Peer-to-Peer Consensus via Authoring Reputation

Abstract—Content publishing in public Internet forums and social media suffers from excess and abuse, such as low quality posts 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 public content dissemination without centralized control. Abuse is mitigated 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 depends solely on human work to create and rate content, preventing abuse while imposing consensus on a peer-to-peer setting.

Index Terms—distributed consensus, peer-to-peer, publish-subscribe, reputation system

1 Introduction

Content publishing in public Internet forums and social media platforms is increasingly more centralized in the hands of 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 network infrastructure, some new challenges arise to deal with malicious users and 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 replicate the whole conversation in all peers and disseminate it proactively until all users receive it [3]. However, this approach does not guarantee consensus since posts can be received in conflicting orders in different peers. As an example, antagonistic messages such as *"X is final"* vs *"Y is final"* might be sent concurrently, preventing the network to determine as a group its intention as X or Y [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 all transactions in the system. This way, Bitcoin prevents double spending [5], which is analogous to conflicting messages in public discussions: deciding between X and Y as a group is the same as transferring bitcoins to X and Y with insufficient funds for both. However, Bitcoin just blindly transfers bitcoins 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 consensus algorithm based on authoring reputation. Inspired by Bitcoin, authors accumulate tokens named *reps*, which serve as currency to rate posts in the network. 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 sequenced as a linked list that orders branches with more reputed authors first (i.e., branches with more work). The 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, which is always the one with less work, is removed from the DAG and the linked list is recalculated. We integrated the proposed consensus algorithm into Freechains, a peer-to-peer publish-subscribe content dissemination protocol [6].

Our main contribution is to make public forums practical with complete decentralization. The proposed reputation and consensus mechanism depends solely on human work to create and moderate content, which distinguishes itself from most systems which rely on external resources to reach consensus. The general idea of the algorithm can be applied to any public forum system that uses DAGs to structure its messages.

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 how it integrates with public forums in Freechains. In Section **??**, we discuss the correspondences

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.

with CRDTs and present a peer-to-peer Git application with automatic merge. In Section **??**, xxx. In Section **??**, yyy.

2 FREECHAINS

Freechains is an unstructured peer-to-peer topic-based publish-subscribe system [6]. Each topic, or *chain*, is organized as a *Merkle DAG*, i.e., a directed acyclic graph immune to modifications. The chain DAG is disseminated peer by peer in the network with gossiping. This way, as an author posts to a chain, other users subscribed to the same chain eventually receive the message. Freechains supports multiple types of chains with different 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 in the network 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

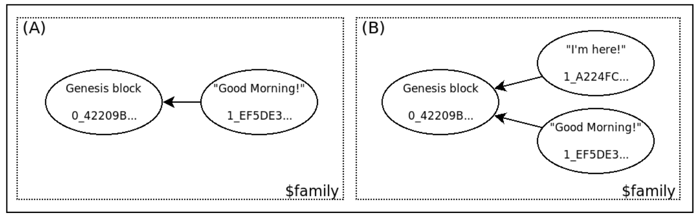

Fig. 2. Two DAG configurations. (A) has a single head pointing to the genesis block. (B) has a fork with two heads pointing to the genesis block.

hash of a block depends on its payload and hash of previous block.

Freechains adheres to the *local-first* software principle [7], allowing networked applications to host their own data and work locally while offline. Except for synchronization, all other operations in the system only affect the local replica. In particular, joining a chain with the same arguments in another peer results in the same genesis state, even if the peers have never met before. Hence, before synchronizing, 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 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>
```

The complementary command *send* synchronizes the DAGs in the other direction. Note that Freechains does not construct a network topology or synchronize peers automatically. There are no preconfigured peers, no root servers, no peer discovery. All connections happen through the *send* and *recv* commands which have to specify the peers explicitly. In this sense, the protocol only gives basic support for communication in pairs of peers and any further automation requires external tools.

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!
```

Now, the new peer is in the same state as the original peer in Figure 2.A. 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 posts a message "I'm here!" **before** the *recv* above, when the local DAG is still in its genesis state. In this case, as illustrated in Figure 2.B, the resulting graph after the synchronization now contains 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, which we use in this example, we can apply simple methods to reach consensus, such as relying on the timestamps of blocks. However, in public forums, a malicious user can modify his local time to affect new block timestamps and 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
```

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 @), which relies on public-key cryptography to attach an identity to a chain and verify the authenticity of posts:

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

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

Freechains is less than 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, 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.

1. http://www.freechains.org

Operation	Description	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.

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 [**?**], [**?**], [8]. 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* operation initially penalizes authors until it consolidates and counts positively; a *like* operation is a positive feedback that helps subscribers distinguish good content amid excess; a *dislike* operation 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, only content authoring generates *reps*, while likes and dislikes only transfer *reps* between users. Still, scarce posts and likes are not yet enough because they demand consensus in the network. Now, we need to track the reputation of users to check if they are allowed to post new messages or spend likes. Since posts and likes are concurrent in the network, we need to figure out how to order them consistently across all peers so that they converge to the same state. As an example, it is possible that an author with a single unit of *reps* receives a dislike at the same time he posts a new message in the network. If accounted before, the dislike blocks the new post, otherwise the post is valid. Even though it is impossible to determine which action happened first, the network as a whole needs to agree on a single order because this decision affects all future operations.

Our solution is to order posts in the network favoring forks with participants that constitute the majority of reputation. This way, in order to manipulate consensus, malicious users first need to cooperate to gain reputation, which is a non-trivial work 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

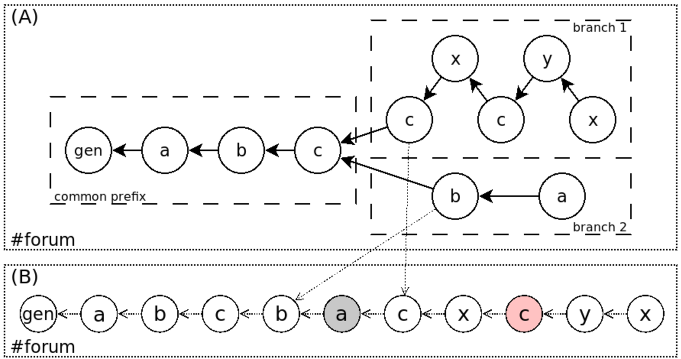

Fig. 4. (A) A public forum DAG with a common prefix and two branches. (B) Total order between blocks of the DAG.

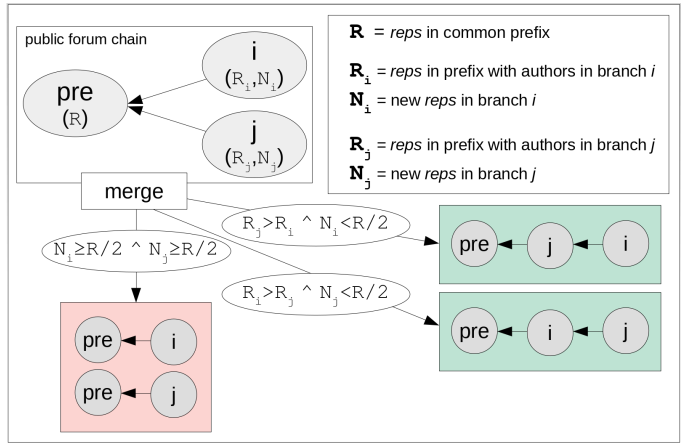

Fig. 5. Merging rules: (a) The branch with more reputation in the common prefix is ordered first. (b) A branch with 50% or more reputation then its prefix is ordered first. (c) The merge fails if rules (a) and (b) conflict.

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 here represents 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 accounted before any operation in *branch 1*. At any point in the consensus timeline, if an operation fails, all remaining blocks in the offending branch are removed from the forum 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. We need to remove the whole remaining branch as if it never existed. Unlike Bitcoin, forks are not only allowed but encouraged due to the local-first principle. The resulting ordered list is only a view of the primary DAG structure. However, the longer a peer remains disconnected, the more conflicting operations it may perform, and the higher are the chances of rejection when rejoining. Note that users in *branch 2* with more reputation may judge *branch 1* as malicious and can react even after the fact. For instance, users a and b can post extra dislikes in *branch 2* so that merging with *branch 1* removes all of its blocks.

Some other considerations about merging forks: 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.

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 are the ones trying to take over the chain by forging an early *branch-2*, even with knowledge about *branch-1*. There is even a third option in which both branches are legit but became disconnected for a long period. It is impossible to determine. Nonetheless, the breach that permits an forged branch to take over a long active chain is compromising. For this reason, the consensus algorithm includes an extra constraint when merging: If a branch creates new *reps* that reach 50% of its prefix, then the algorithm guarantees that this branch takes priority in 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 the branch with more reputation; rule (b) forces the branch with $50\%+$ *reps* of its prefix to go first; rule (c) enforces that previous rules 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, at least for lightweight clients for resource-constrained devices. A branch that reaches the threshold of $50\%+$ *reps* is freezed in the ordered list and becomes safe to remove along with all of its past back to the genesis block. However, this host can no longer verify forks starting past the removed blocks and will fail to synchronize. In this case, the host needs to delegate trust to other nodes for the first verification of these forks.

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

Operation	Rule		Description	Observations
	num	name		
Emission	1.a	pioneer	Chain join counts *+30 reps* to pioneer.	`[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	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.

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* to the pioneer referred in the public key. The pioneer shapes the initial culture of the chain with its first posts and likes, while he gradually transfers *reps* to other authors, which may also transfer to other authors, expanding the community. In this regard, the *post* command in sequence, which is signed by the 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.

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`.

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..’
```

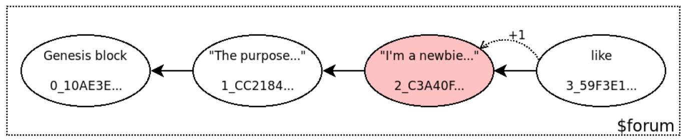

Fig. 7. The `like` approves the newbie message into the `#forum` DAG.

```
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

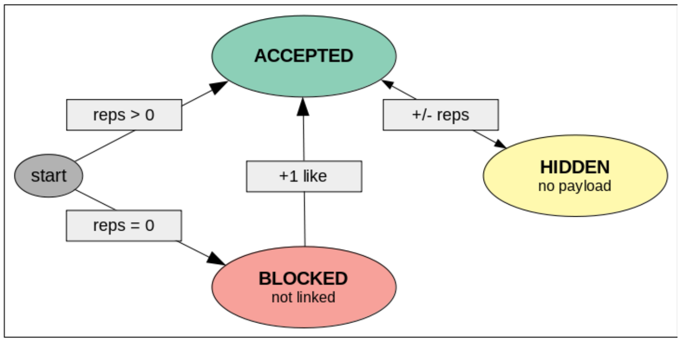

Fig. 8. State machine of posts: *BLOCKED* posts are not linked in the DAG. The payload of *HIDDEN* posts are not retransmitted. *ACCEPTED* posts are linked and retransmitted.

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.

4 Correspondence with CRDTs

Conflict-free replicated conflict-free replicated data type (CRDT) is a data structure which can be replicated across multiple computers in a network, where the replicas can be updated independently and concurrently without coordination between the replicas, and where it is always mathematically possible to resolve inconsistencies that might come up.

2020 Merkle-CRDTs Merkle-DAGs meet CRDTs Hector Sanjuan, Samuli P oyhtari, Pedro Teixeira, and Ioannis Psaras 0 cites We study Merkle-DAGs as a transport and persis- tence layer for Conflict-Free Replicated Data Types (CRDTs), Strong eventual consistency (SEC) addresses these issues by establishing an additional safety guarantee: if two replicas have received the same updates, their state will be the same. Additionally, CRDTs also feature monotonicity. The concept of monotonicity applied to data types is the notion that every update is an inflation, making the state grow, not in size, but in respect to a previous state. This implies that there will always be an order between states. Monotonicity implies that rollbacks on the state are not necessary regardless of the order in which updates happen. This implies that the Ł operation must be idempotent (X Ł X = X), commutative (X ŁY = Y ŁX) and associative ((X ŁY)ŁZ = X Ł(Y ŁZ)). There are two prominent types of CRDTs: state-based and operation-based CRDTs - FC is state-based and nodes are operations, better of both worlds - exactly once op (not 0 not 2) - only holds ops (not full states) The integration of distributed dataset synchronisation fea- tures natively at the network layer of the network is clearly an advanced endeavour, which comes with its own challenges.

sync algorithm??? - two dags if one contains the other, ok - otherwise sync diffs - converge, monotonic - Merkle-Clock -¿ Growing Set Unsurprisingly, the Merkle-Clock representation corre- sponds in fact to a Grow-Only-Set (G-Set) in the state-based CRDT form [33]

Conflict-free Replicated Data Types Marc Shapiro, Nuno Preguia, Carlos Baquero, Marek Zawirski

- what is? - importance for LFS - converge to same data - conflicts are solved - or forked w/o breaking - trivially CRDT as it holds everything - except for merges - but reputation system can help here - more reputation applied first - conflicts ignored but returned - dislikes can clean conflict or even invert if problem is in richest branch

- three levels - transport - no semantics - consensus - reputation semantics, much richer than merkle, application can take adv as Git example - permanent partition - but both working and even communicating (inefficient though) - application - no guarantees - also reordering

- LIMITATION - space - prune on ¿50%, guaranteed to persist

4.1 P2P Git SCM

- source files - wikipedia pages

4.2 A Wiki

- follow links?

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] 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.

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

[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.

PLACE PHOTO HERE

Michael Shell Biography text here.

John Doe Biography text here.

Jane Doe Biography text here.