Peer-to-Peer Consensus via Authoring Reputation

Abstract—Content publishing in public Internet forums suffers from excess and abuse, such as SPAM and fake news. Centralized platforms employ filtering algorithms and anti-abuse policies, but impose full trust from users. We propose a publish-subscribe peer-to-peer protocol to model content dissemination without centralized control. The protocol prevents Sybil attacks with a reputation system that moderates content and, at the same time, delivers network consensus. We trace a parallel with Bitcoin: posts create reputation (vs proof-of-work), likes and dislikes transfer reputation (vs transactions), and aggregate reputation determines consensus (vs longest chain). The reputation system and resulting consensus depends exclusively on human work to create and rate content. We prototype a simple permissionless distributed version control system that relies on reputation consensus to resolve conflicts automatically.

Index Terms—Bitcoin, CRDT, distributed consensus, peer-to-peer, publish-subscribe, reputation system, version control system

1 Introduction

CONTENT publishing in public Internet forums and social media platforms is increasingly more centralized in a few companies [1], [2]. 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, since they collect and control our data, "algorithmize" our consumption, and yet obstruct portability with proprietary standards. Peer-to-peer alternatives [3] eliminate intermediaries and push to end users the responsibility to manage data and connectivity. However, due to decentralization of authority and infrastructure, new challenges arise to enforce overall state consistency while dealing with malicious users.

In an ideal Internet forum, all messages or posts (i) reach even temporarily disconnected users; (ii) are delivered in a consistent order; (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 conversations in peers until they reach all users [3]. However, this approach does not guarantee consensus since posts can be received in conflicting orders [4].

Bitcoin [5] proposes a permissionless consensus protocol founded on scarce virtual assets, the *bitcoin tokens*. The only way to create new bitcoins is to work towards consensus in the network by proposing a total order among transactions in the system. This way, Bitcoin prevents double spending [5], which is analogous to conflicting message delivery in public discussions. However, Bitcoin just supports transfers between users, with no subjective judgment that could affect the actual transactions. In contrast, our goal is to use social interactions between humans to evaluate content and mitigate abuse at the same time.

In this work, we propose a permissionless consensus algorithm based on authoring reputation. Inspired by Bitcoin, authors accumulate tokens named *reps*, which serve as currency to rate posts in the forums. Users can rate posts with likes and dislikes, which transfer *reps* between them. Work is manifested as new posts which, if accepted by others, reward authors with *reps*. This way, like Bitcoin, token generation is expensive, while verification is cheap and made by multiple users. However, unlike Bitcoin, both creation and verification are subjective, based on human creativity and judgement, which match our target domain of content publishing. Posts and likes are linked as blocks in a Merkle DAG that persists the whole conversation and is disseminated in the network with gossiping. To reach consensus, the DAG is ordered by branches with more reputed authors, which contributed with more work to the forum. The resulting list is then verified for conflicting operations, such as likes with insufficient *reps*, which is equivalent to double spending in Bitcoin. In this case, the branch that causes the conflict is removed from the DAG. We integrated the proposed consensus algorithm into Freechains, a peer-to-peer publish-subscribe content dissemination protocol [6]. We also prototyped a permissionless distributed version control system (DVCS) that relies on consensus to apply automatic merges.

Our main contribution is to make public forums with complete decentralization viable in practice. The proposed reputation and consensus mechanism depends exclusively on human work, contrasting with most systems that rely on extrinsic resources, such as CPU power. The general idea of the algorithm can be applied to any system that uses DAGs to structure its messages. As a derived contribution, the consensus implies a total order among messages, which backs the use of CRDTs [7] in collaborative authoring platforms, such as a DVCS.

As a main limitation, Merkle DAGs are ever growing data structures that also carry considerable metadata overhead. In addition, the required per-block validation may become a bottleneck for real-time applications, such as chats and video calls. Finally, we do not claim that the proposed

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 even 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

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

reputation system enforces "good" human behavior in any way. Instead, it provides a transparent and quantitative mechanism that helps users understand the progress of forums and act accordingly.

In Section 2, we introduce the basic functionalities of Freechains to create, rate, and disseminate posts. In Section 3, we describe the general reputation and consensus mechanism, and show how it integrates with public forums in Freechains. In Section 4, we discuss the some correspondences with CRDTs and prototype a simple DVCS. In Section 5, we compare our system with other publish-subscribe protocols, federated applications, and fully peer-to-peer systems. In Section 6, we conclude this work.

2 FREECHAINS

Freechains is an unstructured peer-to-peer topic-based publish-subscribe system, in which each topic or *chain* is a replicated *Merkle DAG* [6]. This way, as an author posts to a chain, other users subscribed to the same chain eventually receive the message. Freechains supports multiple arrangements of public and private communication, which are detailed in Figure 1. In this section, we operate a private group to describe the basic behavior of chains. At the end of the section, we also exemplify a public identity chain. In Section 3, we detail the behavior of public forums, which involve untrusted communication between users and require the proposed reputation and consensus mechanism.

All Freechains operations go through a *daemon* (analogous to Bitcoin full nodes) which validates posts, links them in the Merkle DAGs, persists the chains in the disk, and communicates with other peers to disseminate the graphs. The command that follows starts a daemon 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 key shared 'strong-password' # (i)
A6135D..   <- returned shared key
> freechains '$family' join 'A6135D..'       # (ii)
42209B..   <- hash of chain
> freechains '$family' post 'Good morning!'  # (iii)
1_EF5DE3.. <- hash of 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 2.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. As expected from a Merkle DAG, the hash of a block depends on its payload and hash of previous block.

Freechains adheres to the *local-first* software principle [8], allowing networked applications to work locally while offline. Except for synchronization, all other operations in the system affect only 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, others peers have to initialize the example chain with the same steps:

```
> freechains-daemon start '/var/freechains/'
> freechains key shared 'strong-password'
A6135D..
> freechains '$family' join 'A6135D..'
42209B..
```

Synchronization is explicit, in pairs, and unidirectional. The command *recv* asks the daemon in *localhost* to connect to daemon in *remote-ip* and receive all missing blocks from there:

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

Now, the new peer is in the same state as the original peer in Figure 2.A. The complementary command *send* would synchronize the DAGs in the other direction. Note that Freechains does not construct a network topology nor synchronizes 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, the protocol only gives basic support for communication in pairs of peers and further automation requires external tools.

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:

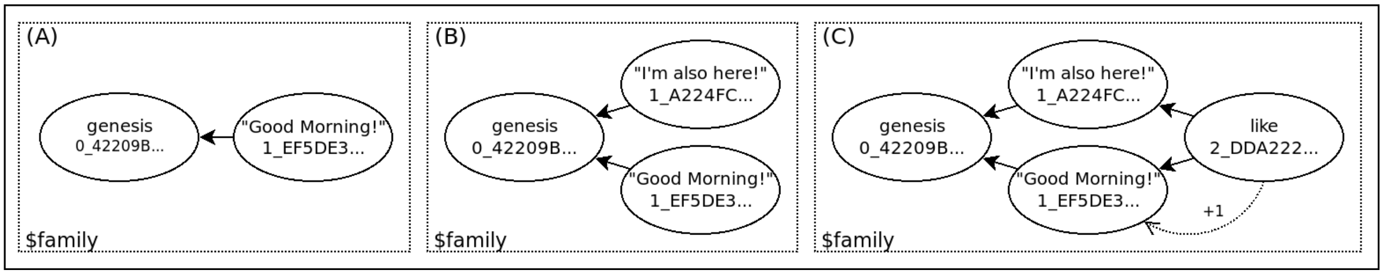

Fig. 2. Three DAG configurations. (A) Single head pointing to genesis block. (B) Fork with heads pointing to genesis block. (C) Like pointing to previous heads and also to its target.

```
> freechains '$family' heads
1_EF5DE3..
> freechains '$family' payload '1_EF5DE3..'
Good morning!
```

However, since the network is inherently concurrent and users are encouraged to work locally, typical graphs are not lists, but DAGs with multiple heads. As an example, suppose the new peer posted a message before the *recv* above, when the local DAG was still in its genesis state. In this case, as illustrated in Figure 2.B, the resulting graph after the synchronization 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 can apply simple methods, such as relying on the timestamps of blocks. However, in public forums, a malicious user could modify his local time to manipulate the order of messages.

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

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

As illustrated in Figure 2.C, a like is a regular block with an extra link to its target. In private groups, likes are unlimited and behave much like typical centralized systems. In public forums, however, likes are restricted, 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 key pubpvt 'other-password'
EB172E.. 96700A..   <- public and private keys
> freechains '@EB172E..' join
F4EE21..
> freechains '@EB172E..' post 'This is Oprah' \
   --sign='96700A..'
1_547A2D..
```

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

Freechains is around 1500 LoC in Kotlin and is publicly available[1]. The binary for the JVM is less than $6Mb$ in size and works in Android and most desktop systems.

1. http://www.freechains.org

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.
Tranfer	Likes & dislikes transfer *reps* between authors.	Highlight content of quality. Combat abusive content.

Fig. 3. General reputation operations in public forums.

3 Reputation and Consensus Mechanism

In the absence of moderation, permissionless peer-to-peer public forums are impractical. At the root of the problem lies Sybil attacks, which use large numbers of fake identities to abuse the system. For instance, it should take a few seconds to generate thousands of public/private key identities and SPAM million of messages into the system. For this reason, we propose a reputation system that works together with a consensus algorithm to mitigate Sybil attacks and make peer-to-peer public forums practical.

Section 3.1 describes the overall reputation and consensus mechanism, which can be applied to any public forum system that uses DAGs to structure its messages. Section 3.2 describes the concrete rules we implemented for public forums in Freechains.

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 distinguish good content amid excess; a *dislike* is a negative feedback that revokes content when crossing a threshold. Figure 3 summarizes the reputation operations and their goals. However, unlimited posts and likes are not satisfactory in the presence of Sybils. Therefore, *reps* must be subject to some sort of scarcity that demands non-trivial work immune to automation.

Bitcoin employs CPU proof-of-work to mitigate Sybil attacks. However, CPU or alternative extrinsic resources are not evenly distributed among humans, specially considering that most communications now use battery-powered devices. In the context of public forums, we understand that the human authoring ability is already an intrinsic resource that we can take advantage. 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

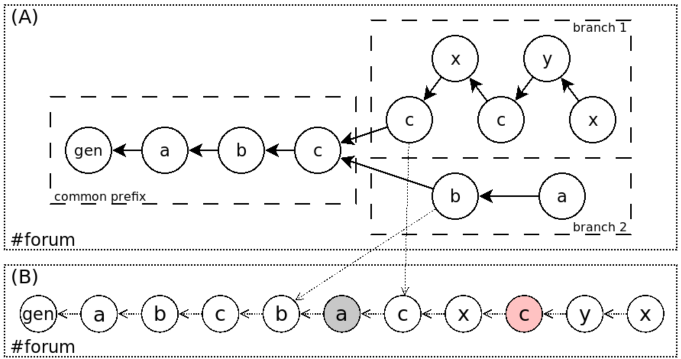

Fig. 4. (A) A public forum DAG with a common prefix and two branches. (B) Total order between blocks of the DAG after consensus.

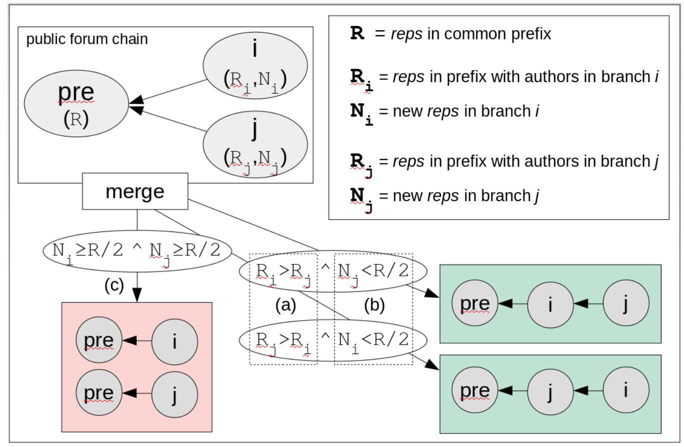

Fig. 5. Merging rules: (a) The branch with more reputation in the common prefix is ordered first. (b) A branch with 50% or more reputation then its prefix is ordered first. (c) The merge fails if rules (a) and (b) conflict.

only content authoring generates *reps*, while likes and dislikes just transfer *reps* between users. Still, scarce posts and likes are not yet sufficient because they demand consensus in the network. As an example, it is possible that an author with a single unit of *reps* receives a dislike at the same time she tries to post a new message in the network. If accounted before, the dislike invalidates the new post, otherwise the post is valid. Therefore, we need the same message ordering across peers to validate operations consistently.

Our solution is to order posts favoring forks with users that constitute the majority of reputation in the network. This way, in order to manipulate accountability, malicious users first need to cooperate, which is non-trivial and contradicts their intent.

Figure 4.A illustrates the reputation criterion. 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. 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, generate a lot of new content; in *branch 2*, only users a and b participate but with less activity. Nonetheless, *branch 2* would take priority because, before the forking point, a and b have more reputation than c, x, and y combined. User c represents here a malicious user trying to cultivate fake identities x and y in separate of the network to accumulate *reps*.

Figure 4.B indicates the consensus order between blocks in the forum. All operations in *branch 2* are considered before any operation in *branch 1*. The ordered list after consensus is only a view of the primary forum DAG structure, created for accountability purposes. 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 the last post by a (in gray) is a dislike to user c, which decreases its reputation. Then, it's possible that the last post by c (in red) is rejected together with all posts by y and x in sequence. Note that in 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.

Some other considerations about forks and merges: Peers that received branches with less reputation first will need to reorder all blocks starting at the forking point. This might even involve removing content in the end user software. This behavior is similar to blockchain reorganization in Bitcoin, when a peer detects a new longest chain and disconsiders old blocks. Likewise, peers that saw branches with more reputation first just need to put the other branch in sequence and do not need to recompute anything. The latter should be the normal behavior expected to happen in the majority of the network. Unlike Bitcoin, forks are not only allowed but encouraged due to the local-first software principle. However, the longer a peer remains disconnected, the more conflicting operations it may perform, and the higher are the chances of rejection when rejoining.

As a counterpoint, suppose users a and b in Figure 4.A have actually abandoned the chain for months and thus *branch-1* is legit. In this case, a and b might be the ones trying to take over the chain. Another possibility is that both branches are legit but became disconnected for a long period. It is simply impossible to judge with confidence. Nonetheless, it is unacceptable that a very old branch affects a long active chain. For this reason, the consensus algorithm includes an extra constraint when merging: If a branch creates enough *reps* to reach 50% of its prefix, then the algorithm favors this branch in future merges. In the example, suppose that the common prefix accumulates *50 reps* considering users a, b, and c. If *branch-1* creates at least 25 new *reps*, then the merge with *branch-2* will fail and the chains will never synchronize again. This situation is analogous to a hard fork in Bitcoin. Figure 5 summarizes the merging algorithm: rule (a) favors branches with more reputation; rule (b) favors branches with $50\%+$ *reps*; rule (c) enforces that rules (a) and (b) do not conflict.

A fundamental drawback of Merkle DAGs is that all replicas in the system need to store the complete graph in order to synchronize and verify new blocks. Tree pruning techniques allow to remove parts of the graph to save

Operation	Rule num	Rule name	Description	Observations
Emission	1.a	pioneer	Chain join counts *+30 reps* equally distributed to 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.
Tranfer	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 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.	

Fig. 6. Specific reputation rules for public forum chains in Freechains. The constants are arbitrary but could be chain parameters (e.g., *30 reps*).

space [9]. The rule (b) in the consensus algorithm allows to prune the chain DAG when crossing the $50\%+$ threshold, at least for lightweight clients in resource-constrained devices. However, these devices can no longer verify older forks and need to delegate trust to more powerful peers. A more pragmatic approach, but which requires cooperation among users, is to revoke past posts, which deletes associated payloads to save some space. This approach is more feasible in private groups and public identities chains, tough.

3.2 Public Forum Chains

We integrated the proposed reputation system in the public forums of Freechains to support content moderation and enforce consensus in the chains. Figure 6 details the concrete rules 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 $\#$):

```
> freechains key pubpvt 'pioneer-password'
4B56AD.. DA3B5F..
> freechains '#forum' join '4B56AD..'
10AE3E..
> freechains '#forum' post --sign='DA3B5F..' \
   'The purpose of this chain is...'
1_CC2184..
```

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

The most basic concern in public forums is to resist Sybils spamming the chains. Fully peer-to-peer systems cannot rely on identity logins or CAPTCHAs due to the lack of a central authority. Better alternatives include (i) building social trust graphs, in which users already in the community vouch for new users, or (ii) imposing economic 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 that an existing user likes 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 pioneer rule `1.a` solves the chicken-and-egg problem imposed by rule `4.a`.

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 key pubpvt 'new-author-password'
503AB5.. 41DDF1..
> freechains '#forum' join '4B56AD..'
10AE3E..     <-- same pioneer as before
> freechains '#forum' post 'Im a newbie...' \
   --sign='41DDF1..'
2_C3A40F..   <-- blocked post
> freechains '#forum' like '2_C3A40F..' \
   --sign='DA3B5F..'
3_59F3E1..
```

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 7 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 active users, the penalty can decrease to zero.

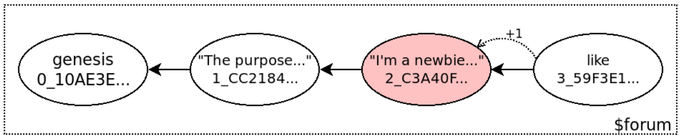

Fig. 7. The `like` approves the newbie message into the `#forum` DAG.

Back to Figure 7, a new user with *0 reps* tries to post a message (hash `C3A40F..`) and is blocked (rule `4.a`), as the red background highlights. Then, the pioneer likes 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 because it is followed by the pioneer, which still controls all *reps* in the chain (rule `2`).

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 7, after 1 day, the pioneer now accumulates *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 at a time. Hence, new posts during the 24-hour period will not award extra *reps* to the author. 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) censoring abuse (SPAM, fake news, illegal content, etc). Access to chains is permissionless in the sense that the actual identities behind posts are irrelevant for acceptance. Instead, it is the quality of content that is judged and accounted 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. The quality of posts is subjective and is up to users to judge then with likes, dislikes, or simply abstaining. 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 become hidden if it has at least 3 dislikes, and more dislikes than likes (rule `3`). However, considering that *reps* are scarce, dislikes should be more directed to combat abuse, but not much to eliminate divergences of opinion.

A post has three possible states: *BLOCKED*, *ACCEPTED*, or *HIDDEN*. Figure 8 specifies the transitions between states. If the author has reputation, a new post is immediately *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. Peers are not required to hold blocked posts and neither retransmit them to other peers.

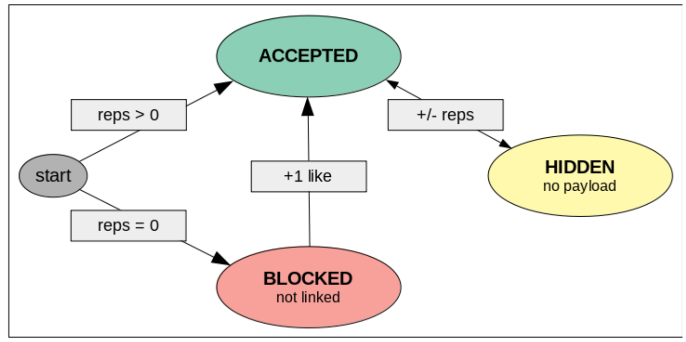

Fig. 8. State machine of posts: *BLOCKED* posts are not linked in the DAG. The payload of *HIDDEN* posts are not retransmitted. *ACCEPTED* posts are linked and retransmitted.

However, if blocked posts do not reach other users, they 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. If the number of dislikes exceeds the threshold (rule `3`), the block becomes *HIDDEN* and its payload is not retransmitted to other peers. The hash of the block depends only on the hash of the payload, so it is safe to remove the actual payload as long as you can prove its hidden state. Later, if the post receives new likes and changes its state again, it means that the payload is still known somewhere and peers can request it when synchronizing again.

4 Correspondence with CRDTs

Conflict-free replicated data types (CRDTs) [10] serve as a robust foundation to model shared data with concurrent updates from collaborative local-first applications [8]. We now introduce a three-layered CRDT scheme to build distributed applications: state-based CRDTs (CvRDT) at transport layer, operation-based CRDTs (CmRDT) at application layer, and CRDTs with arbitrary operations after consensus is applied.

At the transport layer, Merkle DAG chains are trivial CvRDTs because their missing parts are exchanged on synchronization to converge to the same state [7]. 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 [7]. CmRDTs have the advantage to store only small updates enough to reconstruct any version of the data. This contrasts with CvRDTs, which would require to store complete versions. Our proposed consensus algorithm goes one step further and transforms a chain DAG into a totally-ordered set, which leads to a CRDT that does not require commutative operations in the third layer.

Next, we illustrate this three-layered CRDT scheme through an example of a simple permissionless DVCS implemented on top of public chains.

4.1 A DVCS with Automatic Merge

We built a simple DVCS with the functionalities as follows:

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

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

As a concrete example, we model a Wiki article as a chain behaving as a VCS to hold its full edition history. An article can 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 multiple times:

```
> freechains #p2p.md join A2885F.. 2B9C32..
> echo "Peer-to-peer networking is..." > p2p.md
> freechains-vcs #p2p.md commit --sign=699299..
1_4F3EE1..
> echo "The [USENET](#usenet.md), ..." >> p2p.md
> freechains-vcs #p2p.md commit --sign=699299..
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=699299..
```

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`. Finally, the commit posts the patch file back into the chain. Of course, the chain name and signature are parameters in the actual implementation. 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 "Peer-to-Peer does not scale!" >> p2p.md
> freechains-vcs #p2p.md commit --sign=320B59..
23_AE3A1B..
```

A *checkout* needs to recreate the latest version of the file in the repository by applying all patches since the genesis block. The operation expands as follows:

```
> rm p2p.md && touch p2p.md
> for blk in `freechains #p2p.md traverse` do
    freechains #p2p.md 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 `traverse` 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 (peer-to-peer networks do scale), other users in the network will dislike it until the block becomes *HIDDEN* in the chain (as described in Figure 8):

```
> freechains #p2p.md dislike 23_AE3A1B.. --sign=USR1
> freechains #p2p.md dislike 23_AE3A1B.. --sign=USR2
> freechains #p2p.md dislike 23_AE3A1B.. --sign=USR3
> freechains-vcs #p2p.md checkout p2p.md
> cat p2p.md
Peer-to-peer networking is...
The [USENET](#usenet.md), ...
...
```

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

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/peer/Peer/g' p2p.md    <-- fix typo
> freechains-vcs #p2p.md commit --sign=699299..
27_A..

# PEER B (less reputation):
> sed -i 's/networking/computing/g' p2p.md
> freechains-vcs #p2p.md commit --sign=320B59..
27_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
27_B..
> cat p2p.md
Peer-to-Peer networking is... <-- typo fixed
The [USENET](#usenet.md), ...  \- not "computing"
...
```

After they commit the conflicting changes, the peers synchronize in both directions and reach the state of Figure 9. 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 above: once a conflict is 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. Nonetheless, 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, because it keeps the file in an usable state while still showing that there is a conflict to resolve. For instance, the authors can later decide to dislike one of the two commits to revoke it

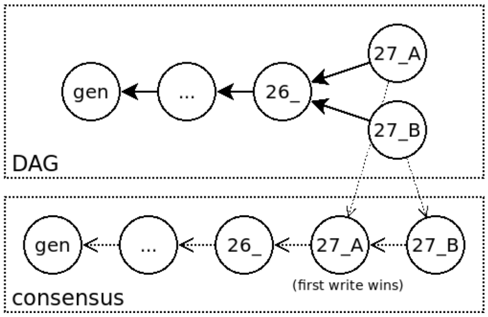

Fig. 9. The branches in the DAG are ordered by reputation. Only the first patch is applied successfully (first write wins).

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* tool 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 DAG between peers, which eventually reach the same 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, which we apply sequentially until a patch conflict occurs. The final state of the file is guaranteed to be consistent, i.e., the result of a sequence of correct patch applications.

Applications that rely only on commutative operations can traverse the DAG in any causal order, possibly in parallel. This is the case of most social apps with threaded conversations, in which branches typically do not interfere with each other. For instance, it is not problematic to rely on block timestamps to display messages in chats, forums, and social media posts. This is also the general case for notifications in these applications, such as status updates and social engagements.

Towards richer distributed collaborative applications, we can employ CRDTs to model data other than raw text. As an example, *Automerge* [11] manipulates JSON objects, which provides non-trivial datasets with robust merging policies.

5 Related Work

Many other systems have been proposed for distributed content dissemination [3], [12]. Here we consider publish-subscribe protocols, federated applications, and fully peer-to-peer systems.

5.1 Publish-Subscribe Protocols

Decentralized topic-based publish-subscribe protocols, such as *XMPP* [13], *ActivityPub* [14], and *gossipsub* [15], 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. Nonetheless, some pubsubs do not rely on server roles and, instead, use peer-to-peer gossip dissemination [16], [17], [18], [19], [15], [17]. 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 [20].

Being generic protocols, pubsubs are typically unaware of the applications built on top of them. In contrast, the pubsub of Freechains is integrated with the semantics of chains and already verifies blocks during connections. For instance, to flood the network with posts, malicious nodes need to spend reputation, which takes hours to recharge (rule 2 in Figure 6). Even blocked posts (Figure 8), which have limited reachability, are not a concern either. Another advantage of a tighter integration 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.

5.2 Federated Applications

Federated applications, 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 applications for social media, chat, and microblogging [12].

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. 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 [12]. 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[2], but which applies only in clients, after the messages have been delivered.

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.

5.3 Peer-to-Peer Systems

Bitcoin [5] is probably the most successful permissionless network but serves specifically for electronic cash. IPFS [21] and Dat [22] are data-centric systems for hosting large files and applications, respectively. Scuttlebutt [23] and Aether [12] are closer to Freechains goals and cover human-centric $1 \rightarrow N$ and $N \leftrightarrow N$ public communication, respectively.

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

Bitcoin adopts CPU *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 [24] that acknowledges that centralization is inevitable (i.e., the richer gets richer), and thus uses a function of time and wealth to elect nodes 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 authoring content for the chains themselves. We believe that human work grows 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 typically 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 communication, which has value in itself.

IPFS [21] is centered around immutable content-addressed data, while Dat [22] 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 [23] 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. This architecture is very similar to $1 \rightarrow N$ public identity chains of Freechains in Figure 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*). Users 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 users 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. In Freechains, new users require a single like for visibility in the community, which relies on the reputation system to prevent abuse. Also, a chain stores its own posts only, instead of off-topic posts from its subscribers.

Aether [12] 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, using 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.

6 Conclusion

We propose a new permissionless consensus algorithm for content dissemination in peer-to-peer networks. The main insight of the algorithm 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 distributed.

The reputation and consensus mechanism is integrated into Freechains, a peer-to-peer protocol that offers permissionless public forums immune to Sybil attacks. The protocol replicates Merkle DAGs that represent causal relationships between messages in the network. The consensus algorithm transforms DAGs into totally-ordered sets, which eliminates conflicts analogous to double spending in Bitcoin. This mechanism can be applied to any system that uses DAGs to structure its messages.

We also conceptualize a three-layered CRDT scheme to build collaborative applications: (i) the Merkle DAG as a CvRDT at the transport layer, (ii) commutative updates as a CmRDT at the application layer, (iii) arbitrary operations as a CRDT after consensus is applied. On top of this scheme, we built a simple distributed version control system that resolves commit conflicts automatically.

References

[1] J. Zittrain, "Fixing the internet," 2018.

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

[4] 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.

[5] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," Tech. Rep., 2019.

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

[7] H. Sanjuan, S. Poyhtari, P. Teixeira, and I. Psaras, "Merkle-crdts: Merkle-dags meet crdts," 2020.

[8] 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.

[9] R. Matzutt, B. Kalde, J. Pennekamp, A. Drichel, M. Henze, and K. Wehrle, "How to securely prune bitcoins blockchain," in *2020 IFIP Networking Conference (Networking)*. IEEE, 2020, pp. 298–306.

[10] 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.

[11] 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.

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

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

[14] C. Webber, J. Tallon, and O. Shepherd, "Activitypub," 2018.

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

[16] 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 2007 inaugural international conference on Distributed event-based systems*, 2007, pp. 2–13.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

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

[22] 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.

[23] 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.

[24] 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.