BitWeav — scalable real-time decentralised and peer-to-peer communications

Liam Edwards-Playne <liamz.co>

June 1, 2013

Abstract

1 Introduction

In the last half-decade we have seen how the micro-blogging atmosphere of *Twitter* has proven the value of online communication. The fundamental principle of micro-blogging, which is the compacting 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. Through trending topics, this public yet often concealed discussion platform which is online communication could give rise to the sharing of influential ideas and the stirring of the masses. We could see this in events such as the Egyptian Revolution of 2011 and Occupy Wall Street. Such an invaluable resource must be kept sustainable, but we have seen that due to the centralised nature of this tool, as well as its issues with corporate control, there is a need for an alternative.

A similar, older and more sustainable platform is *IRC*. Like *Twitter*, *IRC* facilitated public text-based discussion on topics (channels). It had its advantages and disadvantages as it used a moderator-based group discussion device called a channel, but the user bases of such were fragmented across differing *IRC* networks. Nonetheless, its more advanced organised discussion led to it becoming the key communications tool for 'hackers'. Hackers found the setup requiring minimum personal details, in combination with the simplistic and programmable interface which is the *IRC* protocol, allowed for high-quality discussions that focused on the content.

Finally there was *Skype*. The unique feature of *Skype* was its alternate medium for communication — video/audio chat. It allowed individuals to talk privately and securely with ease. Perhaps the most significant but relatively unknown aspect of *Skype* was its decentralised and peer-to-peer network architecture, which showed that a decentralised communication system is possible. For the initial period its communications were completely secure, something that wasn't before brought to the market. Later however its image was tarnished by confirmations of multiple backdoors [1, 2] existing for authorities to wiretap communications — its fault laid in the proprietary nature of the protocol and the closed-source client.

I set out to build an alternative, that would combine the features of *Twitter*, *IRC* and *Skype* while maintaining an open protocol that was decentralised and peer-to-peer. It was to be an experiment in decentralised peer-to-peer innovation, as well as a product in the communications market.

2 Objectives

In this paper I describe both the theory of *BitWeav* and its premier implementation called *Weav*. To guide the development of *BitWeav*, I formulated a list of objectives that expressed the key features necessary for its completion.

1. **Decentralised**: we must 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 must rely totally on open technologies and likewise publish all concepts/implementations under a fair license.

3. **Simple**: the theory and implementation must 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**: like Twitter and Skype the initial concept must be easy to understand for a user of these platforms, and likewise so must the user experience.

5. **Topic-enabled**: messages must be able to be categorised under multiple topics using tags, and distributed to those who are interested.

6. **Threaded**: threaded messaging must be inbuilt, wherein messages can be in reply to other messages, and descend from a common root (a thread)

7. **Privacy-enabled**: private distribution of messages must be supported, whereby only specific contacts will receive messages on a specific topic. Such messages should have perfect forward secrecy.

8. **Media-enabled**: contacts must be able to privately communicate through other mediums like video and audio.

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.

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 generic term for machines and users 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.

The network operation of contacts 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 160 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 160 character limit (for Western alphabets) was chosen because it has shown sufficient for expressing ideas (text messages, *Twitter* 'tweets') in threaded communications.

The content can contain two types of markup: **hashtags** and **URIs**. 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 supported are links, which are *URI*s which can be used to link to resources such as web pages and torrents (using the *magnet URI* scheme).

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 node. 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 all messages regardless of sender. Because of this, any node can forward a message to others, which is an essential feature of *BitWeav*'s message distribution strategy.

These features alone provide something similar to *Twitter* tweets and *IRC* messages, but **lack any structure to the conversation**. Before I explain how threads are implemented, I must first describe how they are identified — the *ID*.

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 hashtag. I make the distinction between **basic ID**s — those which are exchanged between computers, and **human ID**s — those which are exchanged 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 for Bitcoin address generation[3].

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 message's content (excluding the signature, which I describe in chapter 6). Lastly for **topics** the ID is computed over the hashtag itself.

5 Threading

Reply-based threading

There are two aspects to the threading of messages. The first is **reply-based threading** wherein a message is in response to another, which is what Twitter facilitates. The counterpart is **group-based threading**, wherein messages are part of a specific item of discussion on a topic — a thread. *IRC* can be seen to support group-based threading, whereby each channel is a thread of messages. *BitWeav* aims to support both of these.

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

Group-based threading

The other type is group-based threading. In *IRC* conversations, we have channels, which list a multitude of replies in a thread in chronological order. The model I described allows for a lossy form of discussion, but how would this channel concept be implemented?

Firstly I define another field called the **root**, which maps to the ID of the original stimulus message for that conversation. When a node sends a message to a channel, they broadcast on the channel's topic. However this does not allow us to distinguish messages intended for a channel from normal microblogging 'noise'. Instead, we only listen for messages that are on a particular thread.

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. This allows for a lossy decentralised ordering of messages, but it is susceptible to nodes injecting messages into older parts of the chain. To guard against this, we employ a rolling metric 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.

Due to the peer-to-peer nature of the overlay network, group 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. This method dissuades spammers from sending messages with falsified reply fields, as they will be flooded with queries.

In this section I have outlined two optional fields: the reply and root fields. These can be used to implement traditional threaded conversation, like that found in *IRC* channels and *Twitter* replies.

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 transport protocol to preserve the authenticity, integrity, and completeness of data sent within the network. It should be resistant to replay attacks and have perfect forward secrecy enabled at all times. This aspect is implementation dependent.

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* [5] 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 describe my modifications to it.

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 node profiles, each annotated respectively 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 contact details.

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

iii. its interests — 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, $V = 4$ 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 should be larger than twice the expected worst-case network latency + 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 just above the layer where 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 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 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 respective 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.

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.

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

Despite its similarity to the functionality of *Twitter*, the persistence of messages in *BitWeav* is not guaranteed. To best our 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 rely on nodes to store content.

A significant and intentional aspect of *BitWeav* is the temporary nature of its content. Similar to *IRC*, 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. There is no requirement for them to be stored permanently, however the protocol has certain constraints on when a message must be stored temporarily.

For example, 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 Private communications

The final aspect to our communications is private messaging, which I define as a list of interconnected nodes communicating privately on a thread, and unlike public messaging, does not use the PolderCast publish-subscribe model. A node will publish messages to a specific set of nodes for a thread, which I call the **participants**. Such a list of participants is independently maintained by each node in the private thread, so no new nodes are added without each participant reviewing it independently. Additionally, all messages have perfect forward secrecy as due to the transport protocol (see ch. 6), to maintain the privacy of messages sent in the event that the security of one is compromised, a property I have deemed desirable in such a system.

Nodes are given information about new private threads by another node sending an *invite* RPC. From there, they may join by sending *join* RPCs to all the participants. New participants may also be added through firstly being sent an *invite* and then distributing such *join* messages to participants upon accepting the invite.

Inviting nodes.

The **invite** RPC has four fields: thread, participants and publicKey. A private thread is identified by an ID, which is generated by a computation of the ID function over the content of the string "private" appended by some random ID. The **thread** field maps to this ID.

The second field is the **participants**, which maps to an array of profiles for nodes who are to be involved in this thread. Participants won't necessarily know the other participants so we distribute their profiles here.

Finally there is the **publicKey**, which simply is the node's public key, as we cannot be certain that the participants know of the node's key and thus must distribute it.

The procedure for a node P to invite another node Q is as follows:

1. Generate or get the private thread ID.
2. Connect to Q.
3. Construct and send an invite RPC to Q (do not expect a response).

Joining the private thread.

The **join** RPC is very simple, having only the fields *thread* and *publicKey*. The **thread** field maps to the ID of the thread, to distinguish this join from others. The **publicKey** maps to the public key of the joining node — as this RPC is distributed to all the participants of the thread, they may not know of the node already, and thus need this.

The procedure for a node P to join a thread is as follows:

1. Construct a **join** RPC.
2. Connect to each participant Q mentioned in the invite, and send the join RPC to them. Await a response from Q that contains either an indication of acceptance or not, with a reason.
3. Begin listening for messages.

Distributing private messages.

Publishing messages to a private thread is simple. To send a message we do the following:

1. Construct a message.
2. Set the *root* field to the thread ID.
3. Set the *reply* field to the previous message received.
4. Send to all the participants of the conversation.

Receiving a message is just as simple: if the *root* field of the message is the ID of one of the private threads we participate in, and the *publicKey* field is the verified public key of one of the participants, then we append it to user's view.

Video/Audio communications

I won't describe the details of the video/audio communications, because it is outside of this design. Instead I will describe the rationale behind having video/audio communications and the requirements for the implementation.

Rationale. The rationale behind video/audio communications is that is a modern aspect of private communications. Audio is most prominent in mobile communications, and video calling is also gaining support. As well as one of the major inspirations for Weav, Skype, having video/audio communications in-built, this is certainly a wanted feature among people desiring secure private communications.

Requirements. Video and audio calling must support multiple incoming secure audio/video streams in a single private conversation. Such streams must utilise open and free codecs, and must have mechanisms for improvement of user experience (noise reduction, echo cancellation, jitter prevention etc.).

9 Conclusion

Here I presented an open design for a completely decentralised/distributed peer-to-peer communications system that rivals Twitter, Skype and IRC. The messaging system 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.

I have also demonstrated a simple implementation of private messaging that has perfect forward secrecy using a per-message temporal symmetric key. For enhanced communications I have also suggested integrating video/audio calling using WebRTC.

All of these concepts are combined with a novel approach to maintaining quality content, using multiple incentives. 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.

Finally this functionality is facilitated by a relatively small set of eight RPCs: *gossipCyclon*, *gossipVicinity*, *gossipRings*, *publish*, *getMessage*, *getRecent*, *invite* and *join*.

10 Future work

1. Spam prevention using Bayesian filters (fights individuals) and Hashcash (fights spam farms).
2. Integrating a trust framework [4] to fight malicious nodes (omission attacks).
3. Integrating a long-term storage option for message retrieval.
4. Support for off-the-record messaging in the private messaging protocol.
5. Support for display pictures.

References

[1] [cryptography] skype backdoor confirmation. http://lists.randombit.net/pipermail/cryptography/2013-May/004224.html.

[2] Russian security services were given the opportunity to listen to skype. http://www.vedomosti.ru/politics/news/10030771/skype_proslushivayut.

[3] Technical background of version 1 bitcoin addresses. https://en.bitcoin.it/wiki/Technical_background_of_Bitcoin_addresses.

[4] MyungJoo Ham and Gul Agha. Ara: A robust audit to prevent free-riding in p2p networks. In *Peer-to-Peer Computing, 2005. P2P 2005. Fifth IEEE International Conference on*, pages 125–132. IEEE, 2005.

[5] 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.

[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.