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 version control system that relies on reputation consensus to resolve conflicts automatically.

Index Terms—bitcoin, crdts, 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 the decentralization of authority and network infrastructure, some new challenges arise to deal with malicious users and to enforce overall state consistency.

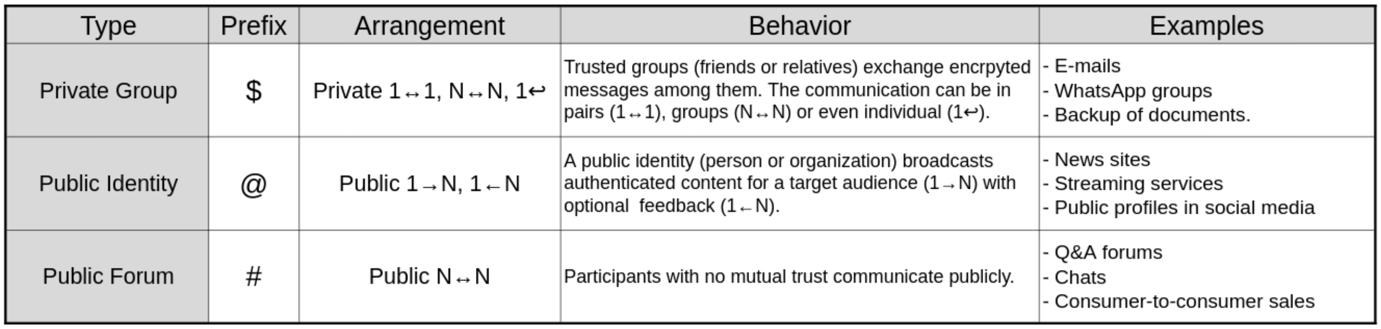
In an ideal Internet forum, all messages or posts (i) reach even temporarily disconnected users; (ii) are delivered in a consistent order; (iii) are respectful and on topic. In a centralized system, items (i) and (ii) are trivially achieved assuming availability and delivery order in the service, while 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 in different peers [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 the problem of conflicting messages in public discussions. However, Bitcoin just supports transfers between users, with no subjective judgment that could affect the actual transactions. In contrast, our challenge is to use social interactions between humans to evaluate content and mitigate abuse.

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 worked more in 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 of a permissionless version control system that relies on consensus to apply automatic merges.

Our main contribution is to make public forums practical with complete decentralization. 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 public forum system that uses DAGs to structure its messages. As a derived contribution, the consensus implies a total order among messages, which backs the design of CRDTs [7] for collaborative authoring platforms, such as distributed version control systems.

As a main limitation, Merkle DAGs are ever growing data structures that also carry considerable metadata overhead. In addition, the required per-block validation is a bottleneck for real-time applications, such as chats and video calls. 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 that helps users understand the evolution of



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.

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 correspondences with CRDTs and prototype a simple version control system. In Section 5, we compare our protocol and consensus mechanism with other publish-subscribe middlewares, 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 crypto 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 crypto 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:

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

However, since the network is inherently concurrent and

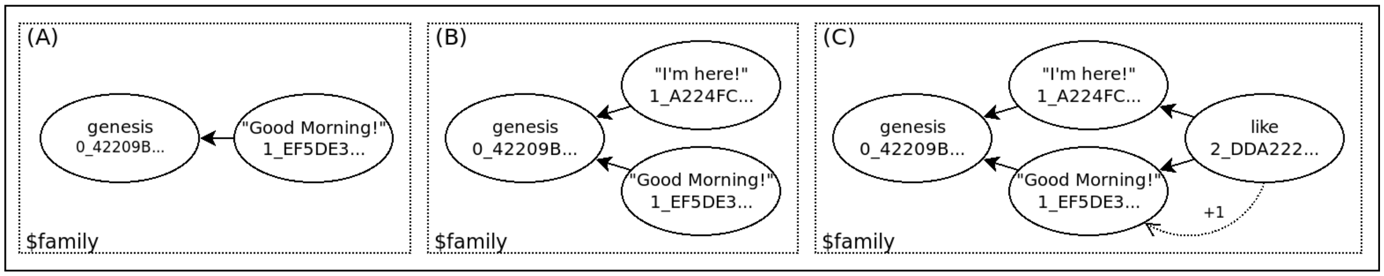


Fig. 2. Three DAG configurations. (A) Single head pointing to genesis block. (B) Fork with heads pointing to genesis block. (C) Like pointing to past block, which also merges previous heads.

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 the fundamental obstacle to reach consensus. In private chains, we can apply simple methods to reach consensus, 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 the consensus algorithm.

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

```
> freechains crypto 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 this 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.

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

1. http://www.freechains.org

Operation	Effect	Goal
Emission	Consolidated posts award *reps* to author.	Encourage content authoring.
Expense	New posts deduct *reps* from author.	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.

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 [?], [?], [7]. Section 3.2 describes the concrete rules we implement 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, without restrictions, posts and likes alone 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, in the context of content publishing, we understand that authoring is already an intrinsic human work 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 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 blocks the new post, otherwise

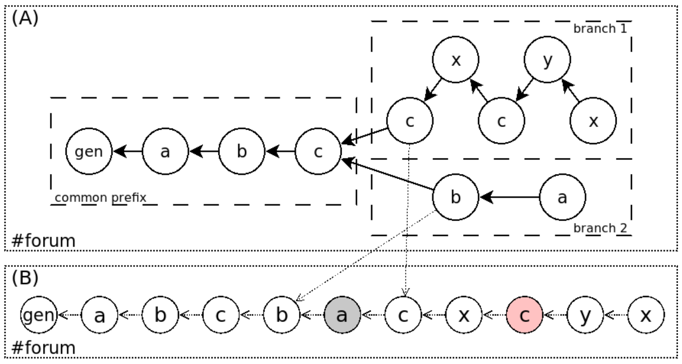


Fig. 4. (A) A public forum DAG with a common prefix and two branches. (B) Total order between blocks of the DAG.

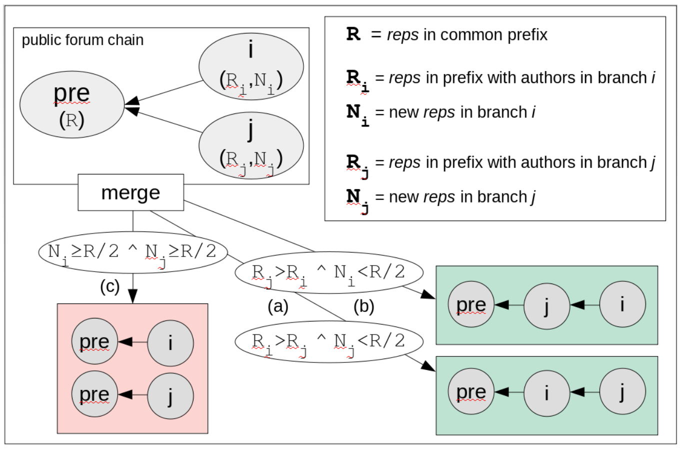

public forum chain
pre
(R)
i
(R_i,N_i)
j
(R_j,N_j)
merge
R = reps in common prefix
R_i = reps in prefix with authors in branch i
N_i = new reps in branch i
R_j = reps in prefix with authors in branch j
N_j = new reps in branch j
N_i ≥R/2 ^ N_j ≥R/2
(c)
R_j>R_i ^ N_i<R/2
(a)
(b)
R_i>R_j ^ N_j<R/2
pre
i
pre
j
pre
j
i
pre
i
j

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.

the post is valid. Therefore, we need the same message ordering across all peers to account the reputation of users and validate the operations consistently.

Our solution is to order posts favoring forks with participants that constitute the majority of reputation in the network. This way, in order to manipulate accountability, malicious users first need to cooperate to gain reputation, 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* takes 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*. However, the whole malicious *branch 1* is vulnerable because users in *branch 2* with more previous reputation take the priority and can overthrow user c.

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 consensus ordered list is only a view of the primary forum DAG structure 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 whole remaining branch as if it never existed. Note also that users in the branch with more reputation may 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 with *branch-1* removes all of its blocks.

Some other considerations about forks and merges: Peers that received branches with less reputation first (*branch 1*) 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 (*branch 2*) just need to put the other branch in sequence and do not need to recompute anything. This should be the normal behavior and is 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. A third possibility is that both branches are legit but became disconnected for a long period. It is simply impossible to determine. Nonetheless, it is unacceptable to permit 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 preserves this branch as first 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) preserves branches with $50\%+$ *reps* as first; 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 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

Operation	Rule		Description	Observations
	num	name		
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.	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 for free.
	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.

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 (rule `3.b`) which will delete payloads and save some space. This approach is more feasible in private groups and public identitis 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 crypto 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 its first posts and likes, while they gradually transfers *reps* to other authors, which may also transfer to other authors, expanding the community. In this regard, the *post* command in sequence above, which is signed by the single pioneer and indicates the purpose of the chain to future users.

Our 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. Other 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.

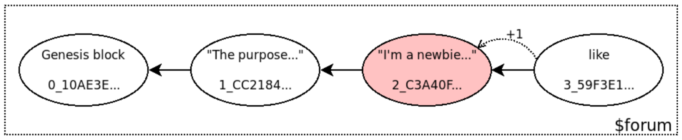


Fig. 7. The `like` approves the newbie message into the `#forum` DAG.

We propose a mix between trust graphs and economic costs. Rule `4.a` imposes that authors require at least *1 rep* to post, while rule `2` imposes a cost of *1 rep* for each new post. To vouch for new users, rule `3.a` allows that existing users like and unblock newbies' posts, but also at the cost of *1 rep*. These rules impose costs not only for welcoming new users, but also for posting new messages, which prevents abuse from malicious users already in the chain. Note that the pioneer rule `1.a` solves the chicken-and-egg problem imposed by rule `4.a`. Access is permissionless in the sense that in honest chains the actual identities behind posts are irrelevant for acceptance, only the quality of content.

The next commands illustrate the reputation rules with numbers. A new user joins the same public chain as before and posts a message, which is welcomed with a like signed by the pioneer:

```
> freechains crypto 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..
> 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 an initial message. A new post penalizes its author with -*1 reps* (rule `2`) which is restored after at most 12 hours. This period depends on the activity that comes after the new post (including it). The more activity from reputed authors, the less time the discount persists. In the example, since the post is from the pioneer, which currently controls all *reps* in the chain, the penalty falls immediately and the pioneer remains with *30 reps*. This mechanism limits the excess of posts in a chain dynamically. For instance, in a slow technical mailing list, it is more expensive to post messages in sequence. However, in a chat with the majority of users participating, the penalty for new posts can be reduced to zero. Back to the figure, a new user with *0 reps* tries to post a message (hash `C3A40F..`), which is initially blocked (rule `4.a`), as the red background highlights. Then, the pioneer likes the blocked message, which reduces himself to *29 reps* and increases new user to *1 rep* (rule `3.a`). Once again, the penalty expires immediately since the like that follows the new post is from the pioneer still with all *reps* in the chain. 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 24 hours after each new post with *1 rep*. This rule stimulates content creation and grows the economy of chains. The 24-hour period allows other users to judge the post before awarding its author, and also regulates the growth speed of the chain. Therefore, after the commands above complete 1 day, the pioneer 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. New posts during the 24-hour period will not awards 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). 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, the scarcity and 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 the number of dislikes is higher than the number of likes (rule `3`). However, considering that *reps* are scarce, dislikes should be more directed to combat abuse, but not much to eliminate divergence of opinion.

A post has three possible states: *BLOCKED*, *ACCEPTED*, or *HIDDEN*. Figure 8 specifies the transitions between the states. If the author has reputation, its 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. However, if blocked posts do not reach other users, they will never have the chance to be welcomed with a like. A reasonable policy for blocked posts is to hold them 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 of a post exceeds the threshold (rule `3`), its payload becomes *HIDDEN* and is not retransmitted to other peers. Since the Merkle DAG depends only on its hash, removing the actual payload is harmless. Also, the DAG itself contains the dislikes that prove the hidden state to other peers. Later, if the post receives new likes and changes its state, it means that the payload is still known somewhere and peers can request it when synchronizing again.

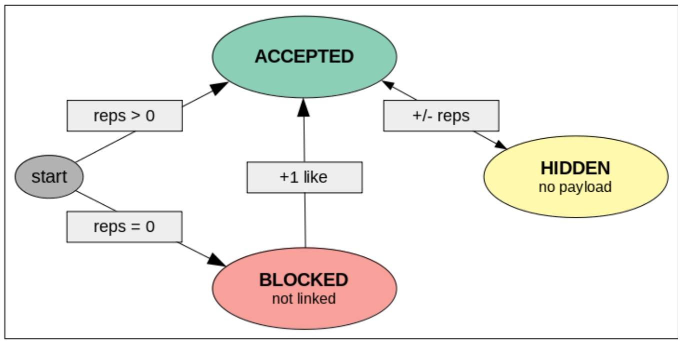


Fig. 8. State machine of posts: *BLOCKED* posts are not linked in the DAG. The payload of *HIDDEN* posts are not retransmitted. *ACCEPTED* posts are linked and retransmitted.

4 Correspondence with CRDTs

Conflict-free replicated data types (CRDTs) are data structures that can be updated concurrently with the guarantee that they converge to the same state on synchronization, despite arbitrary failures [10]. CRDTs serve as a robust foundation to model data in networked local-first applications [8].

Considering Freechains operating as a transport layer for networked applications, the Merkle DAG chains are trivially CRDTs because the DAG itself is the CRDT [7]. More specifically, they are state-based CRDTs (CvRDTs) because, on synchronization, the missing parts of the self-verifiable DAGs are exchanged to converge to the same state. At the application layer, however, DAGs are not CRDTs because branches are delivered to peers in different orders, and thus can lead to different states when processed. An interesting approach [7] is to require blocks in the DAGs to represent commutative operations. Commutative operations, combined with Merkle's exactly-once delivery guarantee, lead to operation-based CRDTs (CmRDTs) at the application layer. CmRDTs have the advantage that it only needs to store the operations to reconstruct any version of the data, instead of storing each complete version of the data as required in CvRDTs. Our proposed consensus algorithm goes one step further and transforms a DAG into a totally-ordered set, which leads to a CRDT that does not require commutative operations.

Next, we illustrate the three-layered CRDT framework through an example of a simple permissionless peer-to-peer Version Control System implemented on top of public chains.

4.1 A P2P VCS with Automatic Merge

We built a simple peer-to-peer Version Control System with the following operations and respective Freechains operations in parenthesis:

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

The system executes under public chains, which means that repositories are permissionless and adhere to the reputation and consensus mechanism of Freechains. The main innovations in this system are that (i) users can rate commits which can thus be rejected, and (ii) checkouts merge commits automatically based on consensus.

Suppose we want to model a Wiki with public forums in Freechains. Each article is a chain behaving as its own VCS, holding the full edition history. Articles can link to other articles referring to other chains.

We use the Freechains operations directly, except for *commit* and *checkout*, which require multiple operations. As an example, we join a public chain with the name of the file we want to track and multiple pioneers. We then edit the file and commit it:

```
> 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 checks out the remote version held in the chain to an output file `p2p.remote`. Then, it diffs the remote against the local version to generate the file `p2p.patch` with our changes. Finally, the commit posts the signed patch file into the chain.

Much time later, another reputed author synchronizes with us, checks out the file, then edits and commits it back:

```
> freechains #p2p.md recv '<our-ip>'
> 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..
```

The checkout operation needs to recreate the file by applying all patches since the genesis block:

```
> 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
    fi
  done
```

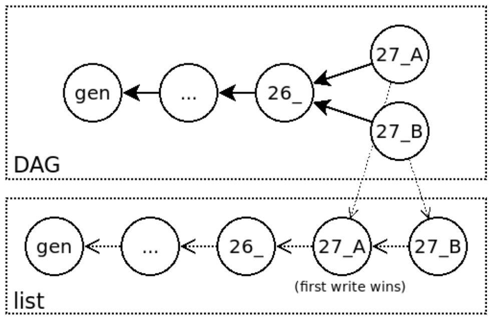


Fig. 9. The branches in the DAG are ordered by reputation. The application ...

The `traverse` operation 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 current version of the file in the chain. If the patch application fails, the command exhibits the hash of the block in the screen.

Since the last edition is clearly wrong, other users in the network will dislike its commit until it 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 applies an empty patch and the offending line disappears from the file. This illustrates how the reputation system enables collaborative permissionless editing.

Next, we create a conflict situation in which two authors in different peers edit the same line 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 out of 1 hunk FAILED -- saving rejects to file p2p.md.rej
27_B..
> cat p2p.md
Peer-to-Peer networking is...
The [USENET](#usenet.md), ...
...
```

After they commit the conflicting changes, the peers synchronize in both directions and reach the state of Figure 9. Then, we checkout the file which applies the patches respecting the consensus order. As a result, although the conflict exists, we see that the first branch is still applied, leaving the file in the longest possible consistent state. Because of the `break` in the checkout operation above, note that 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 preserve further work in the branches with more reputation. However, 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 is the most advantageous, because it keeps the file in an usable state while showing that there is a conflict to solve. For instance, the authors can later decide to dislike one of the two commits to settle the file (possibly inverting their reputation) and remove the warning.

In summary, the proposed reputation and consensus mechanism empowers a simple P2P VCS with cooperative authoring and automatic conflict resolution. It only requires *diff & patch* and the basic API of Freechains with no extra manipulation of the internal structure of the DAG.

We apply the three-layered CRDT framework discussed in the beginning of this section as follows: In the first layer, the whole file commit history of small patches is transported as a DAG between the peers with the basic commands of Freechains. Eventually, all peers reach to same state with the full commit history. However, in order to recreate the file, the peers need to interpret the DAG, which requires a commutative operation in the second layer because of branches. The *patch* tool is mostly commutative, except when branches edit the same lines of a file. Hence, in these situations we resort to the third layer which determines a total order between the patches, which we apply 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.

5 Related Work

Many other systems have been proposed for distributed content dissemination [11]. Here we consider publish-subscribe middlewares, federated applications, and fully peer-to-peer systems.

5.1 Publish-Subscribe Middlewares

Decentralized topic-based publish-subscribe middlewares, such as *XMPP* [12], *ActivityPub* [13], and *gossipsub* [14], assist the development of networked applications by decoupling publishers from subscribers. A key aspect of *pubsubs* is that, although the end points communicate without knowledge of each other, the brokers or servers in between them still have a special role in the network. For instance, they might do authentication and validation of posts, and ultimately serve the queues that connect producers to consumers. Some pubsubs use gossip dissemination in unstructured peer-to-peer networks and do not rely on server roles [15], [16], [17], [18], [14], [16]. They focus on techniques for scalability and performance, such as throughput, simultaneous connections, load balancing, and real-time relaying. These techniques are important but not sufficient in permissionless networks that have to deal with malicious nodes, particularly Sybils [19].

Being middlewares, pubsubs typically deal with low-level communication aspects between nodes but are agnostic about the applications built on top of it. In contrast, the pubsub of Freechains integrates with the semantics of chains and verifies blocks during connections. For instance, the only way to flood the network is spending reputation, which takes hours to recharge (rule `2` in Figure 6). Blocked posts (Figure 8) have limited reachability and are not a concern either. Another advantage of this integration is that the Merkle DAG structure simplifies the synchronization between nodes, provides persistence and prevents duplication of messages. Full persistence facilitates dealing with long churn periods, and de-duplication preserves CmRDTs even if operations are not idempotent.

5.2 Federated Applications

In federated applications, client identities are attached to a single server just like in centralized systems, but servers can communicate in a standardized way to allow communication across borders. SMTP is probably the most popular federated protocol, allowing users from one domain to exchange messages with users of other domains seamlessly. More recently, *Diaspora*, *Matrix*, and *Mastodon* (which is based on ActivityPub) filled the domains of social media, chat, and microblogging under federations [11]. As a drawback, identities in federations are not portable and servers may be banned or shut down for different reason. Also, users may becomes unsatisfied with their local server policies. In any of these cases, the user will have to grab her content, move to another server, and announce her new identity to followers.

Moderation is also a major concern in federated applications [11]. As examples, messages crossing server boundaries may diverge from local policies, and identities may be target of SPAM. In this context, moderation can be applied locally users or in servers as one-size-fits-all solutions. In either case, Sybils can still propagate messages in the network. With no granularity for moderating actions and no form of consensus, it is difficult to make pervasive public forums practical. Matrix supports a moderation system[2], but which is permissioned and applied only after messages disseminate.

As a counterpoint, federated protocols seem to be more appropriate for stream-based real-time applications such as chats and streaming with a large number of small messages. The number of hops and header overhead can be much smaller in client-server architectures in comparison 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 peer-to-peer network but serves the very specific functionality of electronic cash. IPFS [20] and Dat [21] are data-centric systems for hosting large files and applications, respectively.

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

Scuttlebutt [22] and Aether [11] cover human-centric communication between friends and groups, respectively. Scuttlebutt and Aether have similar goals as Freechains, focusing on $1{\rightarrow}N$ and $N{\leftrightarrow}N$ public communication, respectively.

Bitcoin [5] adopts CPU *proof-of-work* to achieve consensus, which does not solve the centralization issue completely due to the high costs of equipment and energy. *Proof-of-stake* is a prominent alternative [23] that acknowledges that centralization is inevitable (i.e., the richer gets richer) and uses a function of time and wealth to elect nodes to mint new blocks. An advantage is that these proof mechanisms are generic and apply to multiple domains. We chose an intrinsic scarce resource to reach consensus, which is human work to create content of quality for the chains themselves. We believe that authoring work grows linearly with effort and is not portable across chains with different topics. This hypothesis supports the intended decentralization. Another distinction to generic public ledgers is that they typically require permanent connectivity to avoid forks, which opposes our local-first principle. This is because a token transaction only makes sense when it is part of the longest chain, which is not the case for messages exchanged locally.

IPFS [20] is centered around immutable content-addressed data, while Dat [21] 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 architecture, 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 are looking for, 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, holding large files in Merkle DAGs replicated across the network is also impractical. An alternative is to use DHT links in Merkle payloads to benefit from both architectures.

Scuttlebutt [22] is designed around public identities that follow each other to form a graph of connections which is also 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 in Figure 1. For group $N{\leftrightarrow}N$ communication, Scuttlebutt uses the concept of channels, which are in fact only hash tags (e.g. #*sports*). Users can tag posts, which appear not only in their feeds but also in virtual feeds representing these channels. However, users only see channel posts from users they already follow. In practice, channels just 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). Conversely, new users also struggle to integrate in channel communities since 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 unrelated posts from its subscribers.

Aether [11] provides peer-to-peer public communities aligned with $N{\leftrightarrow}N$ public forums of Freechains. A fundamental difference is that 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 reputation and consensus mechanisms for moderation.

6 Conclusion

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

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

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

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

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

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

[21] 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.

[22] 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.

[23] 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.