchronization algorithms and conflict resolution mechanisms are used, although they are subject to the corner cases of simultaneous editions of the same item, and not all of them deal with this use case, jeopardizing the guarantee of the users' intention. They make use of administrators to ensure the data integrity, through rules of conduct and moderator entities to protect against haters and malicious users.

Our proposal for a Permissionless Peer-to-Peer JSON Datasets system completely decentralizes any type of moderation, making the network, through its participants and the use of the reputation, keeping the data integrity, from the point of view of haters, SPAM, malicious users and Sybil attacks. This same reputation resource together with CRDT techniques is also used to ensure the data consistency and accuracy, that in the event of competition between editions of the same item, the most liked item on the network prevails, maintaining the intention of the majority of users.

CONCLUSION

This dissertation proposes a permissionless P2P system to manipulate decentralized JSON datasets, in which participation and consensus is not controlled by a central authority. Unlike centralized and federated systems, users can join the network without the need for authorization from a central administrator. In addition, all users can validate the datasets and participate in a consensus mechanism to preserve data consistency, accuracy and integrity.

In a permissionless context, a major challenge is to support concurrent data edition and moderation, such that users share consistent and correct datasets even in the presence of Sybils [9].

Our main contribution is to reconcile CRDTs and permissionless P2P systems, such that decentralized collaborative applications support consistency and correctness. More specifically, we propose to reconcile Automerge [16] and Freechains [14], extending JSON CRDTs with a consensus mechanism that can handle consistency corner cases and also moderate malicious editions: Automerge editions are ordered by the Freechains consensus mechanism providing a priority to solve conflicts. In addition, further moderation can revert priorities or even remove abusive or malicious editions.

To demonstrate the practicability of our work, we conceptualize a P2P Wikipedia using Freechains and Automerge. As Automerge is operated through code, we created 3 tools to make our proof of concept more pratical: AMRW, to allow command line editing of a CmRDT, AMDIFF and AMPATCH, which are diff and patch tools specific for CmRDTs.

The use of Automerge's JSON model allows to represent complex data structures, such as nested objects and arrays. Therefore, Automerge allows the user to create most of the structures present in collaborative applications such as Wikipedia, with entries, articles, topics and discourse.

We noticed that the use of local-first CmRDTs, makes the system independent of stable connections, because it allows all edits to be done local-first, and the update operations recorded in the local CmRDT to be synchronized asynchronously over the network. This updates operations corresponding to the differences between what is present on the network and what is in the user's local replica to be sent.

All the content stored locally transmitted to the network is compressed in binary, by reducing the occupation of the stored CmRDT on disk. Considering that each user has a local replica of entire Wikipedia articles, this content can grow considerably, which is why compression is interesting. Also reducing the content to be synced, allowing asynchronous and sporadic synchronization.

As we are in a permissionless environment, that is, without a central authority that moderates the content, we are subject to malicious users who may abuse the system by vandalizing datasets with SPAM or hate speech. Besides the fact that we are using a CRDT structure, we are subject to corner cases as simultaneous concurrent editing of the same item. In order to maintain the integrity of network content (without spam and illicit) and also the accuracy (solving the simultaneous editions preserving the user intention), we use the Freechains reputation mechanism that moderates content through likes and dislikes. In other words, illicit editions will be discredited by users and therefore blocked from the network. Also, in competing editions, the post with the most likes will prevail, which means that it is the most interesting for most network users.

In summary, observing the P2P Wikipedia user case, we came to the following conclusions: We have an organic growth of the network through the adhesion of new users without the need for authorization, or any entity that approves the entry. Regarding consistency and correctness, we can observe that the user based reputation system together with the CRDT techniques, makes an ecosystem that ensures consistency, integrity and accuracy of datasets, even being in a permissionless environment subject to Sybil attacks.

APPENDIX

Installations and basic operations of the tools developed and used in this work.

1 - AMRW: Automerge-JSON command line editor

Installation

1. Install NPM:

   ```
   $ sudo apt install npm
   ```

2. Install NodeJS:

   ```
   $ sudo apt install nodejs
   ```

3. Install Automerge:

   ```
   $ npm i automerge@1.0.1-preview.7
   ```

4. Clone the amrw repository to use the tool::

   ```
   $ git clone https://github.com/fabiobosisio/amrw.git
   ```

5. Use the editor inside the /home/user/amrw directory

Basic commands

The amrw.js commands are as follows:

Usage:

```
node amrw.js <file> init [verbose]
node amrw.js <file> <path>... <mode> <op> [verbose]
node amrw.js <file> json [verbose]
```

node amrw.js help

Help:

Displays this manual

Verbose:

Enable verbose mode

File:

Filename

Init:

Initializes a minimal JSON-Autmerge file

Json:

Convert an Automerge file to a JSON file and save to disk

Path:

field <fld> Indicates the field to be accessed

|

index <idx> Indicates the index of the accessed field, if it is an array type field

Mode:

read Enables read mode, in which case it is not necessary to include operations

|

write Enables editor mode, it is necessary to include the desired operation (<op>)

Ops:

object ins <fld> <value> Inserts a field with value in the object accessed by path

object set <fld> <value> Modifies a field with value in the object accessed by path

object del <fld> Delete a field with value in the object accessed by path

|

array ins <idx> <value> Inserts a value at the indicated index of the array accessed by path

array set <idx> <value> Modifies a value at the indicated index of the array accessed by path

array del <idx> Delete a value at the indicated index of the array accessed by path

Value:

nestedobject <name> an empty nested object (dictionary) - Unique value for arrays

object an empty object (dictionary)

array an empty array (list)

string <str> a string <str>

number <num> a number <num>

bool (true | false) a boolean

null a null value

2 - AMDIFF: Automerge-JSON command line diff

Installation

1. Install NPM:

```
$ sudo apt install npm
```

2. Install NodeJS:

```
$ sudo apt install nodejs
```

3. Install Automerge:

```
$ npm i automerge@1.0.1-preview.7
```

4. Clone the amdiff repository to use the tool::

```
$ git clone https://github.com/fabiobosisio/amdiff.git
```

5. Use the amdiff app inside the /home/user/amdiff directory

Basic commands

The amdiff.js commands are as follows:

Usage:

node amdiff.js <oldfile> <currentfile> [verbose]

node amdiff.js help

Help:

Displays this manual

Verbose:

Enable verbose mode

Oldfile:

Name of previous file

Actualfile:

Name of actual file

3 - AMPATCH: Automerge-JSON command line patch

Installation

1. Install NPM:

```
$ sudo apt install npm
```

2. Install NodeJS:

```
$ sudo apt install nodejs
```

3. Install Automerge:

```
$ npm i automerge@1.0.1-preview.7
```

4. Clone the ampatch repository to use the tool::

```
$ git clone https://github.com/fabiobosisio/ampatch.git
```

5. Use the ampatch app inside the /home/user/ampatch directory

<u>**Basic commands**</u>

The ampatch.js commands are as follows:

Usage:

node ampatch.js <oldfile> <difffile> <newfile> [verbose]   node ampatch.js help

Help:

Displays this manual

Verbose:

Enable verbose mode

Oldfile:

Name of previous file

Difffile:

Name of the file containing the differences

Newfile:

Name of the new file to be generated

4 - Freechains-JSON: Automerge-JSON synch for Freechains

<u>**Installation**</u>

1. Install java and libsodium (Freechains dependencies):

```
$ sudo apt install default-jre libsodium23
```

2. Install freechains:

```
$ wget https://github.com/Freechains/README/releases/download
/v0.10.0/install-v0.10.0.sh
```

```
# choose one:
sh install-v0.10.0.sh .                      # either unzip to
current directory (must be in the PATH)
sudo sh install-v0.10.0.sh /usr/local/bin  # or     unzip to
system  directory
```

3. Install NPM:

```
$ sudo apt install npm
```

4. Install NodeJS:

```
$ sudo apt install nodejs
```

5. Install Automerge:

```
$ npm i automerge@1.0.1-preview.7
```

For the collaborative proposal to work, it is necessary to make a small adjustment in Automerge [16] so that it allows forks from previous versions of the file. Natively it does not allow comparisons of versions created from a single point fork.

Copy the modified new.js file contained in https://github.com/fabiobosisio/freechains-json/tree/main/Mods to the following directory at your installation location:

```
‘‘‘
/home/user/node_modules/automerge/backend
‘‘‘
```

6. Clone the freechains-json repository to use the tool::

```
$ git clone https://github.com/fabiobosisio/freechains-json.git
```

7. Use the ampatch app inside the /home/user/ampatch directory

Basic commands

The basic use of Freechains-json is very straightforward:

- Commit - Publish the difference between the local file and the most current content on the network. The publication is done in binary to save space on the network.

```
node freechains-json.js --host=localhost:8330 (opcional) commit <chain>
 <file_to_upload> --sign=<pvt> (opcional) --verbose (opcional)
```

- Checkout - Download the content from the network and recompose the file with the latest content.

```
node freechains-json.js --host=localhost:8330 (opcional) checkout <chain>
 <file_to_download> --verbose (opcional)")
```

Execution step by step

- Start a Freechains host:

```
‘‘‘
$ freechains-host start /home/pi/servers/distsys
‘‘‘
```

- Switch to another terminal.
- Create an identity:

```
‘‘‘
$ freechains crypto pubpvt "secret password" - creates two keys, public
and private
96700ACD1128035FFEF5DC264DF87D5FEE45FF15E2A880708AE40675C9AD039E
‘‘‘
```

- Create and Join the public forum '#p2pforum':

‘‘‘

```
    $ freechains chains join '#p2pforum' '96700ACD1...' - type the full
    shared key above
C40DBB...
```

‘‘‘

- Initializing an Automerge-json file using AMRW. (i.e p2p.am)

  ‘‘‘

  ```
  $ node amrw.js p2p init
  ```

  ‘‘‘

- Creating an object:

  ‘‘‘

  ```
  $ node amrw.js p2p write object ins Sections object {"Sections":{}}
  ```

  ‘‘‘

- Post an Automerge JSON based share file (i.e p2p.am):

  ‘‘‘

  ```
  $ freechains-json --host=serverhost:8330 commit \#p2pforum p2p
   --sign=003030E0D03030D...
  ```

  ‘‘‘

- Communicate with other peers:
  - Start another 'freechains' host.
  - Join the same private chain '#p2pforum'.
  - Synchronize with the first host.

  ‘‘‘

```
$ freechains-host --port=8331 start /home/pi/servers/distsys
# switch to another terminal
$ freechains --host=localhost:8331 chains join '#p2pforum' 96700A...
 # type same key
C40DBB...
$ freechains --host=localhost:8330 peer localhost:8331
    send '#p2pforum'
‘‘‘
```

The last command sends all new posts from ‘8330‘ to ‘8331‘.

- Create an identity:

```
‘‘‘
$ freechains crypto pubpvt "secret password" # creates two keys,
 public and private
003030E0D03030DEF5DC264DF87D5FEE45FF15E2A880708AE40675C9AD039E
‘‘‘
```

- Checking out of the actual content of the public forum

```
‘‘‘
freechains-json --host=serverhost:8330 commit checkout #p2pforum p2p
‘‘‘
```

- Use the AMRW to creates a field (changing/including/deleting something) inside the object file resulting of checkout (in our example, p2p.am)

```
‘‘‘
$ node amrw.js p2p field Sections write object ins
    Introduction string "P2P network is..."
{ Sections: { Introduction: 'P2P network is...' } }
‘‘‘
```

- Post the Automerge JSON based share file (i.e p2p.am):

‘‘‘

```
$ freechains-json --host=serverhost:8331 commit #p2pforum p2p
 --sign=003030E0D03030D...
```

‘‘‘

- Sends and receives all new posts from ‘8331‘ to ‘8330‘.

‘‘‘

```
$ freechains --host=localhost:8331 peer localhost:8330 send '#p2pforum'
$ freechains --host=localhost:8331 peer localhost:8330 recv '#p2pforum'
```

‘‘‘

REFERENCES

[1] SPINELLIS, D. Version control systems. *IEEE software*, IEEE, v. 22, n. 5, p. 108–109, 2005.

[2] WU, Q.; PU, C.; FERREIRA, J. E. A partial persistent data structure to support consistency in real-time collaborative editing. In: IEEE. *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*. [S.l.], 2010. p. 776–779.

[3] ATTEBURY, R. et al. Google docs: a review. *Against the Grain*, v. 20, n. 2, p. 9, 2008.

[4] PRESTON-WERNER, T. et al. Github. *URL: https://github. com [As of: 11.01. 2017]*, 2008.

[5] WIKIPEDIA. *Wikipedia*. [S.l.]: PediaPress, 2004.

[6] PINCHEIRA, M. et al. A decentralized architecture for trusted dataset sharing using smart contracts and distributed storage. *Sensors*, MDPI, v. 22, n. 23, p. 9118, 2022.

[7] ANDROUTSELLIS-THEOTOKIS, S.; SPINELLIS, D. A survey of peer-to-peer content distribution technologies. *ACM computing surveys (CSUR)*, ACM New York, NY, USA, v. 36, n. 4, p. 335–371, 2004.

[8] RODRIGUES, R.; DRUSCHEL, P. Peer-to-peer systems. *Communications of the ACM*, ACM New York, NY, USA, v. 53, n. 10, p. 72–82, 2010.

[9] DOUCEUR, J. R. The sybil attack. In: SPRINGER. *Peer-to-Peer Systems: First InternationalWorkshop, IPTPS 2002 Cambridge, MA, USA, March 7–8, 2002 Revised Papers 1*. [S.l.], 2002. p. 251–260.

[10] LITT, G. et al. Peritext: A crdt for collaborative rich text editing. *Proceedings of the ACM on Human-Computer Interaction (PACMHCI)*, 2022.

[11] SCHWARTZ, D. et al. The ripple protocol consensus algorithm. *Ripple Labs Inc White Paper*, v. 5, n. 8, p. 151, 2014.

[12] ANDROULAKI, E. et al. Hyperledger fabric: a distributed operating system for permissioned blockchains. In: *Proceedings of the thirteenth EuroSys conference*. [S.l.: s.n.], 2018. p. 1–15.

[13] NAKAMOTO, S. Bitcoin: A peer-to-peer electronic cash system. *Decentralized business review*, p. 21260, 2008.

[14] SANT'ANNA, F.; BOSISIO, F.; PIRES, L. Freechains: Disseminação de conteúdo peer-to-peer. In: SBC. *Anais Estendidos do XX Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais*. [S.l.], 2020. p. 101–108.

[15] SHAPIRO, M. et al. Conflict-free replicated data types. In: SPRINGER. *Symposium on Self-Stabilizing Systems*. [S.l.], 2011. p. 386–400.

[16] KLEPPMANN, M.; BERESFORD, A. R. Automerge: Real-time data sync between edge devices. In: *1st UK Mobile, Wearable and Ubiquitous Systems Research Symposium (MobiUK 2018). https://mobiuk. org/abstract/S4-P5-Kleppmann-Automerge. pdf*. [S.l.: s.n.], 2018. p. 101–105.

[17] OTTE, S. Version control systems. *Computer systems and telematics*, Citeseer, p. 11–13, 2009.

[18] PREGUIÇA, N.; BAQUERO, C.; SHAPIRO, M. Conflict-free replicated data types (crdts). *arXiv preprint arXiv:1805.06358*, 2018.

[19] JOHNSON-LENZ, P.; JOHNSON-LENZ, T. Post-mechanistic groupware primitives: rhythms, boundaries and containers. *International Journal of Man-Machine Studies*, Elsevier, v. 34, n. 3, p. 395–417, 1991.

[20] HOCHE, M. W. Complexity of social systems. *Complexity*, v. 1, p. 3, 2020.

[21] Marc Rodrigues. *Collabora Online Unlocks the OpenPOWER Architecture*. 2022. Available in: `https://www.collaboraoffice.com/press-releases/collabora-online-unlocks-the-openpower-architecture`. Accessed in: March 22, 2023.

[22] ZOLKIFLI, N. N.; NGAH, A.; DERAMAN, A. Version control system: A review. *Procedia Computer Science*, Elsevier, v. 135, p. 408–415, 2018.

[23] PILATO, C. M.; COLLINS-SUSSMAN, B.; FITZPATRICK, B. W. *Version control with subversion: next generation open source version control.* [S.l.]: " O'Reilly Media, Inc.", 2008.

[24] DEEPA, N. et al. An analysis on version control systems. In: IEEE. *2020 International Conference on Emerging Trends in Information Technology and Engineering (ic-ETITE).* [S.l.], 2020. p. 1–9.

[25] SPINELLIS, D. Git. *IEEE software*, IEEE, v. 29, n. 3, p. 100–101, 2012.

[26] O'SULLIVAN, B. *Mercurial: The Definitive Guide: The Definitive Guide.* [S.l.]: " O'Reilly Media, Inc.", 2009.

[27] CHAO, J. Student project collaboration using wikis. In: IEEE. *20th conference on software engineering education & training (CSEET'07).* [S.l.], 2007. p. 255–261.

[28] Wikipedia. *Wikipedia is not a forum.* 2023. Available in: `https://en.wikipedia.org/wiki/Wikipedia:Wikipedia_is_not_a_forum`. Accessed in: May 10, 2023.

[29] WU, Q.; PU, C. *Consistency in real-time collaborative editing systems based on partial persistent sequences.* [S.l.], 2009.

[30] GOMES, V. B. et al. Verifying strong eventual consistency in distributed systems. *Proceedings of the ACM on Programming Languages*, ACM New York, NY, USA, v. 1, n. OOPSLA, p. 1–28, 2017.

[31] ALDIN, H. N. S. et al. Consistency models in distributed systems: A survey on definitions, disciplines, challenges and applications. *arXiv preprint arXiv:1902.03305*, 2019.

[32] BAILIS, P.; GHODSI, A. Eventual consistency today: Limitations, extensions, and beyond. *Communications of the ACM*, ACM New York, NY, USA, v. 56, n. 5, p. 55–63, 2013.

[33] KINGSBURY, K. *Jepsen: Cassandra.(Sept. 2013).* 2017.

[34] BROWN, R. et al. Riak dt map: A composable, convergent replicated dictionary. In: *Proceedings of the First Workshop on Principles and Practice of Eventual Consistency.* [S.l.: s.n.], 2014. p. 1–1.

[35] Jo Stichbury. *CRDTs solve distributed data consistency challenges*. 2023. Available in: `https://ably.com/blog/crdts-distributed-data-consistency-challenges`. Accessed in: February 15, 2023.

[36] SMITH, Z. Overview of operational transformation. In: *UMM CSci Senior Seminar Conference*. [S.l.: s.n.], 2012.

[37] LEUNG, C. Operational transformation in cooperative software systems. *McGill Science Undergraduate Research Journal*, v. 8, n. 1, 2013.

[38] SHAPIRO, M. et al. *A comprehensive study of convergent and commutative replicated data types*. Tese (Doutorado) — Inria–Centre Paris-Rocquencourt; INRIA, 2011.

[39] SIVASUBRAMANIAN, S. Amazon dynamodb: a seamlessly scalable non-relational database service. In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. [S.l.: s.n.], 2012. p. 729–730.

[40] TERRY, D. B. et al. Managing update conflicts in bayou, a weakly connected replicated storage system. *ACM SIGOPS Operating Systems Review*, ACM New York, NY, USA, v. 29, n. 5, p. 172–182, 1995.

[41] PREGUIÇA, N.; SHAPIRO, M.; MATHESON, C. Semantics-based reconciliation for collaborative and mobile environments. In: SPRINGER. *On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE: OTM Confederated International Conferences, CoopIS, DOA, and ODBASE 2003, Catania, Sicily, Italy, November 3-7, 2003. Proceedings*. [S.l.], 2003. p. 38–55.

[42] KLEPPMANN, M. et al. Local-first software: you own your data, in spite of the cloud. In: *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. [S.l.: s.n.], 2019. p. 154–178.

[43] KLEPPMANN, M. et al. Interleaving anomalies in collaborative text editors. In: *Proceedings of the 6th Workshop on Principles and Practice of Consistency for Distributed Data*. [S.l.: s.n.], 2019. p. 1–7.

[44] KLEPPMANN, M.; BERESFORD, A. R. A conflict-free replicated json datatype. *IEEE Transactions on Parallel and Distributed Systems*, IEEE, v. 28, n. 10, p. 2733–2746, 2017.

[45] Alex Good, Andrew Jeffery. *Binary Document Format.* 2023. Available in: `https://automerge.org/automerge-binary-format-spec/#_column_types`. Accessed in: February 19, 2023.

[46] GRABER, J. *Decentralized Social Ecosystem Review.* [S.l.], 2021.

[47] POUREBRAHIMI, B.; BERTELS, K.; VASSILIADIS, S. A survey of peer-to-peer networks. In: *Proceedings of the 16th annual workshop on Circuits, Systems and Signal Processing.* [S.l.: s.n.], 2005. p. 570–577.

[48] KIM, H.-c. *P2p overview.* [S.l.], 2001.

[49] GALÁN-JIMÉNEZ, J.; GAZO-CERVERO, A. Overview and challenges of overlay networks: A survey. *Int J Comput Sci Eng Surv (IJCSES)*, v. 2, p. 19–37, 2011.

[50] PETERSON, L. L.; DAVIE, B. S. *Computer networks: a systems approach.* [S.l.]: Elsevier, 2007.

[51] ANITHA, A.; JAYAKUMARI, J.; MINI, G. V. A survey of p2p overlays in various networks. In: IEEE. *2011 International Conference on Signal Processing, Communication, Computing and Networking Technologies.* [S.l.], 2011. p. 277–281.

[52] DEMIAN, M. N.; SADEK, R. A.; SELIM, G. I. Upda: Uniform peer-to-peer distribution algorithm for mesh overlay paradigm. In: *International Conference on New Paradigm in Electronics and Information Technology.* [S.l.: s.n.], 2013. v. 2, n. 21.

[53] EGER, K.; KILLAT, U. Bandwidth trading in unstructured p2p content distribution networks. In: IEEE. *Sixth IEEE International Conference on Peer-to-Peer Computing (P2P'06).* [S.l.], 2006. p. 39–48.

[54] SHERMAN, C. Napster. *Online*, Information Today Inc., v. 24, n. 6, p. 16–23, 2000.

[55] GNUTELLIUMS, L. *Gnutella protocol specification version 0.4.* 2008.

[56] POUNDS, E. *Introducing BitTorrent Sync 1.4: An Easier Way to Share Large Files (2014)*. 2015.

[57] CLARKE, I. et al. Freenet: A distributed anonymous information storage and retrieval system. In: SPRINGER. *Designing privacy enhancing technologies: international workshop on design issues in anonymity and unobservability Berkeley, CA, USA, July 25–26, 2000 Proceedings*. [S.l.], 2001. p. 46–66.

[58] LIANG, J.; KUMAR, R.; ROSS, K. W. *Understanding kazaa*. [S.l.]: submitted, 2004.

[59] MORRIS, R. et al. Chord: A scalable peer-to-peer look-up protocol for internet applications. *IEEE/ACM Transactions On Networking*, Citeseer, v. 11, n. 1, p. 17–32, 2003.

[60] RATNASAMY, S. et al. A scalable content-addressable network. In: *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*. [S.l.: s.n.], 2001. p. 161–172.

[61] ROWSTRON, A.; DRUSCHEL, P. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In: SPRINGER. *Middleware 2001: IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg, Germany, November 12–16, 2001 Proceedings 2*. [S.l.], 2001. p. 329–350.

[62] MAYMOUNKOV, P.; MAZIÈRES, D. Kademilia: A peer-to-peer information system based on xor metric, 1st intl. In: *Workshop on Peer-to-Peer Systems*. [S.l.: s.n.], 2002.

[63] ANGELIS, S. D. et al. Pbft vs proof-of-authority: Applying the cap theorem to permissioned blockchain. 2018.

[64] BUTERIN, V. et al. Proof of stake faq, 2016. *URL https://github.com/ethereum/wiki/wiki/Proof-of-Stake-FAQ.[Online*, 2021.

[65] LARIMER, D. Delegated proof-of-stake (dpos). *Bitshare whitepaper*, v. 81, p. 85, 2014.

[66] Diaspora. *Diaspora*. 2017. Available in: `https://diasporafoundation.org`. Accessed in: May 01, 2023.

[67] W3C. *ActivityPub W3C Recommendation 23 January 2018*. 2018. Available in: `https://www.w3.org/TR/activitypub/Overview.html`. Accessed in: May 01, 2023.

[68] Matrix. *Matrix*. 2014. Available in: `https://matrix.org`. Accessed in: May 01, 2023.

[69] AMIRI, M. J.; AGRAWAL, D.; ABBADI, A. E. Sharper: Sharding permissioned blockchains over network clusters. In: *Proceedings of the 2021 international conference on management of data*. [S.l.: s.n.], 2021. p. 76–88.

[70] Mastodon. *Mastodon*. 2016. Available in: `https://joinmastodon.org`. Accessed in: May 01, 2023.

[71] WALLACH, D. S. A survey of peer-to-peer security issues. In: SPRINGER. *Software Security—Theories and Systems: Mext-NSF-JSPS International Symposium, ISSS 2002 Tokyo, Japan, November 8–10, 2002 Revised Papers*. [S.l.], 2003. p. 42–57.

[72] IPFS Docs. *IPFS Gateway*. 2023. Available in: `https://docs.ipfs.tech/concepts/ipfs-gateway`. Accessed in: May 01, 2023.

[73] ROBINSON, D. C. et al. The dat project, an open and decentralized research data tool. *Scientific data*, Nature Publishing Group, v. 5, n. 1, p. 1–4, 2018.

[74] TARR, D. et al. Secure scuttlebutt: An identity-centric protocol for subjective and decentralized applications. In: *Proceedings of the 6th ACM conference on information-centric networking*. [S.l.: s.n.], 2019. p. 1–11.

[75] BACH, L. M.; MIHALJEVIC, B.; ZAGAR, M. Comparative analysis of blockchain consensus algorithms. In: IEEE. *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. [S.l.], 2018. p. 1545–1550.

[76] YADAV, A. K.; SINGH, K. Comparative analysis of consensus algorithms of blockchain technology. In: SPRINGER. *Ambient Communications and Computer Systems: RACCCS 2019*. [S.l.], 2020. p. 205–218.

[77] ĐURđIĆ, D.; STAKIĆ, Đ. Online encyclopedias wikipedia and everipedia as technological innovations and their application in the educational process. In: *BOOK OF PROCEEDINGS*. [S.l.: s.n.], 2021. p. 77.

[78] OKX Learn. *What Is Everipedia?* 2023. Available in: `https://www.okx.com/learn/what-is-everipedia`. Accessed in: May 10, 2023.

[79] Addy Baird. *White House-credentialed media outlet falsely accuses 'far left loon' of Las Vegas shooting.* 2017. Available in: `https://thinkprogress.org/gateway-pundit-geary-danley-5280ad08276f/`. Accessed in: May 16, 2023.

[80] Elaine Lee. *Wharton dropout creates Wikipedia alternative alongside Rap Genius co-founder.* 2016. Available in: `https://web.archive.org/web/20160325012952/http://www.thedp.com/article/2016/03/wharton-dropout-founds-wikipedia-competitor`. Accessed in: May 16, 2023.

[81] SKAF-MOLLI, H.; CANALS, G.; MOLLI, P. Dsmw: Distributed semantic mediawiki. In: SPRINGER. *The Semantic Web: Research and Applications: 7th Extended Semantic Web Conference, ESWC 2010, Heraklion, Crete, Greece, May 30–June 3, 2010, Proceedings, Part II 7.* [S.l.], 2010. p. 426–430.

[82] RAHHAL, C. et al. Multi-synchronous collaborative semantic wikis. In: SPRINGER. *Web Information Systems Engineering-WISE 2009: 10th International Conference, Poznań, Poland, October 5-7, 2009. Proceedings 10.* [S.l.], 2009. p. 115–129.

[83] DOURISH, P. The parting of the ways: Divergence, data management and collaborative work. In: SPRINGER. *Proceedings of the Fourth European Conference on Computer-Supported Cooperative Work ECSCW'95: 10–14 September, 1995, Stockholm, Sweden.* [S.l.], 1995. p. 215–230.

[84] WEISS, S.; URSO, P.; MOLLI, P. Logoot: A scalable optimistic replication algorithm for collaborative editing on p2p networks. In: IEEE. *2009 29th IEEE International Conference on Distributed Computing Systems.* [S.l.], 2009. p. 404–412.

[85] MORRIS, J. C. Distriwiki: a distributed peer-to-peer wiki network. In: *Proceedings of the 2007 international symposium on Wikis.* [S.l.: s.n.], 2007. p. 69–74.

[86] JOHNSON, H. A. Trello. *Journal of the Medical Library Association: JMLA*, Medical Library Association, v. 105, n. 2, p. 209, 2017.

[87] GONZALEZ, R. Figma wants designers to collaborate google-docs style. *Wired, Conde Nast*, v. 26, 2017.

[88] Quip. *Introducing Quip*. 2023. Available in: `https://quip.com/blog/introducing-quip`. Accessed in: May 10, 2023.