Peer-to-Peer Permissionless Consensus via Authoring Reputation

Francisco Sant'Anna *Department of Computer Science, Rio de Janeiro State University*

Abstract—Public Internet forums suffer from excess and abuse, such as SPAM and fake news. Centralized platforms employ filtering and anti-abuse policies, but imply full trust from users. We propose a permissionless Sybil-resistant peer-to-peer protocol for content sharing. Our main contribution is a reputation system that moderates content and, at the same time, delivers network consensus. We can trace a parallel with Bitcoin: new posts create reputation (vs proof-of-work), likes and dislikes transfer reputation (vs transactions), and aggregate reputation determines consensus (vs longest chain). The reputation mechanism depends exclusively on the human authoring ability (*proof-of-authoring*), which is slow and scarce, thus suitable to establish consensus. As an application example, we prototype a permissionless decentralized version control system that, based on consensus, resolves conflicts automatically.

Index Terms—Bitcoin, blockchains, CRDT, distributed consensus, peer-to-peer, publish-subscribe, reputation system, VCS

1 Introduction

Content publishing in Internet forums and social media is increasingly more centralized in a few companies (e.g. Facebook and Twitter) [1], [2], [3]. On the one hand, these companies offer free storage, friendly user interfaces, and robust access. On the other hand, they concentrate more power than required to operate by collecting users' data and "algorithmizing" consumption. Peer-to-peer alternatives [4] eliminate intermediaries, but strive to achieve consistency while dealing with malicious users.

In an ideal Internet forum, all messages or posts (i) reach all users, (ii) are delivered in a consistent order, and (iii) are respectful and on topic. In a centralized system, items (i) and (ii) are trivially achieved assuming availability and delivery order in the service, while for item (iii), users have to trust the service to moderate content. In a decentralized setting, however, none of these demands are easily accomplished. A common approach in gossiping protocols is to proactively replicate and disseminate posts among peers until they reach all users [4], [5]. However, this approach does not guarantee consensus since posts can be received in conflicting orders [6], [7]. Consensus is key to eradicate Sybil attacks [8], 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 in order to satisfy item (iii).

Consensus is notably challenging to the point that decentralized protocols partially abdicate of it. On the one hand, federated protocols [9] offer *multi-user/single-node consensus*, in which multiple users can exchange messages consistenlty within a single trusted server, but not globally across multiple servers. On the other hand, a protocol like Scuttlebutt [10] 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 goal is to provide *multi-user/multi-node consensus* (just *consensus*, from now on) in the context of decentralized content sharing.

Bitcoin [11] is the first permissionless protocol to resist Sybils through consensus. Its key insight is to rely on a scarce resource—the *proof-of-work*—to establish 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 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 original decentralization goals. In particular, a unique timeline implies that all Internet content should be subject to the same consensus rules, which neglects all subjectivity that is inherent to social content. Another limitation of cryptocurrencies is that it is not possible to revoke content in the middle of the blockchain, which is inadmissible considering illegal content (e.g., hate speech).

In this work, we adapt Bitcoin's idea of a scarce resource to reach consensus in the context of content sharing. Our first contribution is to recognize the actual published contents as the protocol scarce resources, since they require human work. Work is manifested as new posts, which if approved by others, reward authors with reputation tokens, which are used to evaluate other posts with likes and dislikes. With such *proof-of-authoring* mechanism, token generation is expensive, while verification is cheap and made by multiple users. Due to decentralization, posts in a timeline form a causal graph with only partial order, which we promote to a total order based on the reputation of authors. The consensus order is fundamental to detect conflicting operations, such as likes with insufficient reputation (akin to Bitcoin's double spending). Our second contribution is to allow that users create diversified forums of interest (instead of a singleton blockchain), each counting as an independent timeline with its own subjective consensus etiquette. Our third contribution is to support content removal without compromising the integrity of the decentralized blockchain. Users have the power to revoke posts with dislikes, and

peers are forced to remove payloads, only forwarding associated metadata. We integrated the proposed consensus algorithm into Freechains [12], a practical peer-to-peer content dissemination protocol that provides strong eventual consistency [13], [14].

As an application example, we prototyped a permissionless decentralized version control system (dVCS) that relies on consensus to apply automatic merges. A dVCS is relevant because (i) it is a collaborative application, (ii) with commits that need evaluation from users, and (iii) with merges that require human intervention, which we propose to automate. Hence, as a forth contribution, we show how the consensus mechanism can empower mostly conflict-free replicated data types (*quasi-CRDTs*) when they do encounter conflicting operations.

In summary, we propose (i) a consensus mechanism based on proof-of-authoring, (ii) for independent user-generated blockchains, (iii) which supports content removal, and (iv) which can automate conflict resolution in quasi-CRDTs. As a main limitation, the forum causal graphs are ever growing data structures that carry considerable metadata overhead. The costs to store and validate posts increase over time, which is a limitation of blockchains in general. Nevertheless, to show the practicability of the protocol, we simulated months of activity of a chat channel and years of a newsgroup forum, both extracted from publicly available archives.

The rest of the paper is organized as follows: In Section 2, we introduce the basic functionalities of Freechains to create, evaluate, and synchronize posts. In Section 3, we describe the reputation and consensus mechanism applied to public forums and evaluate its perfomance. In Section 4, we discuss the correspondences with CRDTs, and prototype a simple dVCS. In Section 5, we compare our system with publish-subscribe protocols, federated applications, and fully peer-to-peer systems. In Section 6, we conclude this work.

2 FREECHAINS

Freechains [12] is an unstructured peer-to-peer topic-based publish-subscribe protocol, in which each topic or *chain* is a replicated *Merkle Directed Acyclic Graph* [15] (just *DAG*, from now on). The DAG represents the causal relationships between the messages, whose cryptographic links ensure persistence and self certification. Considering that the DAG itself represents the state of the peers, the protocol ensures strong eventual consistency [16], [5]. The operation of the protocol is typical of publish-subscribe systems: an author publishes a post to a chain, and subscribed users eventually receive the message.

The goals of this section are twofold: (i) to depict chain DAGs as the result of basic protocol operations; and (ii) to illustrate how permissionless protocols inevitably require some form o Sybil resistance to become practical.

Freechains supports multiple arrangements of public and private communications, which are detailed in Table 1. In this section, we operate a *private group* to describe the basic behavior of chains without consensus. We use the actual command-line tool provided by the protocol to guide the discussion through concrete examples. At the end of this section, we also exemplify a *public identity* chain for the sake of completeness. In Section 3, we focus on the behavior of *public forums*, which involves untrusted communication between users and require the proposed reputation and consensus mechanism.

All Freechains operations go through a *daemon* (akin to Bitcoin's full nodes) that validates posts, links the DAGs, and communicates with other peers to synchronize the graphs. The command that follows starts a daemon in the background to serve further operations:

```
> freechains-daemon start '/var/freechains/' &
```

The actual chain operations use a separate client to communicate with the daemon. The next sequence of commands (i) creates a shared key, (ii) joins a private group chain (prefix $), and (iii) posts a message into the chain:

```
> freechains keys shared 'strong-password'  <- (i)
A6135D..   <-- returned shared key
> freechains '$family' join 'A6135D..'      <- (ii)
42209B..   <-- hash representing the chain
> freechains '$family' post 'Good morning!' <- (iii)
1_EF5DE3.. <-- hash representing the post
```

A private chain requires that all participants use the same shared key to join the group. A `join` only initializes the DAG locally in the file system, and a `post` also only modifies the local structure. No communication occurs at this point. Figure 1.A depicts the state of the chain after the first post. The genesis block with height 0 and hash `42209B..` depends only on the arguments given to `join`. The next block with height 1 and hash `EF5DE3..` contains the posted message.

Freechains adheres to the *local-first* software principle [17], in which networked applications can work locally while offline. Except for synchronization, all other operations in the system only affect the local replica. In particular, joining a chain with the same arguments in another peer results in the same genesis state, even if the peers have never met before. Hence, before synchronizing, other peers have to initialize the example chain with the same step (ii).

In Freechains, the operation to synchronize chain DAGs is explicit, in pairs, and unidirectional. For instance, the command `recv` asks the daemon in *localhost* to connect to daemon in *remote-ip* to receive all missing blocks from there:

```
> freechains '$family' recv '<remote-ip>'
1/1  <-- one block received from <remote-ip>
```

If applied in the new peer, the command above would put it in the same state as the original peer in Figure 1.A. The complementary command `send` would synchronize the DAG in the other direction. Note that Freechains does not synchronize peers automatically. There are no preconfigured peers, no root servers, no peer discovery. All connections happen through the `send` and `recv` commands which have to specify the peers explicitly. In this sense, Freechains is conceptually a *pubsub* on how users publish and consume content, but it still requires extra network automation.

In order to query the state of the replica, the next sequence of commands checks the hash(es) of the block(s) at the head of the local DAG (the latest blocks), and then reads the payload of the single head found:

```
> freechains '$family' heads
1_EF5DE3..     <-- hash of head block
> freechains '$family' payload '1_EF5DE3..'
Good morning!  <-- block payload
```

Type	Prefix	Arrangement	Behavior	Examples
Private Group	$	Private 1↔1, N↔N, 1↩	Trusted groups (friends or relatives) exchange encrpyted messages among them. The communication can be in pairs (1↔1), groups (N↔N) or individual (1↩).	- E-mails - WhatsApp groups - Backup of documents.
Public Identity	@	Public 1→N, 1←N	A public identity (person or organization) broadcasts authenticated content for a target audience (1→N) with optional feedback (1←N).	- News sites - Streaming services - Public profiles in social media
Public Forum	#	Public N↔N	Participants with no mutual trust communicate publicly.	- Q&A forums - Chats - Consumer-to-consumer sales

TABLE 1
The three types of chains and arrangements in Freechains.

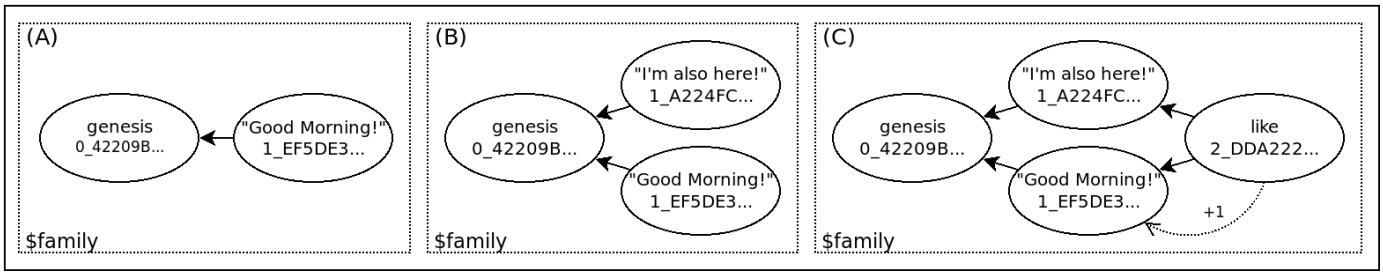

Fig. 1. Three DAG configurations. (A) Single head pointing to genesis block. (B) Fork with heads pointing to genesis block. (C) Like pointing to previous heads and also to its target.

The presented commands to join, post, synchronize, and query the chains are sufficient to create decentralized applications that publish and consume content.

Note that chains do not typically evolve to a simple list of sequential posts, but instead become DAGs with multiple heads. The main reason is that the network is inherently concurrent and users are encouraged to work locally. Hence, continuing with the example in Figure 1, suppose that the new peer posted a message before the `recv` above, when its local DAG was still in its genesis state. In this case, as illustrated in Figure 1.B, the resulting graph after synchronizing would now contain two blocks with height 1. Note that forks in the DAG create ambiguity in the order of messages, which is a fundamental obstacle to reach consensus. In private chains, we could apply simple methods, such as relying on the local source timestamps of the blocks. However, in public forums, a malicious user could modify his local time to manipulate the order of messages.

To conclude the basic operations of chains, users can rate posts with *likes* and *dislikes*, which can be consulted later:

```
> freechains '$family' like '1_EF5DE3..'
2_BF3319..     <-- hash representing the like
> freechains '$family' reps '1_EF5DE3..'
1              <-- post received 1 like
```

As illustrated in Figure 1.C, a like is a regular block with an extra link to its target. Every new block points to the previous heads, establishing a causal logical timeline in the chain. For instance, the JSON that follows represents the block `2_DDA222..` with the backs and extra like links:

```
{
  "id":    "2_DDA222..",  // block hash id
  "backs": ["1_EF5..","1_A22.."] // back links
  "time":  1650722072223, // source timestamp
  "data":  "E95DBF.."     // hash of the payload
  "like":  "1_EF5DE3.."   // like link (optional)
}
```

As we discuss in the next section, like operations in public forums have to be signed by users, and are at the core of our proposed consensus algorithm.

For the sake of completeness, Freechains also supports public identity chains (prefix @) with owners attached to public/private keys:

```
> freechains keys pubpvt 'other-password'
EB172E.. 96700A..  <-- public and private keys
> freechains '@EB172E..' join
F4EE21..           <-- hash representing the chain
> freechains '@EB172E..' post 'This is Pelé' \
   --sign='96700A..'
1_547A2D..         <-- hash representing the post
```

In the example, a public figure creates a key pair and joins an identity chain attached to his public key. Every post in the chain needs to be signed with his private key to be accepted in the network.

Note that the basic operations of Freechains to (i) create decentralized identities, (ii) publish content-addressable data, (iii) maintain Merkle DAGs, and (iv) synchronize peers are not new in the context of peer-to-peer protocols. However, without extra restrictions, any number of users at any number or peers might inadvertently or maliciously post any kind of content and rate posts any number of times, thus threatening the value of the protocol. As discussed in the Introdution, Sybil resistance through consensus is a key requirement to combat abuse. In the next section, we propose a consensus mechanism to support public forums in Freechains. Freechains is around 1500 LoC in Kotlin. The

Operation	Effect	Goal
Emission	Old posts award *reps* to author.	Encourage content authoring.
Expense	New posts deduct *reps* from author temporarily.	Discourage excess of content.
Transfer	Likes & dislikes transfer *reps* between authors.	Highlight content of quality. Combat abusive content.

TABLE 2
General reputation operations in public forums.

binary for the JVM is around $6MB$ in size and works in Android and most desktop systems.

3 Reputation and Consensus Mechanism

In the absence of moderation, permissionless peer-to-peer public forums are impractical, mostly because of Sybils abusing the system. For instance, it should take a few seconds to generate thousands of fake identities and SPAM millions of messages into the system. For this reason, we propose a reputation system that works together with a consensus algorithm to resist Sybil attacks.

Section 3.1 describes the overall reputation and consensus mechanism, which can be applied to other systems using DAGs to structure its messages. Section 3.2 describes the concrete rules we implemented for public forums in Freechains. Section 3.3 details the consensus and synchronization algorithm. Section 3.4 simulates the behavior of chains to evaluate the performance of the protocol.

3.1 Overall Design

In the proposed reputation system, users can spend tokens named *reps* to post and rate content in the forums: a `post` initially penalizes authors until it consolidates and counts positively; a `like` is a positive feedback that helps subscribers to distinguish content amid excess; a `dislike` is a negative feedback that revokes content when crossing a threshold. Table 2 summarizes the reputation operations and their goals. To prevent Sybils, users with no *reps* cannot perform these operations, requiring a welcoming like from any other user already in the system. The fact the likes only transfer reputation (being zero-sum operations) eliminates the incentives from malicious to invite Sybils into the system. The only way to generate new *reps* is to post content that other users approve, which demands non-trivial work immune to automation.

Bitcoin employs proof-of-work to mitigate Sybil attacks. However, CPU or other extrinsic resources are not evenly distributed among humans, specially considering that most communications now use battery-powered devices. Considering the context of public forums, we can take advantage of the human authoring ability as an intrinsic resource instead. Creating new content is hard and takes time, but is comparatively easy to verify and rate. Therefore, in order to impose scarcity, we determine that only content authoring generates *reps*, while likes and dislikes just transfer *reps* between users. Nevertheless, scarce operations are not yet sufficient because they demand consensus to establish an order in time across the network to prevent inconsistent operations. As an example, consider a malicious author with

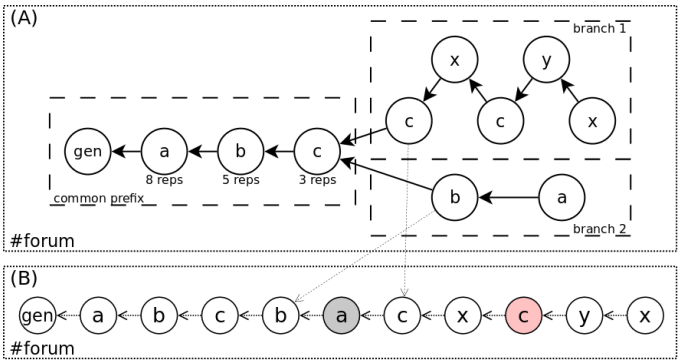

Fig. 2. (A) A public forum DAG with a common prefix and two branches. (B) Total order between blocks of the DAG after consensus.

a single unit of *reps* posting new messages using multiple peers at the same time. According to the *Expense* rule of Table 2, only one of these messages should be accepted. However, without consensus, it is not possible to globally determine which message to accept, since each peer would supposedly accept the first message it sees. Therefore, in order to validate operations consistently, we need the same message ordering across all peers in the network.

Our idea to stablish consensus is to favor DAG forks with posts from users that constitute the majority of the reputation in the network. These forks have more associated work from active users and are analogous to longest chains in Bitcoin. In technical terms, we can adapt a topological sorting algorithm to favor reputation when deciding between branches in the chain DAG. Going back to the malicious user example, the simultaneous messages would appear as forks in the forum DAG. Only the message in the fork with more combined work would be accepted, while all other Sybil messages would be rejected.

Figure 2.A illustrates the consensus criteria. A public forum DAG has a common prefix with signed posts from users a, b, and c. Let's assume that within the prefix, users a and b have contributed with better content and have more reputation combined than c has alone (i.e., $8 + 5 > 3$). After the prefix, the forum forks in two branches: in `branch-1`, only user c remains active and we see that new users x and y (with no previous reputation in the common prefix) generate a lot of new content; in `branch-2`, only users a and b participate but with less activity. Nevertheless, `branch-2` would be ordered first because, before the forking point, a and b have more reputation than c, x, and y combined. User c here might represent a malicious user trying to cultivate fake identities x and y in separate of the network during weeks to accumulate *reps*.

Figure 2.B indicates the resulting consensus order between blocks in the forum. All operations in `branch-2` appear before any operation in `branch-1`. Note that the consensus order exists only for accountability purposes, and is a view of the primary DAG structure. At any point in the consensus timeline, if an operation fails, all remaining blocks in the offending branch are removed from the primary DAG. As an example, suppose that the last post by a (in gray) is a dislike to user c. Then, it's possible that the last post by c (in red), now suppose with *0 reps*, is rejected together with all posts by y and x in sequence. Note that in

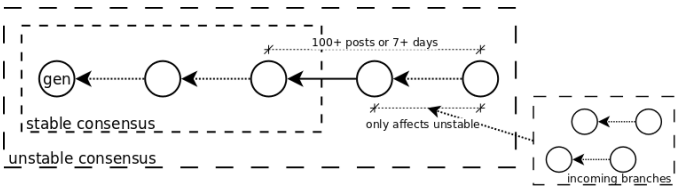

Fig. 3. Stable consensus freezes the order of blocks once they cross the activity threshold. The remaining unstable order may still be affected by incoming branches.

a Merkle DAG, it is not possible to remove only the block with the failing operation, instead, we need to remove the remaining branch completely, as if it never existed. Note also that users in the branch with more reputation can react to attacks even after the fact. For instance, users a and b can pretend that they did not yet see `branch-1` and post extra dislikes to user c from `branch-2` so that a further merge removes all blocks of `branch-1` from the DAG.

There are some other relevant considerations about forks and merges: Peers that first received branches with less reputation will need to reorder all blocks starting at the forking point. This might involve removing content in the end-user software. This behavior is similar to Bitcoin's blockchain reorganization, when a peer detects a new longest chain. Likewise, peers that first saw branches with more reputation just need to put the other branch in sequence with no reordering. This behavior is expected to happen in the majority of the network. Unlike Bitcoin, forks are not only permitted but encouraged due to the local-first software principle. However, the longer a peer remains disconnected, the more conflicting operations it may see, and the higher are the chances of rejection when rejoining.

As a counterpoint to the consensus order in Figure 2.B, maybe users a and b have abandoned the chain for months, and thus `branch-1` is actually legit. In this case, users a and b might be the ones trying to take over the chain. Yet another possibility is that both branches are legit but became disconnected for a long period of time. In any case, it is unacceptable that a very old remote branch affects a long active local chain. For this reason, the consensus algorithm includes an extra constraint that prevents long-lasting local branches to merge, creating *hard forks* in the network. A hard fork occurs when a local branch crosses the predetermined and irreversible thresholds of *7 days* or *100 posts* of activity. In this case, regardless of the remote branch reputation, the local branch takes priority and is ordered first. This situation is analogous to a hard fork in Bitcoin and the branches will never synchronize again. More than simply numeric disputes, hard forks represent social conflicts in which reconciling branches is no longer possible. Figure 3 illustrates hard forks by distinguishing *stable* consensus, which cannot be reordered, from *unstable*, which may still be affected by incoming branches. The activity threshold counts backwards, from the latest local block in the unstable consensus.

In summary, the rules to merge a branch from a remote machine `j` into a branch from a local machine `i` are as follows:

- `i` is ordered first if it crosses the activity threshold of *7 days* or *100 posts*, regardless of `j`.
- `i` or `j` is ordered first, whichever has more reputation in the common prefix.
- otherwise, branches are ordered by an arbitrary criteria, such as lexicographical order of the block hashes immediately after the common prefix.

3.2 Public Forum Chains

We integrated the proposed reputation and consensus mechanism in the public forums of Freechains to support content moderation and mitigate abuse in the chains. Table 3 details the concrete rules we conceived, which are discussed as follows. Authors have to sign their posts in order to be accounted by the reputation system and operate in the chains. The example that follows creates an identity whose public key is assigned as the pioneer in a public chain (prefix # in Table 1):

```
> freechains keys pubpvt 'pioneer-password'
4B56AD.. DA3B5F..  <-- public and private keys
> freechains '#forum' join '4B56AD..'
10AE3E..           <-- hash representing the chain
> freechains '#forum' post --sign='DA3B5F..' \
   'The purpose of this chain is...'
1_CC2184..         <-- hash representing the post
```

The `join` command in rule `1.a` bootstraps a public chain, assigning *30 reps* equally distributed between an arbitrary number of pioneers indicated through their public keys. The pioneers shape the initial culture of the chain with the first posts and likes, while they gradually transfer *reps* to other authors, which also transfer to other authors, expanding the community. The `post` command in sequence is signed by the single pioneer (in this example) and indicates the purpose of the chain for future users.

The most basic concern in public forums is to resist Sybils abusing the chains. Fully peer-to-peer systems cannot rely on logins or CAPTCHAs due to the lack of a central authority. Viable alternatives include (i) building social trust graphs, in which users already in the community vouch for new users, or (ii) imposing explicit costs for new posts, such as proof of work. We propose a mix between trust graphs and economic costs. Rule `4.a` imposes that authors require at least *1 rep* to post, effectively blocking Sybil actions. To vouch for new users, rule `3.a` allows an existing user to like a newbie's post to unblock it, but at the cost of *1 rep*. This cost prevents that malicious members unblock new users indiscriminately, which would be a breach for Sybils. For the same reason, rule `2` imposes a temporary cost of *1 rep* for each new post. Note that the pioneers rule `1.a` solves the chicken-and-egg problem imposed by rule `4.a`: if new authors start with no *reps*, but require *reps* to operate, it is necessary that some authors have initial *reps* to boot the chains.

In the next sequence of commands, a new user joins the same public chain and posts a message, which is welcomed with a like signed by the pioneer:

```
> freechains keys pubpvt 'newbie-password'
503AB5.. 41DDF1..  <-- public and private keys
> freechains '#forum' join '4B56AD..'
10AE3E..           <-- same pioneer as before
> freechains '#forum' post 'Im a newbie...' \
   --sign='41DDF1..'
2_C3A40F..         <-- blocked post
```

Operation	Rule num	Rule name	Description	Observations
Emission	1.a	pioneers	Chain join counts *+30 reps* equally distributed between the pioneers.	`[1.b]` A post takes 24 hours to consolidate. New posts during this period will not be rewarded later. Only after this period, the next post starts to count 24 hours.
	1.b	old post	Consolidated post counts *+1 rep* to author.	
Expense	2	new post	New post counts *-1 rep* to author temporarily.	The discount period varies from 0 to 12 hours and is proportional to the sum of authors' reps in subsequent posts. It is 12 hours with no further activity. It is zero if further active authors concentrate at least 50% of the total reputation in the chain.
Transfer	3.a	like	Like counts *-1 rep* to origin and *+1 rep* to targets.	The origin is the user signing the operation. The targets are the referred post and its corresponding author. If a post has at least 3 dislikes and more dislikes than likes, then its contents are hidden. `[3.b]` Users can revoke their own posts with a single dislike.
	3.b	dislike	Dislike counts *-1 rep* to origin and *-1 rep* to targets.	
Constraints	4.a	min	Author requires at least *+1 rep* to post.	
	4.b	max	Author is limited to at most +30 reps.	
	4.c	size	Post size is limited to at most 128Kb.	

TABLE 3
Reputation rules for public forum chains in Freechains. The chosen constants ($30\ reps$, $24h$, etc) are arbitrary and target typical Internet forums with moderate traffic. A future revision of the protocol could support them as chain parameters.

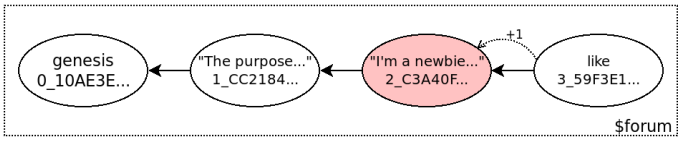

Fig. 4. The `like` approves the newbie message into the `#forum` DAG.

```
> freechains '#forum' like '2_C3A40F..' \
   --sign='DA3B5F..'
3_59F3E1..          <-- hash representing the like
```

Note that chains with the same name but different pioneers are incompatible because the hash of genesis blocks also depend on the pioneers' public keys.

Figure 4 illustrates the chain DAG up to the like operation. The pioneer starts with *30 reps* (rule `1.a`) and posts the initial message. New posts penalize authors with *-1 reps* during at most 12 hours (rule `2`), which depends on the activity succeeding (and including) the new post. The more activity from reputed authors, the less time the discount persists. In the example, since the post is from the pioneer controlling all *reps* in the chain, the penalty falls immediately and she remains with *30 reps*. This mechanism limits the excess of posts in chains dynamically. For instance, in slow technical mailing lists, it is more expensive to post messages in sequence. However, in chats with a lot of active users, the penalty can decrease to zero quickly.

Back to Figure 4, a new user with *0 reps* tried to post a message (hash `C3A40F..`) and is blocked (rule `4.a`), as the red background highlights. But the pioneer liked the blocked message, decreasing herself to *29 reps* and increasing new user to *1 rep* (rule `3.a`). Note that the newbie post is not penalized (rule `2`) because it is followed by the pioneer like, which still controls all *reps* in the chain.

Note that with no additional rules to generate *reps*, the initial *30 reps* would constitute the whole "chain economy" forever. For this reason, rule `1.b` awards authors of new posts with *1 rep*, but only after 24 hours. This rule stimulates content creation and grows the economy of chains. The 24-hour period gives sufficient time for other users to judge the post before awarding the author. It also regulates the growth speed of the chain. In Figure 4, after 1 day, the pioneer would now accumulate *30 reps* and the new user *2 reps*, growing the economy in *2 reps* as result of the two consolidated posts. Note that rule `1.b` awards at most one post of each author at a time. Hence, new posts during the 24-hour period will not award each of them with extra *reps*. Note also that rule `4.b` limits authors to at most *30 reps*, which provides incentives to spend likes and thus decentralize the network.

Likes and dislikes (rules `3.a` and `3.b`) serve three purposes in the chains: (i) welcoming new users, (ii) measuring the quality of posts, and (iii) revoking abusive posts (SPAM, fake news, illegal content, etc). The quality of posts is subjective and is up to users to judge them with likes, dislikes, or simply abstaining. This way, access to chains is permissionless in the sense that the actual peers and identities behind posts are not directly assessed by the protocol, but instead by the other users in the system. The reputation of a given post is the difference between its likes and dislikes, which can be used in end-user software for filtering and highlighting purposes. On the one hand, since *reps* are finite, users need to ponder to avoid indiscriminate expenditure. On the other hand, since *reps* are limited to at most *30 reps* per author (rule `4.b`), users also have incentives to rate content. Hence, these upper and lower limits work together towards the quality of the chains. Note that a dislike shrinks the chain economy since it removes *reps* from both the origin and target. As detailed next, the actual contents of a post may be revoked if it has at least 3 dislikes, and more dislikes than likes (rule `3`). However, considering that *reps* are scarce, dislikes are encouraged to combat abusive behavior, but not to eliminate divergences of opinion.

A post has three possible states: `BLOCKED`, `ACCEPTED`, or `REVOKED`. Figure 5 specifies the transitions between states. If the author has reputation, a new post is immediately

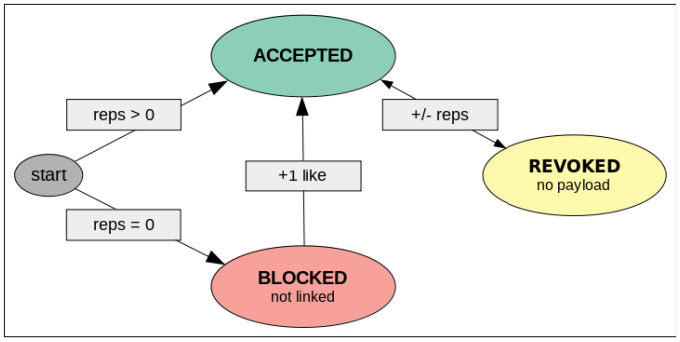

Fig. 5. State machine of posts: `BLOCKED` posts are not linked in the DAG. `ACCEPTED` posts are linked and retransmitted. The payload of `REVOKED` posts are not retransmitted.

`ACCEPTED` in the chain. Otherwise, it is `BLOCKED` and requires a like from another user. Blocked posts are not considered part of the chain DAG in the sense that new posts do not link back to it. In addition, peers are not required to hold blocked posts and neither retransmit them to other peers. However, if blocked posts are not disseminated, new users will never have the chance to be welcomed with a like. A reasonable policy is to hold blocked posts in a temporary bag and retransmit them for some visibility in the network. Rule `4.c` limits the size of posts to at most *128kB* to prevent DDoS attacks using gigantic blocked posts. Once accepted, a post becomes part of the chain and can never be removed again, since Merkle DAGs are immutable by design. However, if the number of dislikes exceeds the threshold (rule `3`), the block becomes `REVOKED` and its payload is not retransmitted to other peers. Note that a block hash does not depend on its associated payload, but only on the payload hash. Hence, it is safe to remove the payload as long as one can prove its revoked state. Later, if the post receives new likes, it means that the payload is still known somewhere and peers can request it when synchronizing again. We consider that revoking posts is fundamental in the context of content publishing, and thus, an important contribution of this work.

In summary, the operations and constraints in Table 3 together with the consensus rules of Figures 2 and 3 empower Freechains with permissionless public forums.

3.3 The Consensus and Synchronization Algorithm

In order to validate publications consistently across the network, the consensus algorithm needs to execute at the core of the protocol, and at the time peers synchronize their chain DAGs. From the perspective of a peer on the receiving side, the overview of the consensus and synchronization algorithm is as follows:

a) **Consensus:** Calculate the consensus order of blocks considering the local DAG.
b) **Synchronization:** Receive the next missing block respecting topological order.
c) **Verification:** Verify that the block is valid and add it to the local DAG.
d) **Repeat:** Restart from step (a) until no blocks are missing.

Step (a) is discussed and motivated in extent in the previous sections. The actual algorithm is detailed further and adapts topological sorting to favor branches with more reputation.

Step (b) is very similar to a recent work on *Byzantine Causal Broadcast* [18], which also synchronizes DAGs representing causal messages: (i) starting from the heads of the DAGs; (ii) traverse each head backwards until a common block is found; (iii) now traverse forward all missing blocks, respecting topological order; (iv) each block represents an iteration of Step (b). What distinguishes our algorithm is that `BLOCKED` blocks are not synchronized, which mitigates denial of service attacks with very long malicious branches. We also limit the number of iterations on Step (b.ii) to ensure that the algorithm terminates. If necessary, a remote peer that is disconnected for too long can try older heads to adapt to this constraint.

Step (c) verifies if the next missing block found in Step (b) can be added to the local DAG considering the consensus order found in Step (a). It verifies the following properties: (i) Merkle DAG structure is consistent (e.g., block hash and back pointers); (ii) Back pointers are not in *BLOCKED* state; (iii) Author signing the block has enough reputation. In case of success, the block is added to the DAG and the next iteration of the consensus algorithm already considers it. Otherwise, the block is marked as *BLOCKED*, and the remaining branch is ignored in Step (b.iii).

The topological sorting for the Consensus Step (a) is an adaptation of Kahn's algorithm [19]:

a) The set S of all blocks in the DAG with no incoming edges starts with the chain genesis.
b) Removes from S the block B in the branch with more reputation (Figure 2) and adds it to the end of the consensus list L. (This is the only step that differs from Kahn's algorithm, which would take any block from S.)
c) Adds to S all blocks directly in front of B, excluding those reachable from any node still in S.
d) When S is empty, the list L holds the consensus order.

Step (b) requires to find the maximum value in set S according to the reputation criteria. The comparison function needs to find the non-common authors in the suffix of the two branches in order to compare the sum of their reputations. For this reason, the algorithm also needs to track negative, neutral, and positive blocks according to Table 3. We take advantage of the hard fork rule that freezes the consensus list L when crossing the activity threshold (stable consensus in Figure 3). We use it as a checkpoint to cache the reputations, which limits the input size to a short fixed size that does not depend on arbitrary chain sizes.

Currently, we hold the DAG structure directly in the file system with no database support. Each chain uses a separate directory, and each block uses two separate files: a *JSON* for the metadata and a payload exactly as posted. We save the cache and current consensus order in an index file. We build the DAG by traversing the chain directory.

3.4 Experiments

In this section, we perform experiments to evaluate the performance and practicability of the protocol. As detailed further, we evaluate the following parameters: (a) metadata overhead, (b) consensus runtime, (c) graph forks, and (d) blocked messages.

We simulate the behavior of two publicly available forums as if they were using Freechains: a chat channel from the Wikimedia Foundation[1], and the *comp.compilers* newsgroup[2]. Chats and newsgroups represent typical public forums with faster interactions with shorter payloads (chats), and slower interactions with larger payloads (newsgroups). We only simulate the first *10.000* messages of the forums, which represent 3 months of activity in the chat and 9 years in the newsgroup.

A simulation spawns N peers, each joining the same chain with the same arguments. For each message in the original forum, we (i) set the timestamp of all peers to match the date, (ii) create a pair of keys if the author is new, (iii) post the message from a random peer in N, (iv) like the post if the author has no reputation, and (v) synchronize the chain with M random peers. Since all messages are part of the original archive, we always perform step (iv) to unblock messages from newbies. Evaluation parameter (d) counts extra likes to unblock existing users, which could signal excessive bookkeeping in the chain. Step (v) will inevitably create forks in the chain for any `M<N`, which is accounted by evaluation parameter (c).

For the newsgroup, we use `N=15` and `M=5`, which represents a larger number of peers with few interconnections to stress the local-first nature of the protocol. For the chat, we use `N=5` and `M=3`, which represents a smaller number of peers with more interconnections. We executed each simulation 4 times in a conventional desktop PC (`i7` CPU, `8GB` RAM, `512GB` SSD). Since the variations were negligible, we always discuss the median measures.

Evaluation item (a) measures the protocol overhead due to blockchain metadata, which consists of a timestamp, author signature, and hashes (block id, payload, backlinks, and likes). The original chat archive is `800kB` in size, or `80B` for each message, which includes a timestamp, an username, and the actual payload. The simulated chat chain is `8MB` in size, which indicates a $10x$ overhead. The original newsgroup is `30MB` in size, or `3kB` for each message, which includes a timestamp, a sender, a subject, and the actual payload (typically much longer). The simulated newsgroup chain is `42MB` in size, which indicates a 50% overhead. It is clear that the metatada overhead is not negligible, specially for short chat messages, but decreases as the payload increases.

Item (b) evaluates the consensus step depicted in Section 3.3. We measured the time to sort blocks in a local DAG both for the first time (without any caches), and incrementally (from stable consensus caches). For the chat chain, it takes `125s` and `50ms` for the initial and incremental sorts respectively, while for the newsgroup chain, it takes `100s` and `70ms`. The incremental sort is limited to 7 days or 100 posts, regardless of the size of the chain, which conveniently settles an upper bound on the input size of the consensus algorithm. Given that we use plain JSON files in the file system, we consider an incremental consensus under `100ms` to be a practical upper bound.

Item (c) evaluates the number of forks in chain DAGs, which indicates the level of asynchrony between peers following the local-first principle (supposedly). We calculate the ratio of forks over the total number of messages, e.g., a DAG with 100 messages and 10 forks has a ratio of 10%. We found a ratio of 18% for the chat and 14% for the newsgroup, which confirms that the simulation achieves a reasonable level of asynchrony.

Item (d) evaluates how much bookkeeping is required to sustain active users in the forums. Even though the newbie rule `4.a` in Table 3 is key to combat Sybils, ideally it should not deny access to active users with low reputation recurrently. The evaluation counts the number of blocked messages requiring extra likes after the welcoming likes (which are disconsidered) and calculates the ratio over the total number of messages in the chain. As an example, if 10 users posted 110 messages requiring 20 likes, we discount the 10 initial messages and welcoming likes and find a ratio of 10% (`(20-10)/(110-10)`). For the chat with 80 users, we found a ratio of 3.7%. For the newsgroup with 5000 users, we found a ratio of 3.5%. Considering that users were not aware of the reputation rules, the low ratios indicate that their "natural" posting behavior matches the rules constraints. We assume that users would use the revoke mechanism to combat abusive content, having no further effects on our evaluation.

1. Chat: https://archive.org/download/WikimediaIrcLogs/
2. Newsgroup: https://archive.org/download/usenet-comp

4 Correspondences with CRDTs

Conflict-free replicated data types (CRDTs) [13] serve as a robust foundation to model concurrent updates in collaborative local-first applications [17]. However, CRDTs are not a panacea and often require human intervention to solve specific conflicts (*quasi-CRDTs*), such as manual merges in version control systems (VCSs). Our observation is that, since the proposed consensus algorithm already relies on human interactions, we can use it to resolve conflicts automatically.

We propose a three-layered CRDT scheme to build decentralized collaborative applications: state-based CRDTs at transport layer (CvRDTs), operation-based CRDTs at application layer (CmRDTs), and quasi-CRDTs with arbitrary operations after consensus is applied.

At the transport layer, Merkle DAG chains are trivial CvRDTs because they converge to the same state on synchronization [20]. At the application layer, however, DAGs loose this property because branches might be processed in different orders across peers, resulting in incompatible states. An interesting approach is to require blocks to represent commutative operations, thus resulting in CmRDTs [20]. CmRDTs have the advantage to store only small updates that are sufficient to reconstruct any version of the data. At the third layer, the consensus algorithm transforms a chain DAG into a totally-ordered set, which supports quasi-CRDTs with non-commutative operations. Note that even full-CRDTs can benefit from consensus, since they

often encounter corner cases that require arbitrary choices (e.g., aggregating updates or ressurecting data [21]).

Next, we illustrate this three-layered CRDT scheme through an example of a simple permissionless decentralized VCS (dVCS) implemented on top of public chains.

4.1 A dVCS with Automatic Merges

We built a simple dVCS with the functionalities (and associated Freechains operations) as follows:

- initialize a repository (`join`)
- commit local changes to repository (`post`)
- checkout repository changes to local (`consensus/get`)
- synchronize with remote peer (`send/recv`)
- rate and revoke commits (`like/dislike`)

The system relies on public forum chains, which means that repositories are permissionless and adhere to the proposed reputation and consensus mechanism. The main innovations in this system are that (i) users can rate commits to reject them, and (ii) checkout operations resolve conflicts automatically based on consensus order.

As a concrete example, we model a Wiki article as a chain behaving as a VCS holding its full edition history. To model a complete Wiki platform, we assume that an article could refer to other articles using hyperlinks to other chains.

Most VCS operations, except `commit` and `checkout`, map directly to single Freechains commands. For instance, to create a repository, we simply join a public chain with the name of the file we want to track. In the commands that follow, we create a repository with multiple pioneers, and then edit and commit the file twice:

```
> freechains '#p2p.md' join A2885F.. 2B9C32..
2F11BF..
> echo "P2p networking is..." > p2p.md
> freechains-vcs '#p2p.md' commit --sign=69929..
1_4F3EE1..
> echo "The [USENET](#usenet.md), ..." >> p2p.md
> freechains-vcs '#p2p.md' commit --sign=69929..
2_B58D22..
```

The `commit` operation expands as follows:

```
> freechains-vcs '#p2p.md' checkout p2p.remote
> diff p2p.remote p2p.md > p2p.patch
> freechains '#p2p.md' post p2p.patch --sign=69929..
```

A commit first makes a checkout from the repository into temporary file `p2p.remote`. Then, it compares this version against our local changes, saving the diffs into file `p2p.patch`. A patch contains the minimal set of changes to apply back into the repository, and represents a CmRDT operation in our model. Finally, the commit posts the patch file back into the chain. Evidently, the chain name and signature (`#p2p.md` and `--sign=...`) are parameters in the actual implementation of `commit`. The checkout operation is expanded further.

Much time later, the other pioneer synchronizes with us, checks out the file, and then edits and commits it back:

```
> freechains '#p2p.md' recv '<our-ip>'
2/2    <-- two commits above
> freechains-vcs '#p2p.md' checkout p2p.md
> echo "P2P does not scale!" >> p2p.md
> freechains-vcs '#p2p.md' commit --sign=320B5..
3_AE3A1B..
```

A `checkout` recreates the latest version of the file in the repository by applying all patches since the genesis block. Recall that each patch represents a CmRDT operation to recreate the final state of the data. The `checkout` expands as follows:

```
> rm p2p.md && touch p2p.md
> for blk in `freechains '#p2p.md' consensus` do
   freechains '#p2p.md' get payload $blk > p2p.patch
   patch p2p.md p2p.patch
   if [ $? != 0 ]; then
    echo $blk  <-- hash of failing patch
    break      <-- ignore remaining patches
   fi
 done
```

The `consensus` operation of Freechains returns all hashes since the genesis block, respecting the consensus order. The loop reads each of the payloads representing the patches and apply them in order to recreate the file. If any of the patches fail, the command exhibits the hash of the offending block and terminates. We discuss this behavior further, when we illustrate commit conflicts.

Since the last commit above is clearly wrong (P2P networks do scale!), other users in the network will dislike it until the block becomes `REVOKED` in the chain:

```
> freechains '#p2p.md' dislike 3_AE3A1B.. --sign=USR1
> freechains '#p2p.md' dislike 3_AE3A1B.. --sign=USR2
> freechains '#p2p.md' dislike 3_AE3A1B.. --sign=USR3
> freechains-vcs '#p2p.md' checkout p2p.md
> cat p2p.md
P2p networking is...
The [USENET](#usenet.md), ...  <-- no 3rd line
```

This way, the checkout operation above will apply an empty patch associated with the revoked block, effectively removing the wrong line from the file. This mechanism illustrates how the reputation system enables collaborative permissionless edition and curation.

Next, we create a conflicting situation in which two authors edit and commit the same line of the file concurrently:

```
# PEER A (more reputation):
> sed -i 's/P2p/P2P/g' p2p.md    <-- fix typo
> freechains-vcs '#p2p.md' commit --sign=69929..
4_A..

# PEER B (less reputation):
> sed -i 's/networking/computing/g' p2p.md
> freechains-vcs '#p2p.md' commit --sign=320B5..
4_B..

# SYNCHRONIZE (exchange conflicting forks):
> freechains '#p2p.md' recv '<our-ip>'
1 / 1
> freechains '#p2p.md' send '<our-ip>'
1 / 1
> freechains-vcs '#p2p.md' checkout p2p.md
1 hunk FAILED -- saving rejects to file p2p.md.rej
4_B..
> cat p2p.md
P2P networking is... <-- typo fixed (not computing)
The [USENET](#usenet.md), ...
```

After they commit the conflicting changes, the peers synchronize in both directions and reach the state of Figure 6. When we checkout the file, the patches are applied respecting the consensus order. As a result, we see that the first branch is applied, but not the second, leaving the file in the longest possible consistent state. This happens because of the `break` in the checkout operation: once a conflict is

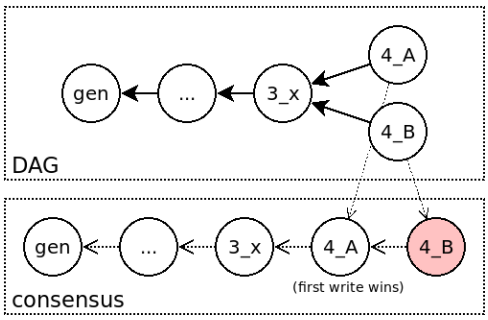

Fig. 6. The branches in the DAG are ordered by reputation. Only the first patch is applied successfully (first write wins).

found, no further patches apply in any of the remaining branches. We chose to adopt a *first write wins* resolution to favor work in the branches with more reputation. 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.

4.2 Discussion

In summary, the proposed reputation and consensus mechanism empowers a simple dVCS with cooperative authoring and automatic conflict resolution. It only requires the standard *diff & patch* tools and the basic API of Freechains.

We apply the proposed three-layered CRDT scheme as follows: The first layer transports the whole commit history of small patches as a CvRDT between peers, which eventually reach the same DAG state. At this layer, the DAG is just raw data with no attached semantics. In the second layer, peers need to interpret the DAG as a CmRDT of small editions to recreate the file. The *patch* tool is mostly commutative, except when branches modify the same lines. Hence, in these situations, we resort to the third layer with the consensus order, and apply the patches sequentially until a conflict occurs. The final state of the file is guaranteed to be consistent, i.e., the result of a sequence of correct patch applications.

Towards richer decentralized collaborative applications, we can employ CRDTs to model data other than raw text. As an example, *Automerge* [21] manipulates JSON objects, which supports non-trivial datasets with robust merging policies, but which could still benefit from consensus to resolve corner cases.

5 Related Work

Many other systems have been proposed for decentralized content sharing [4], [9]. Here we consider the classes of publish-subscribe, federated, and peer-to-peer protocols.

5.1 Publish-Subscribe Protocols

Decentralized topic-based publish-subscribe protocols, such as *XMPP* [22], *ActivityPub* [23], and *gossipsub* [24], decouples publishers from subscribers in the network. A key limitation of *pubsubs* is that the brokers that mediate communication still have a special role in the network, such as authenticating and validating posts. Nevertheless, some pubsubs do not rely on server roles, and instead, use peer-to-peer gossip dissemination [25], [26], [27], [28], [24], [26]. Most of these protocols focus on techniques to achieve scalability and performance, such as throughput, load balancing, and real-time relaying.

However, these techniques alone are not sufficient to operate permissionless networks with malicious Sybils [29]. Being generic protocols, pubsubs are typically unaware of the applications built on top of them. In contrast, as stated in Section 2, the pubsub of Freechains is conceptually at the application level and is integrated with the semantics of chains, which already verifies blocks at publishing time. For instance, to flood the network with posts, malicious peers need to spend reputation, which takes hours to recharge (rule 2 in Table 3). In addition, blocked posts (Figure 5) are not a concern either, because they have limited reachability. Another advantage of a tighter integration between the application and protocol is that Merkle DAGs simplify synchronization, provide persistence, and prevent duplication of messages. Full persistence resists long churn periods, and de-duplication tolerates CmRDTs with operations that are not idempotent.

In summary, Freechains and *pubsub* middlewares operate at different network layers, which suggests that Freechains could benefit of the latter to manage the interconnections between peers.

5.2 Federated Protocols

Federated protocols, such as e-mail, allow users from one domain to exchange messages with users of other domains seamlessly. *Diaspora*, *Matrix*, and *Mastodon* are more recent federations for social media, chat, and microblogging [9], respectively.

As a drawback, identities in federations are not portable across domains, which may become a problem when servers shutdown or users become unsatisfied with the service [30]. In any of these cases, users have to grab their content, move to another server, and announce a new identity to followers.

Moderation is also a major concern in federations [9]. As an example, messages crossing domain boundaries may be subject to different policies that might affect delivery. With no coordinated consensus, it is difficult to make pervasive public forums practical. For this reason, Matrix supports a permissioned moderation system[3], but which applies only within clients, after the messages have already been flooded in the network.

As a counterpoint, federated protocols seem to be more appropriate for real-time applications such as large chats rooms. The number of hops and header overhead can be much smaller in client-server architectures compared to peer-to-peer systems, which typically include message signing, hash linking, and extra verification rules.

3. Matrix moderation: https://matrix.org/docs/guides/moderation

5.3 Peer-to-Peer Protocols

Bitcoin [11] is probably the most successful permissionless network, but serves specifically for electronic cash. IPFS [15] and Dat [31] are data-centric protocols for hosting large files and applications, respectively. Scuttlebutt [10] and Aether [9] are closer to Freechains goals and cover human-centric $1{\rightarrow}N$ and $N{\leftrightarrow}N$ public communication, respectively.

Bitcoin adopts proof-of-work to achieve consensus, which does not solve the centralization issue entirely, given the high costs of equipment and energy. Proof-of-stake is a prominent alternative [32] 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. In contrast, we chose an intrinsic resource, which is authored content in the chains themselves. We believe that human work grows more linearly with effort and is not directly portable across chains with different topics. These hypotheses support the intended decentralization of our system. Another distinction is that generic public ledgers require permanent connectivity to avoid forks, which opposes our local-first principle. This is because a token transaction only has value as part of the longest chain. This is not the case for a local message exchange between friends, which has value in itself.

IPFS [15] is centered around immutable content-addressed data, while Dat [31] 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, which are optimal to serve large and popular content, but not for search and discovery. 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. On the one hand, DHTs are probably not the best architecture to model decentralized human communication with continuous feed updates. On the other hand, replicating large files across the network in Merkle DAGs is also impractical. An alternative is to use DHT links in Merkle payloads to benefit from both architectures.

Scuttlebutt [10] 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. Scuttlebutt is aligned to $1{\rightarrow}N$ public identity chains of Freechains in Table 1. For group $N{\leftrightarrow}N$ communication, 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.

Aether [9] provides peer-to-peer public communities aligned with $N{\leftrightarrow}N$ public forums of Freechains. A fundamental difference is that Aether is designed for ephemeral, mutable posts with no intention to enforce global consensus across peers. 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. In contrast, Freechains relies on its permissionless reputation and consensus mechanisms for moderation.

Regarding the structure of messages, append-only Merkle DAGs have been proposed as self-certified archives, and as CRDTs that provide strong eventual consistency [16], [5]. However, when these DAGs are open to permissionless writes, they are subject to abuse, which may degrade the content and performance of the network. Another aspect is that a DAG itself has no attached semantics, which weakens its consistency property at the application level, which may interpret the DAG in conflicting orders. The proposed consensus mechanism addresses permissionless writes and provides a total order for applications (at the cost of rollbacks).

A similar dVCS have been recently proposed [7], with a DAG representation and merging policies. To resolve *competition conflicts*, they propose to use an external scoring function, such as users' reputations. Our proposed consensus mechanism internalizes such reputation scoring function. Integrated reputation also prevents SPAM and abusive behavior from byzantine nodes, which could otherwise generate very large graphs that take forever to synchronize (as discussed for Byzantine Causal Broadcast [18]).

6 Conclusion

In this paper, we propose a permissionless consensus and reputation mechanism for content sharing in peer-to-peer networks. We enumerate four main contributions: (i) human authored content as a scarce resource (*proof-of-authoring*); (ii) diversified public forums, each as an independent blockchain with subjective moderation rules; (iii) abusive content removal preserving data integrity; and (iv) three-layered CRDTs to build collaborative applications.

The key insight of the consensus mechanism is to use the human authoring ability as a scarce resource to determine consensus. This contrasts with extrinsic resources, such as CPU power, which are dispendious and not evenly distributed among people. Consensus is backed by a reputation system in which users can rate posts with likes and dislikes, which transfer reputation between them. The only way to forge reputation is by authoring new content under the judgement of other users. This way, reputation generation is expensive, while verification is cheap and decentralized.

The reputation and consensus mechanism is integrated into Freechains, a peer-to-peer protocol that offers multiple arrangements of public and private communications. Users have the power to create public forums of interest and apply diverse moderation policies. In particular, users can

revoke content considered abusive according to the majority, not depending on centralized authorities. We simulate the behavior of existing chat and newsgroup forums as if they were using Freechains to show the practicability of the protocol as a descentralized alternative for public forums.

On top of the proposed three-layered CRDT architecture, we prototyped a decentralized version control system that resolves commit conflicts automatically: (i) at the transport layer, the full commit history is held in a Merkle DAG counting as a CvRDT; (ii) at the application layer, the commit patches operate as (mostly) commutative CmRDT operations; and (iii) after consensus is applied, merge conflicts are resolved automatically.

Finally, we do not claim that the proposed reputation system enforces "good" human behavior in any way. Instead, it provides a transparent and quantitative mechanism to help users understand the evolution of forums and act accordingly. Human creativity contrasts with plain economic resources (e.g., proof-of-work), which do not appraise social interactions and also tend to concentrate over the time.

REFERENCES

[1] J. Zittrain, "Fixing the internet," vol. 362, no. 6417. American Association for the Advancement of Science, 2018, pp. 871–871.

[2] N. Masinde and K. Graffi, "Peer-to-peer-based social networks: A comprehensive survey," *SN Computer Science*, vol. 1, no. 5, pp. 1–51, 2020.

[3] D. Koll, J. Li, and X. Fu, "The good left undone: Advances and challenges in decentralizing online social networks," *Computer Communications*, vol. 108, pp. 36–51, 2017.

[4] S. A. Theotokis and D. Spinellis, "A survey of peer-to-peer content distribution technologies," *ACM Comput. Surv.*, Dec. 2004.

[5] M. Kleppmann, "Making crdts byzantine fault tolerant," in *Proceedings of the 9th Workshop on Principles and Practice of Consistency for Distributed Data*, 2022, pp. 8–15.

[6] C. Sun, X. Jia, Y. Zhang, Y. Yang, and D. Chen, "Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems," *ACM Transactions on Computer-Human Interaction (TOCHI)*, vol. 5, no. 1, pp. 63–108, 1998.

[7] B. Nasrulin and J. Pouwelse, "Decentralized collaborative version control," in *Proceedings of the 2nd International Workshop on Distributed Infrastructure for Common Good*, 2021, pp. 11–16.

[8] J. R. Douceur, "The sybil attack," in *International workshop on peer-to-peer systems*. Springer, 2002, pp. 251–260.

[9] J. Graber, "Decentralized social ecosystem review," BlueSky, Tech. Rep., 2021.

[10] D. Tarr *et al.*, "Secure scuttlebutt: An identity-centric protocol for subjective and decentralized applications," in *Proceedings of ACM ICN'19*, 2019, pp. 1–11.

[11] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," Tech. Rep., 2009.

[12] F. Sant'Anna, F. Bosisio, and L. Pires, "Freechains: Disseminação de conteúdo peer-to-peer," in *Workshop on Tools, SBSeg'20*.

[13] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-free replicated data types," in *Symposium on Self-Stabilizing Systems*. Springer, 2011, pp. 386–400.

[14] V. B. Gomes, M. Kleppmann, D. P. Mulligan, and A. R. Beresford, "Verifying strong eventual consistency in distributed systems," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 1–28, 2017.

[15] J. Benet, "Ipfs-content addressed, versioned, p2p file system," *arXiv preprint arXiv:1407.3561*, 2014.

[16] F. Jacob, C. Beer, N. Henze, and H. Hartenstein, "Analysis of the matrix event graph replicated data type," *IEEE access*, vol. 9, pp. 28 317–28 333, 2021.

[17] M. Kleppmann, A. Wiggins, P. van Hardenberg, and M. McGranaghan, "Local-first software: you own your data, in spite of the cloud," in *Proceedings of Onward'19*, 2019, pp. 154–178.

[18] M. Kleppmann and H. Howard, "Byzantine eventual consistency and the fundamental limits of peer-to-peer databases," *arXiv preprint arXiv:2012.00472*, 2020.

[19] A. B. Kahn, "Topological sorting of large networks," *Communications of the ACM*, vol. 5, no. 11, pp. 558–562, 1962.

[20] H. Sanjuan, S. Poyhtari, P. Teixeira, and I. Psaras, "Merkle-crdts: Merkle-dags meet crdts," *arXiv preprint arXiv:2004.00107*, 2020.

[21] M. Kleppmann and A. R. Beresford, "A conflict-free replicated json datatype," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 10, pp. 2733–2746, 2017.

[22] P. Saint-Andre, K. Smith, R. Tronçon, and R. Troncon, *XMPP: the definitive guide*. " O'Reilly Media, Inc.", 2009.

[23] C. Webber, J. Tallon, and O. Shepherd, "Activitypub," *W3C Recommendation, W3C, Jan*, 2018.

[24] D. Vyzovitis and Y. Psaras, "Gossipsub: A secure pubsub protocol for unstructured, decentralised p2p overlays," Protocol Labs, Tech. Rep., 2019.

[25] R. Baldoni, R. Beraldi, V. Quema, L. Querzoni, and S. Tucci-Piergiovanni, "TERA: Topic-Based Event Routing for Peer-to-Peer Architectures," in *Proceedings of the International Conference on Distributed Event-Based Systems*, 2007, pp. 2–13.

[26] J. A. Patel, É. Rivière, I. Gupta, and A.-M. Kermarrec, "Rappel: Exploiting interest and network locality to improve fairness in publish-subscribe systems," *Computer Networks*, vol. 53, no. 13, pp. 2304–2320, 2009.

[27] M. Matos, A. Nunes, R. Oliveira, and J. Pereira, "Stan: exploiting shared interests without disclosing them in gossip-based publish/subscribe." in *IPTPS*, 2010, p. 9.

[28] F. Rahimian, S. Girdzijauskas, A. H. Payberah, and S. Haridi, "Vitis: A gossip-based hybrid overlay for internet-scale publish/subscribe enabling rendezvous routing in unstructured overlay networks," in *2011 IEEE International Parallel & Distributed Processing Symposium*. IEEE, 2011, pp. 746–757.

[29] D. Vyzovitis, Y. Napora, D. McCormick, D. Dias, and Y. Psaras, "Gossipsub: Attack-resilient message propagation in the filecoin and eth2. 0 networks," Protocol Labs, Tech. Rep., 2020.

[30] A. Auvolat, "Making federated networks more distributed," in *2019 38th Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 2019, pp. 383–3831.

[31] D. C. Robinson, J. A. Hand, M. B. Madsen, and K. R. McKelvey, "The dat project, an open and decentralized research data tool," *Scientific data*, vol. 5, no. 1, pp. 1–4, 2018.

[32] L. M. Bach, B. Mihaljevic, and M. Zagar, "Comparative analysis of blockchain consensus algorithms," in *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. IEEE, 2018, pp. 1545–1550.

Francisco Sant'Anna received his PhD degree in Computer Science from PUC-Rio, Brazil in 2013. In 2016, he joined the Faculty of Computer Science at the Rio de Janeiro State University, Brazil. His research interests include Programming Languages and Concurrent & Distributed Systems.