BitWeav — scalable and decentralised peer-to-peer micropublishing

Liam Edwards-Playne <liamz.co>
http://bitweav.org

July 29, 2013

Abstract

In the last five years *Twitter* has demonstrated the value of micropublishing as a communications medium. Conversations of only 140 characters in each message, in combination with casual categorisation using tags, meant that strangers could relay ideas to a grand audience with magnificent ease. But the service is not without its faults – its proprietary and centralised nature went against its reliability and sustainability as a platform which many used each day. Similar projects have arisen to create an open, secure and decentralised micropublishing alternative, but have shared a key fault which is they have not facilitated autonomous discovery of content, akin to *Twitter*'s trending topics. In this paper I present a design for a completely decentralised and peer-to-peer micropublishing system called *BitWeav*, that rivals *Twitter* and complements decentralised protocols such as *Tent*. It utilises a scalable publish-subscribe design called *PolderCast*, that facilitates publishing and subscribing to topics without 'rendez-vous' nodes. The messaging aspect supports threading and maintains the authenticity, integrity and completeness of individual messages. Finally this functionality is facilitated by a small set of six RPCs, responsible for overlay maintenance (*gossipCyclon*, *gossipVicinity*, *gossipRings*), message distribution (*publish*) and message retrieval (*getMessage*, *getRecent*).

1 Introduction

Twitter

In the last five years *Twitter* has demonstrated the value of micropublishing as a communications medium. The fundamental principle of the platform, which is the compaction of ideas into 140 characters, has sparked a revolution in how we communicate. Users could spark a conversation on topics (tags) with strangers, discussing anything and everything in real-time. These informal exchanges, combined with the real-time nature of the platform, allowed for information to be relayed quickly and with ease. Tags allowed for these messages to be loosely categorised based on topic, and the trending topics feature facilitated the discovery of topics and ideas that would otherwise be concealed. We could see this in events such as the Egyptian Revolution of 2011 and Occupy Wall Street, which were largely provoked by conversations on *Twitter*.

Such an invaluable tool is not without its flaws though. To begin, *Twitter* was not heavily suited to threaded conversations at first, and only now has some support for them. It would be valuable to have both a mechanism for individual threads within a topic, and individual replies within those threads. The current mechanism for replies brings about additional computational stress; to find the previous reply in a thread you must determine what time the last message was sent from that user.

Aside from this there was also the system itself, which is proprietary and centralised, which contributes to multiple issues:

- Due to the platform being closed-source, there are multiple security concerns which can be raised as it doesn't permit proper security reviews — this can lead to unknown compromise of user data.
- *Twitter* tracks user behaviour for targeted advertising over multiple services, which is a significant privacy issue for most people.
- The API has limited availability due to the costs associated with providing its services freely. This hinders external development and makes some extensions difficult/impossible to realize.
- The lack of openness means that users have a lack of control over their data.
- The service is completely centralised, which raises concerns about the stability and reliability of its services and its vulnerability to censorship.

It is for these reasons that a decentralised and peer-to-peer micropublishing alternative would be desirable. There exist two currently available designs that facilitate decentralised micropublishing: *FETHR* and *Tent*.

Alternatives

FETHR. An early alternative that stimulated my research into this area and subsequent creation of this project was *FETHR* [5]. *FETHR* is a design for an open decentralised micropublishing protocol that runs atop HTTP. It improves on *Twitter*'s semantics by including fields for threading, and achieves decentralisation by having each user maintain and serve their own profile. The concept seemed sound but it lacked any tags or discovery mechanisms for content, and instead relied on an RSS-like manual approach to sourcing content.

Tent. A similar project that recently has been very popular after release of a public implementation is *Tent* [1]. *Tent* is a communications protocol that runs atop HTTP like *FETHR*. This protocol is more generic to different types of content though, as it focuses on being a all-encompassing decentralised social protocol. It improves on *FETHR* through additional types for all sorts of content, a more strictly defined publish-subscribe protocol and HTTP header discovery of hosts (more autonomy). *Tent* is promising, but still lacks this autonomous content discovery mechanism.

BitWeav

While both *FETHR* and *Tent* have provided the mechanisms for decentralised social content storage, **their fatal flaw is that they have not facilitated autonomous discovery of content** — both rely on the user to manually enter their subscriptions, and have no method of categorising content with tags (as *Twitter* does) nor discovering content using a mechanism such as 'trending topics'.

With this in mind, I set out to build just this — a design for a decentralised and peer-to-peer micropublishing network that would function as the missing piece to decentralised social protocols on the Internet. It would be a peer-to-peer overlay orientated around a set of topics, wherein nodes could publish short messages (akin to 'tweets') and subscribe to such topics.

The idea behind these short messages is that *BitWeav* would act as a generic discovery/distribution mechanism for both messages published to the network (in the micropublishing sense) and URLs of resources from decentralised social protocols like *Tent*.

2 Objectives

To guide development of *BitWeav* I devised a set of objectives that outlined the key features necessary for a successful design.

1. **Decentralised**: we should not rely on any set of nodes to provide for any other set of nodes, in the circumstances where if their connection were severed it would produce irreparable damage to the network.

2. **Open**: we should rely totally on open technologies and likewise publish all concepts/implementations under a fair license. Source code should be open-source, the user should have control over their data and networked API services should be provided freely.

3. **Simple**: the theory and implementation should be as simple as possible. The content distributed by the system must be contained within a single type — the message. All messages, contacts and topics must be uniquely identified by a single data type — the ID.

4. **Easy to use**: the concepts (messages, replies etc.) should be similar to those found in other social platforms, to make it more approachable for users.

5. **Timely**: messages should be distributed to subscribers promptly; hard real-time guarantees are unnecessary, but delays of more than a few minutes will severely impact the utility of a micropublishing system [5].

6. **Scalable**: the overlay network should use a non-gossip based approach to publishing messages. Parameters regarding dissemination fanout should be configurable by nodes for more efficient operations.

7. **Topic-enabled**: messages should be able to be loosely categorised under multiple topics using tags, and distributed to only those who are interested.

8. **Threaded**: messages should have fields to facilitate threading, wherein they can be in reply to other messages, and descend from a common thread (the root).

9. **Secure**: the authenticity, integrity, and completeness of public content sent within the network must be preserved by the system and independently verifiable by nodes. Replay attacks must be defended against.

10. **Sufficient for URLs**: the maximum length of a message should be sufficient for holding a single URL, and at most a magnet link for a torrent.

3 Contacts and messages

At the core of *BitWeav* are contacts and messages. Contacts are the users who access the *BitWeav* network, and messages are the basis of their communications.

Contacts

A contact is a user accessing the *BitWeav* network. Contacts authenticate using a **public/private key-pair**. They are identified by the computation of an identification function (described in ch. 4) over the public key, which is referred to as their **address**. Since the public key is mathematically bound to the private key, this process makes forging an address infeasible. Removing the ability for nodes to generate specific addresses also improves resistance against chosen-key attacks on the overlay network (more on this in ch. 8).

Finally, it is important to note that contacts are separate from the nodes (network hosts) that they operate from. The network operation of nodes is described in chapter 7.

Messages

A **message** is a generic container for micro-content. Micro-content is the basis of the micro-publishing idea: when content is small, it introduces constraints on users to relay more information into fewer words.

The bulk of the message is the **content**. The content is a maximum of 200 bytes in size and is interpreted as an array of *UTF-8* characters. *UTF-8* was chosen as the encoding, as it is a standardised format that has wide usage on the Web and has sufficient support for other alphabets. The 200 character limit (for Western alphabets that is) was chosen because while 140 characters has shown sufficient for expressing ideas in threaded communications (text messages, *Twitter* 'tweets'), I believe 200 characters is more suitable for content containing non-shortened URLs.

It is also necessary to include a language field, to ensure true multilingual support. The **language** field is based on the IETF specification for language codes [2]. It is a UTF-8 string of maximum 13 characters in length, which supports all present languages and scripts, as well as any future ones too. When viewing received messages, I advise that they be filtered according to the user's language.

Now I come to describing the content, whose markup is inspired by the simplicity of *Twitter* 'tweets'. To improve the uniformity and thus quality of content, all breaking and special characters (newlines, tabs etc.) are filtered out from messages by nodes on the network (the mechanics of these incentives are described in ch. 7.3). There are two types of markup supported, which are **hashtags** and **URLs**. Hashtags, which are space-delimited strings that begin with a hash '#', are used to identify topics that a message is associated with. A message may have multiple hashtags. The other form of markup which is supported are links, which are *URL*s that are automatically hyperlinked for all protocols, an important step for improving interoperability — this facilitates the linking of many types of content, including web pages, torrents (using the *magnet URI* scheme [9]) and *Tent* profiles.

The final part to the message is its authentication section, which contains two mandatory fields. The first is the **from** field, which is the public key of the publishing contact. In conjunction with the second field, the **signature**, which is a digital signature computed over the remainder of the message, this authenticates the author of the message, regardless of its sending node. Because of this, any node can forward a message they did not construct, to others, which is an essential feature of *BitWeav*'s message distribution strategy.

These features alone can provide something similar to *Twitter* 'tweets' but still lack any ability to do threading. Before I explain how this is implemented, I must first describe how the identification mechanism works.

4 Identification

As a requirement of the simple objective, we must have a single system of identification for contacts, messages and topics. I define a type called the **ID** to identify all of these items.

The purpose of the ID is to provide a unique fixed-size identifier for a contact, message or thread. I make the distinction between **basic IDs** — the underlying data type which is exchanged between computers, and **human IDs** — representations of this data for transferral between humans. The important difference between the two is that human IDs have additional steps for computation and encoding, including a recognisable version/identifier, a checksum to guard against input errors and Base58 encoding for ease of input. My human ID approach is the same algorithm used for Bitcoin address generation [8].

Basic ID generation

1. Having some arbitrary array of data as input to the *ID* function — the content.
2. Perform SHA2-256 hashing on the content.
3. Perform RIPEMD-160 hashing on the result of the SHA2-256 hashing.

Human ID generation

1. Given the result of basic ID generation from above as input to the *humanID* function.

2. Prepend the version byte 0x49 or "W" to the basic ID.
3. Perform a double SHA2-256 hashing on the extended basic ID.
4. Append the first 4 bytes of the double SHA2-256 hashing (the checksum) to the extended basic ID.
5. Encode in Base58 for human representation.

For **contacts**, the ID is computed over the contact's public key. For **messages**, the ID is computed over the entirety of the message. Lastly for **topics** the ID is computed over the hashtag itself.

5 Threading

There are two aspects to the threading of messages. The first is **replies**, wherein a message is in response to another, and the second is **threads**, wherein messages are part of a specific item of discussion on a topic.

Replies. As each message can be identified using its ID, it is simple to implement reply-based threading. I define a field **reply**, which maps to the ID of the message that one is in reply to.

Threads. Threads are trivial. Firstly I define another field called the **root**, which maps to the ID of the original stimulus message for that conversation. A special use of the root field is when a contact is publishing the message on its own profile, wherein the root field will be equal to the contact's address and the message will be in reply to the ID of the last message the contact had published.

This leaves us with only one problem, which is that of message ordering. The *reply* field aids to develop a lossy ordering of messages. When we receive a message on the topic and thread we are listening on, we append it to the conversation thread according to which message it was in reply to.

Due to the peer-to-peer nature of the overlay network, threaded messaging may be impeded by messages not being sent — unintentionally or on purpose. When we receive a message that is in reply to another message we have no record of, we contact the sending node to get the message. If the node does not respond, we discard their message and blacklist them. This method dissuades spammers from sending messages with falsified reply fields, as they will be flooded with queries and subsequently blacklisted.

In this section I have outlined two optional fields which can be used to implement traditional threaded conversation and contact profiles: the reply and root fields.

6 Security

A principle objective of *BitWeav* is to be secure; The idea that security should be ubiquitous is fundamental to its design. Nodes must communicate via a secure reliable transport protocol to preserve the authenticity, integrity, completeness and privacy of data sent within the network. It should be resistant to replay attacks.

This aspect is implementation dependent, and is intentionally not described here as it is out of the scope of this paper.

7 Overlay network

The principle objective of the overlay network is to **distribute messages to interested nodes in a scalable and reliable manner**, a design called a publish-subscribe system. I considered a couple of publish-subscribe systems, namely *SCRIBE* [4] and *PUB-2-SUB* [7], however I found their shared weakness to be relying on nodes which aren't necessarily interested in certain topics (so called 'rendez-vous' nodes) to aid in the distribution of messages on that topic.

I eventually discovered *PolderCast* [6], a recent publish-subscribe system from 2012 that achieves this objective. The critical advantage of PolderCast lies in its gossip-based network design that is resilient to node churn while requiring only minimal node state necessary for routing — both publishers and subscribers are allowed to join and leave at any moment, without any prior notice, due to topic subscription being implicit in the routing process. It also features a layered network module design that proves useful when developing additional functions for the overlay network.

In this section I will outline the design of *PolderCast* and how I use it in conjunction with message distribution/retrieval.

7.1 PolderCast

The overlay network is composed of nodes and topics, both being uniquely identified by an ID (a basic ID as described in ch. 4). **Nodes** are networked hosts that communicate with other nodes using a simple query-reply RPC framework. Nodes organise themselves around **topics**, to which they can subscribe to and publish messages on. At a conceptual level, each topic is modelled as a ring that connects all corresponding subscribers of it and only them. These rings are then augmented by random links across the topics which produces a single, connected and navigable overlay.

The basis to the management of the overlay are three layered network modules: *Cyclon*, *Vicinity* and *Rings*. Each module maintains its own routing table (called the view), managed by a separate gossiping protocol, which gossips periodically, asynchronously, and independently from the other two modules.

The **view** stores a list of node profiles, each annotated by an age value. A node Q being a **neighbour** of node P means that P has a copy of Q's profile in the respective module's view. A node's **profile** contains:

i. its network details.

ii. the public key of its contact, and subsequently from this its unique ID or address (which is derived client-side).

iii. its interests, which are a signed and timestamped copy of the IDs of topics the node is subscribed to, each annotated with a priority that node assigns to finding neighbours of that topic.

   Clearly the interests of a node may vary over time, so this section of a node's profile is timestamped so nodes may identify newer copies. It is also signed for obvious reasons of security.

I also maintain two variables for each module: the view size V and gossip size G. The **view size** defines the maximum number of neighbours to maintain: $V = 20$ for Cyclon and Vicinity, and $V = 4$ for Rings. The **gossip size** defines the maximum number of neighbours to be included in a gossip message, and is by default 20.

Finally, the network is based on a relatively small set of functions that provides for overlay network maintenance, message distribution and message retrieval, which I will proceed to describe.

7.2 Overlay network maintenance

As briefly mentioned, we have three modules, each with their own gossiping protocol. Each module's gossiping protocol is based on the concept of a **shuffle**, a process whereby nodes initiate neighbour exchanges periodically, yet not synchronized, with their oldest neighbour at a fixed period $T = 10s$. The interval time T should be larger than twice the expected worst-case network latency plus processing delays as to prevent sending a gossip message before a response from the previous one.

I describe this shuffling process for each module for an initiating node P. The common rules following gossiping are as follows:

1. Whenever a node gossips with another neighbour Q, it temporarily removes Q from the respective module's view, anticipating that it will respond and will be inserted anew.

2. Whenever a node sends a message, it increments the age value of all nodes in the respective module's view.

3. A neighbour's age is retained when that neighbour is handed from one node to another.

I define three RPCs in this category: **gossipCyclon**, **gossipVicinity** and **gossipRings**.

7.2.1 Cyclon

This module operates at the lowest-level, being situated above the layer where network communications take place. It forms an overlay and maintains random links between nodes using an epidemic algorithm. Each node knows a small, continuously changing set of neighbours, and occasionally contacts one to exchange some of their neighbours.

Shuffling

1. Select a neighbour Q with the highest age among all neighbours, and a subset of $G - 1$ other neighbours.

2. Replace Q's entry with a new entry of age 0 and with P's details.

3. Send the updated subset to Q.

4. Receive from Q a subset of G entries.

5. Discard entries pointing at P and nodes that P already knows of.

6. Verify entries are valid by establishing a connection to such nodes and discard any invalid entries.

7. Update P's view to include all remaining entries, by firstly using empty slots (if any) and secondly replacing entries among the ones sent to Q.

On reception of a shuffling query a node Q replies by sending back a random subset of at most of G its neighbours, and updates its own view to accommodate all received entries.

Bootstrapping As *Cyclon* is the lowest-level module, it is where we perform bootstrapping. To bootstrap the application, the bootstrap node is inserted into the *Cyclon* module's view and three initial shuffles are performed with the node (to reliably simulate a random walk, as per the original design).

7.2.2 Vicinity

This module maintains random links between nodes that share interest in one or more topics. Such links serve as input to the *Rings* module and are used by the dissemination protocol to propagate events to arbitrary subscribers of a topic.

Shuffling

1. Let P select its oldest neighbour Q for gossiping.

2. P merges its views from all three modules.

3. P replaces Q's entry with its own in this new merged view, setting the age to 0.

4. P selects the G nodes closest to Q by applying the proximity function on its behalf, and sends them to Q.

   The proximity function is designed to ensure that the *Rings* module is supplied with arbitrary neighbours for all its topics. In that respect, candidates subscribed to topics annotated with higher priority by the target node are ranked closer compared to candidates of lower priority topics. Among candidate nodes that rank equally in terms of topic priorities, proximity is determined by the number of topics shared with the target node: the more shared topics, the closer their ranking.

5. Receive from Q its G closest to P nodes.

6. Merge these into the view and rebuild it by selecting the G closest nodes and discarding the others.

Upon reception of a shuffling query, Q merges the received neighbours with the union of all its views, and updates its *Vicinity* view to the V closest neighbours. It then repeats steps 2 to 4 for P.

7.2.3 Rings

This final module discovers a node's successor and predecessor for each topic in its subscription, and quickly adapts to new successors/predecessors in dynamic networks.

Nodes are placed into rings in the order of their node IDs. Each node maintains G neighbours for each topic in its subscription: $V/2$ with lower and $V/2$ with higher ID. It periodically picks a node from its *Rings* view, and the two nodes exchange up to G neighbours to help each other improve their views.

The sorting function used to determine the lower/higher nature of an ID is simply a comparison between two IDs (which are treated as 160 bit integers) to find the largest number.

Shuffling

1. Let P select its oldest neighbour Q for gossiping.
2. P collects all subscribers of topics which P and Q have in common, considering the union of views of all three modules.
3. P sorts them by ID, and for each topic in common with Q it selects the $V/2$ ones with just lower and the $V/2$ ones with just higher ID than Q's ID.
4. If more than G nodes have been selected, it randomly picks G of them.
5. It sends the selected nodes to Q, and Q does the same in return.
6. Upon reception of such nodes, P merges them with the current view for the topic, and it selects the $V/2$ ones with just lower and the $V/2$ ones with just higher ID than P's ID.

7.3 Message distribution

There are three categories of topics that nodes may publish/subscribe to:

1. Hashtags
2. Threads
3. Contacts

I define the **publish** RPC, which contains a message object and provides a simple mechanism for distributing messages independent of author. The protocol for disseminating messages is characterised by a dissemination fanout F, which is typically $F = 2$. A node receiving an event for topic T for the first time, propagates it F times, depending on the following rules:

1. If the event has been received through the node's successor (or predecessor), it is propagated down the line to its predecessor (or successor) and $F - 1$ arbitrary subscribers of T.
2. If the event was received through some third node, or if it originated at the node in question, it is propagated to both the successor and the predecessor, as well as to $F - 2$ other subscribers of T.

3. If a copy of this event has already been received in the past, it is simply ignored.

Rules

Nodes follow certain rules regarding the publishing of messages. I employ a novel incentive to dissuade nodes from publishing unrelated messages on topics: when a node P receives a *publish* RPC from a node Q, and the message is not suited for that topic (the requirements of which I describe below), P does not forward that message and immediately terminates its connection with Q.

A node must check all the requirements for each message, something relatively cumbersome but definitely necessary for a high-quality content network. A node must only broadcast to a topic if:

1. The topic is the ID of one of the hashtags in the message.
2. The topic is the thread that the message is published on.
3. The topic is the ID of the *from* field, and therefore is a profile message.

The incentive here is that if a node does not filter messages, it will be evicted from the network by honest nodes.

7.4 Message retrieval

A significant and intentional aspect of *BitWeav* is the temporary nature of its content. A node only receives messages when it is part of the network. When a node is disconnected for a period of time, the only means to accessing old messages is to request them from nodes who might of kept them, if any (this is discussed further in ch.8). There is no requirement for them to be stored permanently, however the protocol has certain constraints on when a message must be stored temporarily.

When a node P is listening in on a thread, and another node Q sends it a message in reply to another message that P has no record of, P will request that message from Q. In this circumstance, if Q has published a message without retaining it, both Q and the message will be dropped by P. This is also true if Q has published a message in reply to a non-existent message. This mechanism whereby a node is dropped if they do not temporarily store a message they have replied to produces the incentive for nodes to retain messages and thus improve quality of service.

RPCs

The RPC a node will use to request a specific message, perhaps in a circumstance when they may have missed a message on a thread, is the **getMessage** RPC. This requires only a message ID and returns the message object.

The other RPC a node might use when they have recently rejoined the network and are looking for recent updates on topics is the **getRecent** RPC. This RPC takes an ID of a topic and returns a list of relevant messages.

8 Design analysis

In this section I will discuss some of my design decisions and also some issues I have identified.

Chosen-key attack. The PolderCast design uses the sorting function to determine who are the node's successors and predecessor for each topic. If a node were to precisely control what ID they have, they would be able to insert themselves into another node's Rings module with relative ease and disrupt the publishing of messages. By basing the ID of a contact, and thus the node, on the ID of the public key, nodes are not able to precisely control their ID.

Omission attacks. Nodes may damage the network by deliberately omitting messages from being published. This is somewhat averted by also forwarding the message to $F - 1$ arbitrary subscribers of the topic, but it is not a fully-fledged solution. Nonetheless, this is a problem yet to be solved in P2P, and basing my critique on its prevalence in existing systems, it is unlikely to be a huge issue.

Threaded messaging. The threaded messaging implementation allows for a lossy decentralised ordering of messages, but it is susceptible to nodes injecting messages into older parts of the message sequence. A rolling metric could be employed to decide whether a message is recent enough: as the time differences between messages becomes smaller (and network synchronisation becomes harder), the accepted range for recent messages becomes larger.

Adapting quickly to subscription changes. The subscriptions of a contact may change quite frequently, something the PolderCast may not be able to adapt to quickly. Given that subscription changes means that a node's profile changes, the speed at which nodes adapt to subscription changes of others is the rate at which node profiles are exchanged. The modules in which node profile's are exchanged, *Cyclon* and *Vicinity*, exchange quite frequently at $T = 10s$, and independently too. Although this is quite frequent, it is not responsive enough for general user interactions. The issue is subject to user interactions, and more attention will be paid to it when *BitWeav* is built and we can gauge whether work will need to be done.

A possible solution would be to create a new shuffle message specifically for finding nodes subscribed to a topic, and inserting them into the routing table for a 'quick join'.

Message persistence. Despite its similarity to the functionality of *Twitter*, the persistence of messages in *BitWeav* is not guaranteed. To best my knowledge, the only peer-to-peer system that has provided a strong incentive for nodes to store content is the *Bitcoin* network, which incentivises the processing and storing of transactions (mining) through digital income in the form of Bitcoins. As *BitWeav* is a free content network, there is no basis for a reward, so instead we must trust nodes to store messages.

One feasible solution would be to enhancing the user experience with centralised storage pattern. Contacts would use a HTML <link> tag on a web page

to reference their profile. This would allow a user to add a contact by simply pasting a link to a website. The linked profile would also have an archive of that contact's messages, which can be served to other nodes when the contact's node is offline. This concept will most likely be implemented in the proof of concept, but has intentionally not been described here as its implementation is dependent upon a number of aspects which extend the scope of this paper.

Spam prevention. Naturally, there is an issue of spam in such an open network, both from individuals and large clusters of nodes (so called 'spam farms'). There are multiple methods to which spam may be prevented.

A node could introduce rate limiting as the volume of messages they receive from other nodes increases. This method would be dependent on determining a fair 'rate', but would be relatively cheap in terms of performance. As nodes maintain a relatively small routing table, recording statistics would not be very costly. A disadvantage of this method would be that it is still vulnerable to large collections of nodes conspiring to spam another.

An alternative method that has proven somewhat effective is *Hashcash* [3]. *Hashcash* can prevent spam farms by requiring nodes to pay a computational cost when sending messages. This cost can be later adjusted to account for developments in computing efficiency. *Hashcash* could be effective in limiting the pace at which spam farms send spam messages, but does not prevent individual spam.

A combination of these two methods could provide a computationally cheap method of subverting spam farms. In addition to rate limiting, random messages from nodes could be tested using Bayesian filtering, an effective statistical technique for detecting spam. While no peer-to-peer network has been devised that effectively prevents spam, I believe that a combination of these methods could reduce it. Spam prevention may be integrated into the *BitWeav* design at a later point, but my attention is diverted for now by the initial proof-of-concept implementation.

9 Conclusion

I have presented a design for a completely decentralised and peer-to-peer micropublishing system that rivals *Twitter* and complements other decentralised social protocols like *Tent*. This design facilitates topic tagging and threading, and each message is authenticated by the node that sent it using asymmetric cryptography. Each message, contact and topic is identified by a single data type called the ID, and this ID can be represented simply for human recognition.

All of these concepts are combined with a novel approach to maintaining quality content and improving the user experience. Content is filtered from topics if it is not on-topic and nodes are required to temporarily store messages in a thread if they are to reply to them. Additionally some mechanisms have been described which could reduce message spam significantly at minimal cost.

Finally this functionality is facilitated by a small set of six RPCs, responsible for overlay maintenance (*gossipCyclon*, *gossipVicinity*, *gossipRings*), message distribution (*publish*) and message retrieval (*getMessage*, *getRecent*).

References

[1] Tent – the protocol for decentralized communication and evented data storage. `https://tent.io/about`.

[2] Shervin Afshar. What data type should I use for IETF language codes? — StackOverflow. 2013. `http://stackoverflow.com/a/17863380/453438`.

[3] Adam Back. Hashcash – a denial of service counter-measure. 2002.

[4] Antony Rowstron, Anne-Marie Kermarrec, Miguel Castro, and Peter Druschel. Scribe: The design of a large-scale event notification infrastructure. In *Networked group communication*, pages 30–43. Springer, 2001.

[5] Daniel R. Sandler and Dan S. Wallach. Birds of a fethr: open, decentralized micropublishing. In *Proceedings of the 8th international conference on Peer-to-peer systems*, IPTPS'09, pages 1–1, Berkeley, CA, USA, 2009. USENIX Association.

[6] Vinay Setty, Maarten van Steen, Roman Vitenberg, and Spyros Voulgaris. Poldercast: Fast, robust, and scalable architecture for p2p topic-based pub/sub. In *Middleware 2012*, pages 271–291. Springer, 2012.

[7] Duc A Tran and Cuong Pham. Pub-2-sub: A content-based publish/subscribe framework for cooperative p2p networks. In *NETWORKING 2009*, pages 770–781. Springer, 2009.

[8] Bitcoin Wiki. Technical background of version 1 bitcoin addresses. 2013. `https://en.bitcoin.it/wiki/Technical_background_of_Bitcoin_addresses`.

[9] Wikipedia. Magnet URI scheme — Wikipedia, The Free Encyclopedia . 2013. `http://en.wikipedia.org/wiki/Magnet_URI_scheme`.