CS438: Decentralized Encyclopedia

Bastien Faivre
bastien.faivre@epfl.ch

Antoine Mouran
antoine.mouran@epfl.ch

Pedro Palacios Almendros
pedro.palaciosalmendros@epfl.ch

Martynas Rimkevicius
martynas.rimkevicius@epfl.ch

Luis Romero Rodriguez
luis.romerorodriguez@epfl.ch

Oriol Saguillo Gonzalez
oriol.saguillogonzalez@epfl.ch

I. Executive Summary

Crucial in modern society, online encyclopedias are widely relied upon for factual information and wield substantial opinion-shaping influence. Centralized moderation, enabling censorship, poses a significant threat to freedom of speech, particularly in today's polarized society.

This project seeks to create a decentralized rich-text encyclopedia without centralized moderators, to avoid censorship or a single narrative trumping all other alternative opinions. To achieve this, every node can propose article modifications (using a rich-text CRDT primitive to merge asynchronous modifications). When a node wants to read an article, it first collects all modifications from that article, weights the votes (positive or negative) from the web of trust network, and decides whether a given modification is shown to the user by default. Therefore, every user may have a different local view of each article. The user can always choose manually to override this decision to explore other alternatives. There is also a global reputation system that ranks all nodes in the system according to all the votes to their proposed article modifications by every peer in the system. This allows nodes to discover other nodes outside their comfort zone and avoid echo chambers. Our system intends to be mostly sybil-resistant and partially byzantine-fault-resistant.

The main building blocks of the project are a voting system based on a Web of Trust and its complements to ease article search and create a peer ranking, a new CRDT that allows merging of articles to create unique views and a graphical user interface to easily interact with the encyclopedia.

The individual components of the system are the following:

1) Article Management: it is a low-level component that allows article-modification creation, storage, sharing, and search. All article modifications are cryptographically signed and verified to avoid impersonation.
2) CRDT: enables combination and merging of concurrent article modifications. We have implemented a subset of a modern rich-text CRDT algorithm called Peritext [1].
3) Web of Trust: allows trusting of certain users using their public key and creates a network where closer nodes in the web of trust graph are given more importance for voting. Nodes can dynamically add or remove other nodes as directly connected peers in the web of trust graph.
4) Voting System: allows the users to express their opinions on article modifications, shape their local view of the article, share that opinion with their web of trust network, and affect the global reputation of nodes. Votes are digitally signed and a proof-of-work system is used as a rate-limiter.
5) Global reputation: this module computes a global rank of nodes according to the votes their proposed modification has received. State-of-the-art decentralized reputation management named EigenTrust [2] is used.
6) Encyclopedia: this module handles vote weighting, decides whether a modification is shown or not, flattens the modifications dependency graph for CRDT to merge it, and offers ready-to-call functions to the frontend user.
7) Frontend: given the complex presentation requirements of the project (rich-text rendering and editing, article modification graph rendering, and interactivity) a lot of effort has gone into developing an appropriate web-based frontend.

This project is an expansion of Peerster. The encyclopedia is based on a vote broadcasting system for articles and their modifications. Each node uses its own Web of Trust graph to weight article votes during searches, to make it possible to filter uninteresting articles. We have partially implemented the Peritext [1] rich-text CRDT algorithm to enable the merging of articles, with the possibility of creating unique articles for each peer. For global peer reputation, we have implemented the EigenTrust [2] reputation management algorithm. For the front end, we have created a ReactJS web application, using the QuillJS [3] rich-text editor.

We have created multiple unit tests and an integration test, which are currently run on the CI pipeline to detect regressions. Our unit tests cover 88% of our implementation's statements. We have also created some performance benchmarks to analyze how our system would scale and plot the results.

II. Background and Related Work

This decentralized encyclopedia is an expansion of the Peerster system proposed by the DEDIS laboratory [4]. Most of the system reuses and is dependent on the basic functionalities offered by this solution. Node message processing, network topology management by each independent peer, message traffic, and data-sharing solutions are reused by the proposed encyclopedia. The only parts of the system that are not reused are the blockchain and consensus support. Our reputation-based system for peer ranking is based on the EigenTrust algorithm [2]. Article construction and merging are based on a subset of the functionalities proposed by Peritext [1] to create a rich-text CRDT.

This is not the first attempt at creating a decentralized encyclopedia. Some examples include:

- Gollum [5] uses a git-like fork article modification model, but lacks a voting system, as it runs on top of git: modifications to an article must be manually approved and merged by each node, and concurrent modifications cause merge conflicts. It does not use a rich-text CRDT.
- Federated Wikipedia [6] is an asynchronous real-time text editor with the ability to fork, branch, and merge pages. Once again, our unrestricted management of articles differs from the git-like system.
- Everpedia [7] is a blockchain-based encyclopedia in which the different nodes may propose changes. There is only one official version of each article, which is the one determined by the blockchain consensus. It carries all the associated overhead of running a blockchain.

The contributions that we have brought in this area and have not been used before are the Web of Trust, CRDT [1] modifications, and a Voting System to decide the modifications a peer wants to see in the article. Our main goal with the combination of these technologies is to have a personalized Encyclopedia for each of the nodes that participate, picking the modifications they prefer without any central moderator involved in the process.

III. Design

Our design is centered around the goal of creating a decentralized encyclopedia that isn't vulnerable to censure and is mostly sybil-resistant.

We achieve most of our claimed sybil-resistance through the use of the Web of Trust by only considering the votes of trusted peers. If a trusted node is compromised, a node can choose to remove it from the list of trusted nodes. The global reputation ranking is byzantine-resistant due to the use of the state-of-the-art algorithm EigenTrust [2].

Our implementation is partially byzantine-resistant. The design and basic architecture of the building blocks (web of trust, vote weighting, and voting) take into account that nodes outside the web of trust cannot be trusted, and authentication is done through digital signatures in all relevant messages. However, some non-impersonation attacks are still possible. As in the original Peerster, malicious traffic in the network is unchecked and may negatively affect any peer. Although not all possible byzantine attacks are prevented, the fixes for the remaining byzantine attacks are orthogonal to the project: they shouldn't entail complex architectural changes and revolve around packet and message traffic management. We have chosen to focus on implementing all the core features of the system rather than preventing all possible byzantine attacks.

In the following subsections, we will explore the design of the different core building blocks of the system.

A. Cryptography

Cryptographic tools such as asymmetric-key signature and hash functions are used for authentication and spam avoidance. Being able to associate data with a source is key in a permissionless system with attribution. It is used for authentication in the Voting System, Web of Trust, Global reputation, and Article management modules. Each node has its own instance of a cryptography object that can sign data. When a message is signed, it is associated with a nonce to avoid replay attacks. The nonce and signature are verified at the destination. This object also offers functions to generate proofs of work and to verify them to prevent spamming, acting as a rate limiter. In our system, this rate limiter is used in the Voting System.

B. Article management

The article management module is in charge of the upload, download, storage, retrieval, and search of articles. Any peer in the network should be able to freely upload any new article or propose any modification to an existing one in the encyclopedia without any dependence on any other node or peer (except messages relaying to non-neighboring nodes). From the user's perspective (and outside this module) an article is seen as the bag of all possible combinations of proposals under the same article name. In this module, an article is defined as a modification identified by its name, author, a hash of its content, and identifiers of previous articles it is based on (possibly none if it is a completely new proposal). Names of articles are not necessarily related to those of its predecessors.

Uploading a new article exposes it to any requests coming from the network for either search or download. Searching for an article uses a regular expression to return all found articles with a matching name both locally and in the network through search requests. These search requests fill the catalog of peers that any given article may be downloaded from. This behavior is very close to that of the data-sharing module from Peerster.

This module is exposed to a variety of possible Sybil attacks. The encyclopedia uses a system of Global Reputation and a Web of Trust to rank any article in the network. Therefore it is particularly important to prevent any kind of impersonation attack in the system. This module guarantees that any received article has not been tampered with and that the authors are always correctly identified.

Not all possible attacks are currently prevented in this module though. It is possible to delay article downloads by not answering requests (or eventually lying about available

articles in the node) and forcing them to manually restart them for example. This kind of behavior should become more difficult as the number of cooperative nodes increases. In the same way, it was in Peerster, it is possible to flood the network with useless or wrong messages to delay processing or to overwhelm a node's storage with fake information about articles (answering search requests with as many files as possible to overwhelm other nodes since the catalog and now the storage are updated naively) since there is currently no system to identify and possibly ignore misbehaving nodes. Other modules tackle this problem through the reputation systems.

C. CRDT

Our implementation of Peritext CRDT [1] allows us to create an article representation that merges concurrent article modifications without any conflicts. After the creation of or an edit of an article, the proposal of applied changes is serialized to bytes to be held in storage. The changes created by the user in the front end are provided to the CRDT in a Delta structure, which has to be parsed and translated into CRDT actions. Out of the existent chosen for display proposals, it produces text in HTML format which is then used in the front end.

D. Web of Trust

To weight votes to article modifications based on how much we trust the voting nodes, our decentralized encyclopedia uses a Web of Trust. Users can add a new trusted node or remove an existing trusted node. The web of trust is a directed graph in which each node is a peer. There is an edge $a \to b$ in the web of trust graph if peer a has called `AddNewTrustedNode`(b). Note that the trust relationship is not symmetric.

Each node can get the minimum distance to every node in its connected component in the web of trust. This distance is the minimum number of hops from the peer to every other peer in the web of trust. This minimum distance is a good metric of how much we trust other nodes, and can be used to weigh the votes of peers to modifications (the votes of peers closer to the node will be taken more into account than the votes of the nodes further in the web of trust graph).

E. Voting System

Users are able to vote on any modification proposal by any peer. Their vote is digitally signed to avoid impersonation and broadcast to all the peers in the network, not just to the ones in the Web of Trust. Voting accomplishes two different objectives:

- Peers which contain our node in their Web of Trust may weight this vote in order to render their local version of the article.
- Up-voting and down-voting may change the reputation of the modification creator in the global trust ranking. Although this trust ranking isn't directly related to any article rendering, other peers can use this ranking to explore nodes outside of their web of trust.

Note that a modification cannot be rendered unless all of its ancestors are also rendered. Therefore, an up-vote to a modification carries an implicit up-vote to all the ancestor modifications. A down-vote however only affects the directly voted modification proposal.

Vote messages are also accompanied by a *proof of work*. This proof of work is an easy way to avoid vote spamming and decrease Sybil attacks as a small amount of work has to be done for each message sent. It is a concept coined by [8]. Using this mechanism, messages without proof of work or with insufficient difficulty can be discarded directly.

Votes are then stored in a *vote catalog* which is indexed by the article identifier and the voter's public key. Other modules make use of this voting catalog to implement their respective functionalities.

F. Global reputation

In order to avoid echo chambers and to give users the opportunity to explore nodes outside their Web of Trust, we have implemented a global reputation ranking based on the EigenTrust [2] algorithm. This algorithm takes into account the votes received by every article modification creator to each of their proposals by every other node in the system, and outputs a sybil-resistant ranking of all the nodes in the system, even those outside our peer Web of Trust. Nodes whose modification proposals have been more upvoted will appear higher in the ranking. As the votes are broadcast across the system, each peer can locally compute the ranking.

G. Encyclopedia

This sub-module exposes a higher-level API than the article management to the user. Among other features, it enables the following functionalities:

- Support for having multiple tags in each article. Every modification may add new tags to the given article.
- Retrieving all of the different modifications for an article with a given title. From those modifications retrieved, determine which modifications will be rendered by default or not, based on what the peer has voted or the weighted voting from our web of trust network.
- Searching for articles by name and tag. For each article with a matching title, a single entry is returned with that title (although that article may contain many modifications). Moreover, a list of tags containing the union of tags in every finally rendered modification is also returned per article title. Only including tags of rendered modifications avoids attacks where a rogue node may create modifications for every article in the network with every possible tag, effectively spamming the system and rendering tags useless.
- Handling modification dependencies. Every modification may have parent modifications from which it depends. This sub-module makes sure that a modification is only rendered if all of their parents are rendered. Moreover, it linearizes the modification dependency graph respecting

the parent relationship, to be able to merge them sequentially using our rich-text CRDT implementation.
- Computing the parameters for a new modification. When the user creates a new modification for an article based on a given graph (where some modifications are rendered, and some are not), this module computes the set of immediate parent modifications that the new modification depends on and also returns the previous modifications linearized CRDT data, which is needed to generate the CRDT metadata for the new modification.

H. Frontend

To interact with the system, users access a web app (note that, for things related to the network, especially *add peer*, users use the front end of Peerster). With this specific front end, users can connect to a node through its proxy address. Then, users can create an article given a title, tags, and text. Users also have access to a network hub, where they can see, add, and remove peers from their trusted table. They also have access to the reputation ranking. Moreover, users can search for articles and read them. When reading, users can choose which version of an article they want to see. Finally, users are allowed to propose a change based on their current view of an article.

Furthermore, the interface aims to prevent malicious use of the application by checking inputs. For instance, users are not allowed to create empty proposals. This is however just a simple layer of security, but it does not prevent attacks to the node API. This is why all data coming from the front end needs to be checked by the API.

IV. Implementation

A. Cryptography

A peer-wide cryptography structure stores all the node-specific cryptographic data such as the asymmetric-key signer, and a storage of seen nonces. The nonces are sent with every signed message to avoid replay attacks. If we receive a signed message with an already processed nonce, we discard it. Note that the nonce storage is not the most space-efficient solution, as it will grow indefinitely. As an improvement, it could be implemented by associating every signed message with a sequential ID, and keeping a counter and a small set for out-of-order received IDs, but we decided to focus on other aspects, given that the original Peersteer implementation had many ever-growing memory structures such as this one.

The module exports a sign message function that signs the message payload concatenated with the current nonce. Every time a message is signed, the peer's nonce is incremented to avoid any replay attack. A message's validity is checked by the `VerifySignedMessage` function checking the correspondence between the public key, the nonce, the payload, and the signature.

Several proof of work related functions are also provided. One of them allows to *generate a proof of work message* composed of a payload and a nonce. It first takes the payload of a message and concatenates a nonce to it, and the bytes are then hashed using SHA256. If there are enough zeros at the beginning of the hash (equal to or bigger than the set difficulty), then the proof of work is valid. Otherwise, the nonce is incremented and the operation is repeated until the nonce generates a valid proof of work. The other function allows to *verify the proof of work message*: the payload is concatenated to the nonce, and a hash is applied to it, requiring only one iteration. If the hash begins with enough zeros (equal to or bigger than the set difficulty) then it is a valid proof of work.

B. Article management

Most of the implementation of the article management module is derived from the original data-sharing from Peerster. Unfortunately, the article search introduces new complexities because of the relationships between article modifications. In the original Peerster, a call to SearchAll (or SearchFirst) would fill the catalog, enabling subsequent downloads of found files. In this module, this now becomes more complicated because it might be necessary to download articles we have not searched for depending on the meta-information related to the article (because of parent-children relationships). Moreover, the search itself must also be modified since the naming store cannot be reused because uniqueness of article names is neither wanted nor guaranteed.

To solve these problems we propose multiple modifications. On article upload, the content itself is stored in the blob store the same way it was in Peerster. Nevertheless, an additional "ArticleMetafile" is stored which contains the information mentioned in III-B to identify an article (after an upload is done, 4 new objects have been created in storage, these two and their respective metafiles). To avoid impersonation and tampering, the whole author-signed ArticleMetafile is stored. Note that the content does not need to be signed since it is already identified by its hash (and the hash is already verified in Peerster's data-sharing) from the signed ArticleMetafile. The metahash of the ArticleMetafile is considered a unique identifier of the article. To allow iteration over all articles in storage, auxiliary storage fulfilling a similar role to the naming storage during searches is updated on every article upload. Any article download must take into account this structure with two stored files on each peer and request both of them from the network if necessary. Both of these methods reuse their counterparts in data-sharing.

The article search can unfortunately not reuse the original consensus and tag-based methods from Peerster. Any article search must now be performed over the new ArticleMetafiles, using the article name. To allow for later downloads, search replies contain the whole ArticleMetafile. They also give the information necessary to update the requesting peer's catalog. It is possible that an article search does not return any information about its parents (for example, when the article name has been changed between parents and child). When an article in such a case is requested, a DFS search tries to reconstruct the whole modification tree and update the catalog. To accomplish this, multiple searches for the parent articles'

names are performed. This is a possible point for performance improvement since it would be possible to do a specific search for the article we are looking for instead of a generic one that may return useless information by creating a new search procedure with its own network messages. Note that this tree traversal is only done for the articles we are interested in from the search, not necessarily all of them. Any subsequent download must perform this traversal too in case we do not already have the necessary ArticleMetafiles.

To avoid breaking the original Peerster functionality, these procedures needed the creation of new search requests and reply messages, very similar to their counterparts from data-sharing, but no new data requests or reply messages were used. In order to increase the resistance of the system to malicious peers, Peerster's whole messaging system could be expanded to filter traffic coming from malicious peers through the use of heuristics. The catalog should be adapted accordingly and wrapper functions around Download should be created to retry downloads when replying peers are misbehaving for, in its current state, a non-cooperative peer may force a manual restart of any download relayed through him.

Even though most of this module is scalable (assuming searches through the use of broadcasts remain functional), the major bottleneck is given by the size of reply messages (limited by maximal UDP message size) to search requests and the storage of useless ArticleMetafiles. To solve these problems, the module could be expanded by adding an option to clear the article storage and catalog, together with some way of dividing search replies into multiple messages. Additionally, it would be a good idea to only reply with ArticleMetafiles qualified as good by some heuristic.

C. CRDT

Our CRDT is composed of two main parts - plain text CRDT which handles the insertion and deletion of characters, and marking handlers which handle marking the characters such as bold, italic, and a link (that stores a URL). Each change to the text is handled as an operation (insert, delete, mark), and this operation is applied to the existing characters in the text or new characters are created. Our implementation is similar but not the same as the one described in the Peritext paper [1]. The deviation from the original material was due to time constraints, therefore dropping commenting, and the original algorithm being implemented in Typescript, which has union types and had an immediate effect on the frontend representation they had used. Because of this, our implementation became more verbose compared to the original from Litt *et al.* [1].

Each character holds its special ID which essentially is a Lamport timestamp, which is used to put the character in its correct position. Because of the concurrent editing capabilities, we cannot store the index of the character, so we use the Lamport timestamp of the previous character to keep the ordering of the text correct and persistent. The character itself is in the ASCII format due to an early choice in the design to store the characters as runes (integers), therefore it does not support Unicode characters. The last thing the character has to store are marks applied to that character and its liveness. The liveness is a boolean indicating whether the character was deleted (has become a tombstone) since the characters cannot be removed due to the order preservation by IDs.

Character marks are stored in two anchors of the character: *before* and *after*. These anchors are important not only for the representation of the character, but also for the merging of markings, and extension of markings when the text is inserted. When the text is marked, to create the marking operation start and end anchors are chosen. The first character's *before* anchor is chosen as the start and, depending on the marking, the last character's *after* anchor (for link) or the character's, which follows the last character, *before* anchor is chosen (for bold and italic) as the end. Then, when the marking operation is applied, the start and end anchors trivially get the operation saved, and the *before* anchors of in-between characters get the marking applied. This is where we deviate from the paper implementation - in the paper, Litt *et al.* put the marking in every anchor that is between the start and the end anchors. We found this unnecessary because our traversal of the anchors was already through the characters, thus eliminating the need for the doubling of saved marks. To represent the character, the latest marks are taken from both of the anchors to decide whether they show up (e.g. mark was applied, then deleted, and then applied again).

Character insertion is trivial in a plain text CRDT but is non-trivial with markings. It follows the paper description of *tombstones* - deleted characters that are still kept, and the extension of bold and italic marking but not link marking. Marking extensions were decided by Litt *et al.* and are the rule of thumb in most rich-text editors. When a character is inserted in between the marking anchors, it inherits the marking. Usually, that is the expected behavior for the bold and italic marks, it is intended to extend the marking. The link marking end is placed at *after* anchor so that the link does not extend. The marking anchors could belong to a *tombstone*, so if it has a link marking has to be inserted after the deleted sequence of characters such that it does not inherit its markings.

We use Delta JSON array parsing which is created by the QuillJS library [3] to forward the changes from the frontend to CRDT. Delta array stores each action that is done by the user in order. However, sometimes this is not the case when the front end mixes the order of two close actions up, but it is happening because of the library. Then that action is parsed by the CRDT infrastructure such that an operation corresponding to the action is created. Since the Delta JSON does not have Lamport counter indexes, the indexing and handling of deleted characters have to be adapted to our operation IDs during the parsing.

To export our CRDT in a storable format we decided to serialize our collection of operations into bytes, such that this can be shared between peers. Each collection of bytes is a proposal that is comprised of insert, delete, and mark operations. To use the bytes of the proposal we would create

a new fresh instance of CRDT and apply all the operations stored. Since the operations of a single proposal are guaranteed to be in order, because it is done in a single instance of creating a proposal, we can apply operations straightforwardly. This storage of operations is not efficient because for any insert, delete or mark of a character there is a whole operation (which holds more information that just the character) stored for later use.

D. Web of Trust

Our Web of Trust implementation uses a mechanism similar to link-state routing. Every node has complete knowledge of the full Web of Trust graph. Whenever a node trusts a new node or removes a pre-existent trusted node, it broadcasts its new list of directly trusted nodes (its direct edges in the Web of Trust graph) by gossiping. Every node listens to these modifications and updates their local copy of the Web of Trust graph. These broadcast messages are signed and have a nonce, to avoid impersonation or replay attacks.

Once a node wants to find out its distance to other nodes in the Web of Trust (for example to weight votes for an article modification), it first checks if it has a cached copy available. If it doesn't, or the cached copy has been invalidated, it executes a BFS search starting from the origin node and annotates the minimum distance in number of hops to every other reachable node. This distance vector is then cached for fast future accesses.

Note that in case some trusted node is compromised (for example, starts trusting a byzantine node), the full node has to be untrusted. As potential future work, an improvement would be to allow blocking certain nodes from your Web of Trust view, even if you transitively trust them. This blocking could be performed either by public key or other heuristics (for example, block nodes that trust too many peers).

E. Voting System

The Voting system is based on `VoteMessage`, which contains the following fields:

1) The identifier of the article the user is voting on.
2) The actual vote is represented by a boolean with a true value for an up-vote and a false one for a down-vote.
3) The meta-files of the article and its parents in the case of a positive vote.

The vote messages are broadcast to all peers by the `SendVote` function which is the user-facing function to cast a vote to an article modification.

First, a `VoteMessage` is generated with a given article identifier and a boolean representing the vote. If the cast vote is a down-vote, only the voted article meta-file is included in the vote message. If the vote is otherwise an up-vote, all meta-files of the article modification ancestors are also included in the vote message. The ancestors' meta-files are retrieved by iterating recursively over all the article's parents. The ancestor meta-files are needed because an up-vote also implicitly up-votes the ancestor modifications of the voted modification (as a modification cannot be rendered unless all their ancestors are also rendered).

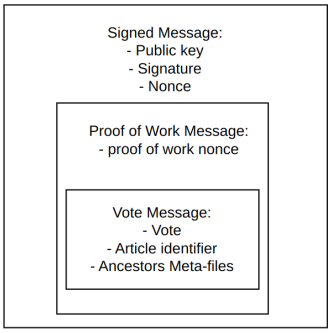

Fig. 1: Wrapped vote message

The vote message is then wrapped with a proof of work and is digitally signed. Therefore, the `VoteMessage` is nested into a `ProofOfWorkMessage`, itself nested into a `SignedMessage`, as can be seen in figure 1. Finally, the wrapped vote message is broadcast using the standard rumor system.

When received, the vote message is handled by the rumor message callback function and then processed by the signed message handler. First, the receiver checks the validity of the signed message using the signature and the nonce with the `VerifySignedMessage` function of the Cryptography module. It is then passed to the proof of work message handler, which then uses the `VerifyProofOfWork` function provided by the Cryptography module. Finally, it is interpreted as a vote message, and the vote is added to the vote catalog. If the vote is an up-vote, then up-votes to all the ancestors of the voted article are also registered in the voting catalog.

F. Global reputation

The global reputation ranking is generated using the vote catalog, which contains all the votes from all the nodes in the system to each article modification. We have adapted the EigenTrust [2] algorithm in order to generate the global trust value for each node:

1) *Generation of the satisfaction matrix*: to compute the global reputation values, we first need to know the total satisfaction from each peer to every other peer. This means that we need to obtain a square matrix indexed by the voters and authors that contains the satisfaction of a given peer with every other peer in the system. This satisfaction is computed as the sum of up-votes from peer i to article modifications proposed by peer j minus the sum of down-votes.
2) *Matrix normalization*: the satisfaction matrix $s_{i,j}$ of every peer must be normalized into a matrix $c_{i,j}$, as described in the EigenTrust paper. For each value of the matrix $s_{i,j}$ we assign $c_{i,j} = \frac{\max(0, s_{i,j})}{\sum_j \max(0, s_{i,j})}$.
One main problem we faced during the normalization of the satisfaction matrix, which remains an open issue

in our implementation, is the assignation of pre-trusted peers, peers that are assumed to be trusted by everyone in the network. As it is explained in the paper [2] in section 4.5, we need these pre-trusted peers to avoid Sybil attacks and handle the presence of inactive peers (peers that haven't voted on any other modification proposal, but may have authored one). These pre-trusted peers are usually the creators of the network, as they are assumed to have no malicious intent. We tried using the trusted nodes from the Web of Trust as pre-trusted nodes, but we learned that the output reputation ranking didn't make much sense during real tests using our voting system, as each peer had a wildly different ranking (even when giving very small weights to the pre-trusted nodes). Seeing these results, we decided not to implement the pre-trusted nodes, and leave that implementation with pre-trusted nodes in the branch `eigentrust_extended`, which hasn't been merged into the main project branch. The alternative solution we came up with in order to solve the inactive peers problem was assigning to each peer a self-satisfaction value of 1 (therefore, every peer at least likes itself, and there are no zero rows in the normalized matrix). This helps guarantee the algorithm's convergence even without pre-trusted peers.

3) *EigenTrust iterations*: the implementation of the algorithm is based on the following pseudo-code, described in the original paper [2]. We have used maps as the primary data structure to encode the matrices and vectors, and created functions to compare the distance between two vectors and the multiplication of the transposed matrix with the distribution vector.

Algorithm 1 EigenTrust Algorithm

$\bar{t}_0 = \bar{e}$
repeat
  $\bar{t}^{(k+1)} = C^T \bar{t}^{(k)}$
  $\delta = \|\bar{t}^{(k+1)} - \bar{t}^{(k)}\|$
until $\delta <$ error

4) *Sorting*: Once we have computed the final reputation assignment vector, the values are sorted into a ranking by decreasing reputation. Each ranking entry contains the public key of the node and its global reputation.

G. Encyclopedia

The article storage system allows modifications to have different names from their parent modifications, so that system is used to implement tags. A modification name consists of the article title (which is immutable across all modifications of an article), and a list of tags separated by a separator character.

After receiving all modifications for a given article a linearization must be chosen to always merge CRDT modifications in the same way, so all peers have a consistent view of an article if they choose to accept the same modifications. Moreover, as some modifications may depend on parent modifications, this linearization must take those dependencies into account: a child modification may not appear in the linearization chain before their parents. To guarantee these properties, a stable topological sort algorithm has been implemented. It's a modification of Kahn's algorithm that uses priority queues (implemented using Go's heaps) instead of sets to have a deterministic behavior.

Once the full list of modifications is linearized, it's filtered to only include the articles the node wants to see. The peer can choose to override any votes to visualize any modification he wants, to explore other modifications in the article. If the peer doesn't override the decision, its vote is taken into account (if the peer has voted). If the peer hasn't voted, then the Web of Trust votes are taken into account. The votes of each peer (an up-vote is encoded as 1 and a down-vote as -1) are weighted by a function of their distance to our node:

$$w(d) = \frac{1}{1+d}$$

Therefore, if the set of peers in our Web of Trust is P, then the computed raw vote is defined as

$$V = \sum_{p \in P} w(\text{distance}(p)) \cdot \text{vote}(p)$$

The peer will accept this modification (render it) if the vote is greater than or equal to some threshold ($V \geq T$). This threshold is defined to be 0, but the implementation could be adapted to work with any weighting function and threshold.

Note that even if the votes for some modification are over the threshold and it's accepted, it will only be rendered if all their parents are rendered too.

To determine the immediate parents for a new modification proposal based on the full graph of rendered modifications, the immediate parents are selected as all the leaf rendered nodes, that is, nodes that are rendered and no other rendered node refers to them as parents.

Finally, the encyclopedia module offers some wrappers around the article management search and download. For the search, the article titles are de-duplicated: instead of returning a list of all modifications, a list of unique article titles (each of which may have multiple modifications) is returned. For each article title, a list of tags is shown too. To avoid byzantine attacks where a node proposes all tags to all article titles and devaluates the meaning of tags, not all the tags of all proposed modifications are returned, but only those of modifications that would be rendered if the article was to be viewed.

A wrapper was also created around article download, which now takes an article title and performs a search for all modifications belonging to that article title. The modifications are topologically sorted and filtered based on voting. The final filtered result is returned.

H. Frontend

The front end uses React as a basis. Its features (mainly states and *useEffect*) allow us to have a proper functional web app to handle all actions required by the system. Regarding the design, the library MaterialUI is used; it provides beautiful

and functional components such as text fields and buttons. Regarding the CRDT part, we use the QuillJS library as mentioned in the CRDT implementation. The library provides a text area with a toolbar to allow the user to make the text bold or italic and to add a link. As described in the design section, the front end is made of four main components:

1) **Creator**: This component allows the user to create an article. For that, the components provide a text field for the article's title and another one for the tags. Notice that the one for the tags has a special behavior: it prohibits the use of empty tags, tags with spaces, or duplicated tags. Finally, the QuillJS text area is present to write the article content. When writing the article, the front end records the ordered sequence of Delta JSON events (see CRDT implementation) that will be sent alongside the title and the tags to the node API to create the article.
2) **Search**: This component provides a search bar where the user can enter a pattern that will be used to search for matching articles in the system. Upon user action, the component will contact the node API to initiate a search based on the given pattern. Upon results received, the component displays the results and the user can choose to read one of the results.
3) **Reader**: Upon being chosen by the user, an article is displayed with the Reader component (a node API call is used to retrieve all the information related to the article). This is the most complex component.
   On the left side, the user has access to an interactive directed acyclic graph showing all versions of the article (the library react-d3-dag is used). Upon hovering over the nodes, an info popup shows up with relevant information regarding this proposal such as the author or the votes. Furthermore, the color of the nodes indicates if their content is shown in the rendered rich text (see below). Clicking on a node changes its visibility status and the status of its parents or children depending on the case: if the node becomes visible, then all its parents will also be visible (due to the CRDT principle); and if the node becomes invisible, then all its children will also be invisible.
   On the right side, the content of the current shown article version is shown. This includes the tags and the QuillJS text area with the article text. Furthermore, the rendered proposals are shown with thumbs up and down to allow the user to vote for them. A user can change its vote whenever it wants.
   When editing an article, the user can extend the set of tags following the same rules as defined in the creator component. Moreover, the user can modify the text in the QuillJS text area. As for the creator component, the front end records the set of Delta JSON. To submit the proposal, the front end sends the current graph, the new set of tags, and the set of Delta JSON to the node API.
4) **Network**: Finally, the network tab shows the user its public key in the system, a table with its trusted peers (with the associated distance to them), and a table with the reputation ranking. All this information is obtained and updated via node API calls. Indeed, the user can add and remove peers from its trusted set via the provided text field and an underlying node API call.

Note that this component is complex in terms of protection against malicious or erroneous usage.

V. Evaluation

A. Correctness

To evaluate the correctness of our program, we have created many unit tests for each module. We have reached an 88% statement coverage on our extension-related code. We have implemented the tests described in the project interim report (some of them with some modifications), often with extensions and additional new ones:

1) Cryptography: There are unit tests for signing and verifying digital signatures, wrapping messages inside `SignedMessage`s, and proof of work. It is also checked that we don't accept invalid signatures.
2) Article Management: Unit tests check basic functionality described for the module both locally and in simple network topologies. Name changing between parents and children and multiple-parent relationships with simple article graph topologies are tested. All three test cases from the interim project report are also tested.
3) CRDT: The unit tests check for the correctness of the implementation of the base CRDT (the examples of intended behavior could be found in the Peritext paper [1]). The translation of delta from the front end is also tested in different-sized scenarios. Proposal (de)serialization together with delta translation is tested in a few ways, such as a single big proposal, many proposals from different nodes, and proposals from the same node that diverge and are merged afterward.
4) Web of Trust: The unit tests check that nodes can add and remove trusted peers and the distances update correctly on all peers. Several network topologies and situations such as nodes arriving later are also explored.
5) Voting System: Tests from the interim project report have been modified. Correct exchange of votes and expected update of relationships in the ranking in simple network topologies are tested. For the Eigentrust implementation, normalization of the matrix and the example case given by Pennsylvania University are tested [9].
6) Encyclopedia: Unit tests check that the topological ordering is respected with different graph topologies, that new modification parameters are computed correctly, that the votes are weighted correctly, and that the modifications are filtered based on the defined policies. User-facing functions (upload with tags, search, and fully retrieving an article by title) are tested under simple use cases.

We currently only have one integration test aimed at verifying correct interaction and coordination between modules. It consists of a simple scenario involving 3 nodes that interact

with each other using the different functions provided by the public client API. It simulates a simple usage of the encyclopedia by the front end of several peers interacting with each other.

B. Performance

A benchmark for vote propagation has been run with sets of nodes of different sizes to measure the time required to reach a synchronized state. The results can be seen in figure 2.

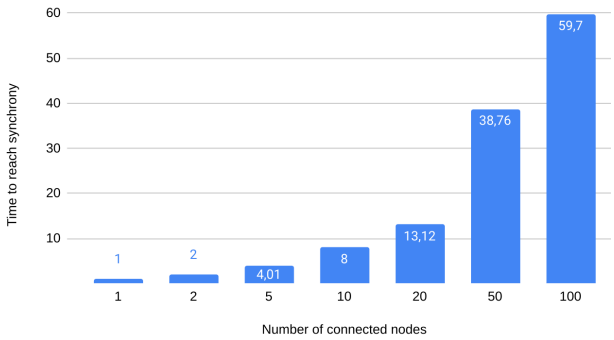

Fig. 2: Vote synchronization benchmark

In a fully connected graph, each node uploads an article. Then it votes on its article, which performs a broadcast. The benchmark stops when every node knows about every vote. The synchronization duration increases as the number of nodes becomes larger. In a very large network, it would take a long time to have every node reach the synchronized state. The local view of an article might not be up to date in this case, since the last votes might not have reached all nodes.

A possible optimization is to refrain from sending meta-files with votes in an active and mature network for two reasons: nodes likely already have the meta-files, and the volume of meta-files would unnecessarily increase with frequent modifications. At a certain scale, it is prudent to let the recipient find meta-files independently by querying the voting node or asking reliable neighboring nodes in case the voting node is unavailable.

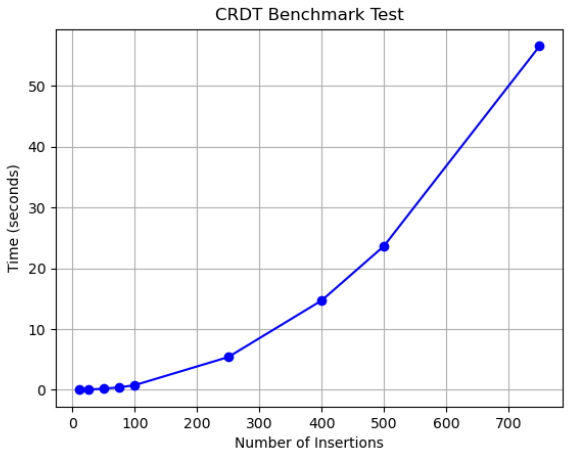

Fig. 3: CRDT benchmark

A CRDT benchmark has been run to show the time it takes to deserialize, display the current article state, apply a single change (single delta), and serialize the new proposal. The results can be seen in figure 3. It was run with a variable amount of these proposals since the number of proposals affects the length of the stored CRDT, thus the deserialization and text generation taking longer. The change was always inserted at the start of the text, therefore change insertion does not affect the performance of the test significantly. The biggest effect is the HTML produced which traverses every character with its marks at the anchors (no markings were applied here either). The result was close to the polynomial growth of time in relation to the number of insertions due to many loops involving deserialization, text display, and delta translation. The CRDT benchmark can be reproduced using the project's Makefile to launch the performance tests. Any special test conditions must be added there.

The CRDT could be optimized in several ways. One of them would be optimizing the storage of markings - if a marking is deleted, the mark from those characters is removed as well, while it is currently stored. Another optimization would be to use a more suitable JSON to parse from the front end or write our own, as Litt *et al.* [1] did.

VI. Conclusion

Our decentralized encyclopedia is an extended version of Peerster. On this platform, any user can upload rich-text articles without any kind of censorship or centralized intervention. Each peer has a unique view of the encyclopedia that depends on his choice of trusted peers. Each participant is therefore its own and only moderator. To give anyone a chance to explore new views of the encyclopedia, we offer a peer ranking to identify peers that have been democratically chosen as trustworthy. Additionally, a GUI has been developed as an easy way to use the encyclopedia. Correctness of the project is guaranteed by a high testing unit reaching a coverage of 88% and manual testing.

REFERENCES

[1] G. Litt, S. Lim, M. Kleppmann, and P. van Hardenberg, "Peritext: A crdt for collaborative rich text editing," Proc. ACM Hum.-Comput. Interact., vol. 6, no. CSCW2, nov 2022. [Online]. Available: https://doi.org/10.1145/3555644

[2] S. D. Kamvar, M. T. Schlosser, and H. Garcia-Molina, "The eigentrust algorithm for reputation management in p2p networks," in Proceedings of the 12th international conference on World Wide Web. ACM, 2003, pp. 640–651.

[3] "QuillJS documentation," https://quilljs.com/docs/quickstart/, accessed: 2024-01-17.

[4] "Decentralized distributed systems laboratory," https://www.epfl.ch/labs/dedis/, accessed: 2024-01-21.

[5] Gollum, "Gollum: A simple, git-powered wiki with a sweet api and local frontend," https://github.com/gollum/gollum.

[6] P. Foundation. Federated wiki. [Online]. Available: https://wiki.p2pfoundation.net/Federated_Wiki

[7] S. K. et. al. Everipedia. [Online]. Available: https://en.wikipedia.org/wiki/Everipedia

[8] C. Dwork and M. Naor, "Pricing via processing or combatting junk mail," Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 740 LNCS, pp. 139–147, 1993. [Online]. Available: https://link.springer.com/chapter/10.1007/3-540-48071-4_10

[9] A. G. West. Reputation management algorithms & testing. [Online]. Available: https://rtg.cis.upenn.edu/qtm/doc/p2p_reputation.pdf