Proposed Bitmessage Protocol Technical Paper

Jonathan Warren
bitmessage@jonwarren.org

Revision 1
January 14, 2013

Abstract. The purpose of this paper is to help researchers analyze and critique the Bitmessage protocol before an implementation is completed. Comments and suggestions are welcome.

1. Introduction

Bitmessage is a proposed P2P communications protocol used to send email-like messages to another person or to many subscribers. Non-technical details about how the system would work are described in another document available at: http://bitmessage.org/bitmessage.pdf
With this document and linked source code files, we aim to describe the protocol in enough detail to allow researchers to critique its security.

2. Goals

The goal of the project is to develop a messaging protocol with the following features:

- Decentralized
- Trustless
- Messages well-encrypted
- Messages authenticated
- Not require users to exchange and manage keys
- Hide "non-content" data like the sender and receiver of messages from eavesdroppers. This may be difficult to accomplish.

3. Proposed solution

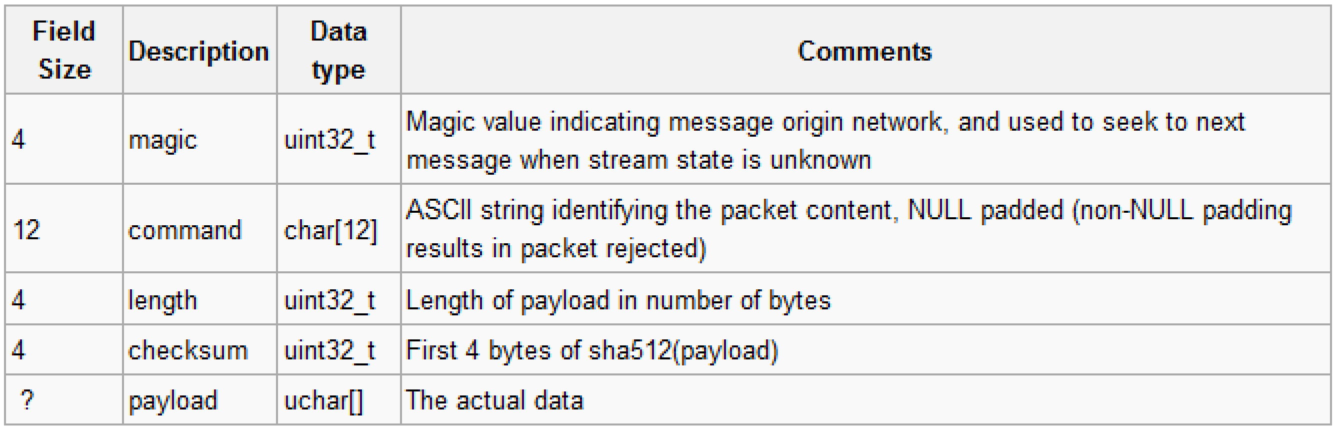
To accomplish these goals, we propose that nodes of a P2P network exchange messages in the same way that Bitcoin nodes exchange transactions: by forwarding them on a best effort basis. Just like with Bitcoin transactions, all nodes will receive all messages. This is meant to hide the receiver of a message, as receiving a message can be a passive process. In practice, however, receiving a message is not passive as the receiver will usually send an acknowledgement. Below we also discuss possible methods of countering attackers who eavesdrop on individual users' Internet connections in order to find out if they are the sender or receiver of a message, or are the owner of a particular identity.

Messages should be signed by the sender of a message and then encrypted for the receiver of the message. Each node will be responsible for attempting to decrypt each message with each of their private keys. To limit the number and size of messages flowing through the network, a proof-of-work scheme is incorporated. A positive side-effect is that this may also limit spam.

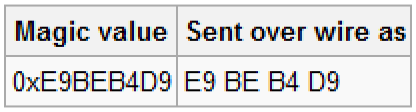
Public keys and requests for public keys are exchanged in the same way that messages are exchanged: by forwarding them through the network. Addresses exchanged by humans are a base58-encoded-hash of their public keys. To send a message, a Bitmessage client requests the public key based on the hash and waits for the public key to arrive through the network. After it does, it can use the public key to encrypt the message bound for the recipient.
In order to scale, nodes self divide into *streams*. This is discussed in more detail in the other document, http://bitmessage.org/bitmessage.pdf.
Because all nodes receive all messages, a natural extension of the protocol is to support broadcasting messages to subscribers. Users may subscribe to an address through their user interface and any *broadcast* type messages sent through the network by that address will be displayed to the user. Receiving a broadcast is an entirely passive process.
Thus there are four types of 'objects' that are propagated throughout a Bitmessage stream: getpubkey, pubkey, msg, and broadcast.

4. Interaction Between Nodes

Every message sent between nodes has this message header:

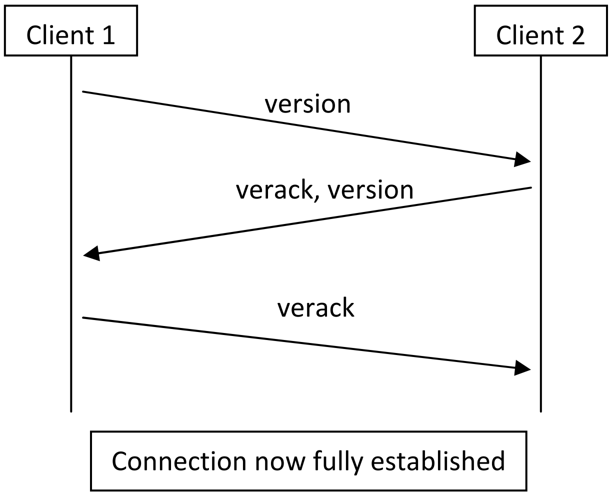


Field Size	Description	Data type	Comments
4	magic	uint32_t	Magic value indicating message origin network, and used to seek to next message when stream state is unknown
12	command	char[12]	ASCII string identifying the packet content, NULL padded (non-NULL padding results in packet rejected)
4	length	uint32_t	Length of payload in number of bytes
4	checksum	uint32_t	First 4 bytes of sha512(payload)
?	payload	uchar[]	The actual data



Magic value	Sent over wire as
0xE9BEB4D9	E9 BE B4 D9

After connecting to a node using TCP, the initiator sends a version message that describes the version of the protocol it is using to the other node. If the other node accepts, it sends a verack packet. The node receiving the incoming connection then repeats the same process itself.



The version message has this format (along with the header above):

Field Size	Description	Data type	Comments
4	version	int32_t	Identifies protocol version being used by the node
8	services	uint64_t	bitfield of features to be enabled for this connection
8	timestamp	int64_t	standard UNIX timestamp in seconds
26	addr_recv	net_addr	The network address of the node receiving this message
26	addr_from	net_addr	The network address of the node emitting this message
8	nonce	uint64_t	Random nonce used to detect connections to self.
1+	user_agent	var_str	User Agent (0x00 if string is 0 bytes long)
1+	stream numbers	var_int_list	The stream numbers that the emitting node is interested in.

This structure uses the following var_int_list, var_str, and net_addr structures:

Variable length integer

Integer can be encoded depending on the represented value to save space. Variable length integers always precede an array/vector of a type of data that may vary in length.

Value	Storage length	Format
< 0xfd	1	uint8_t
<= 0xffff	3	0xfd followed by the length as uint16_t
<= 0xffffffff	5	0xfe followed by the length as uint32_t
-	9	0xff followed by the length as uint64_t

Variable length string

Variable length string can be stored using a variable length integer followed by the string itself.

Field Size	Description	Data type	Comments
1+	length	var_int	Length of the string
?	string	char[]	The string itself (can be empty)

Variable length list of integers

n integers can be stored using n+1 variable length integers where the first var_int equals n.

Field Size	Description	Data type	Comments
1+	count	var_int	Number of var_ints below
1+		var_int	The first value stored
1+		var_int	The second value stored...
1+		var_int	etc...

Network Address

Field Size	Description	Data type	Comments
4	time	uint32	the Time
4	stream	uint32	Stream number for this node
8	services	uint64_t	same service(s) listed in version
16	IPv6/4	char[16]	IPv6 address. The original client only supports IPv4 and only reads the last 4 bytes to get the IPv4 address. However, the IPv4 address is written into the message as a 16 byte IPv4-mapped IPv6 address (12 bytes *00 00 00 00 00 00 00 00 00 00 FF FF*, followed by the 4 bytes of the IPv4 address).
2	port	uint16_t	port number

The **verack** message is blank: it is just the message header above with "verack" as the command.
Now that the connection is fully established, each node should advertise this new node's address to its peers in an **addr** message:

addr

Field Size	Description	Data type	Comments
1+	count	var_int	Number of address entries (max: 1000)
30x?	addr_list	net_addr	Address of other nodes on the network.

It should also send a large addr message to the new peer listing random peers of which it is aware to help it become a better connected node. Thus far, this protocol is almost exactly that used by Bitcoin.
At this point, each node should send an inventory (**inv**) message listing the objects of which it is aware.

inv

Field Size	Description	Data type	Comments
?	count	var_int	Number of inventory entries
32x?	inventory	inv_vect[]	Inventory vectors

This structure references inventory vectors. Inventory vectors are used for notifying other nodes about objects they have or data which is being requested. Two rounds of SHA-512 are used, resulting in a 64 byte hash. Only the first 32 bytes are used; the later 32 bytes are ignored. The rationale for not using SHA-256 is that Bitcoin uses SHA-256 throughout, including for the proof-of-work for blocks. Bitmessage should use a different algorithm so that using Bitcoin mining hardware to do Bitmessage POWs is at least not completely trivial. SHA-512 is used throughout Bitmessage (except for certain places within the encryption implementation). Changing the POW algorithm to one designed to run "poorly" on GPUs and custom hardware would be a positive change but rapidly developing GPGPU hardware makes it difficult to judge what algorithms will be appropriate in the future.

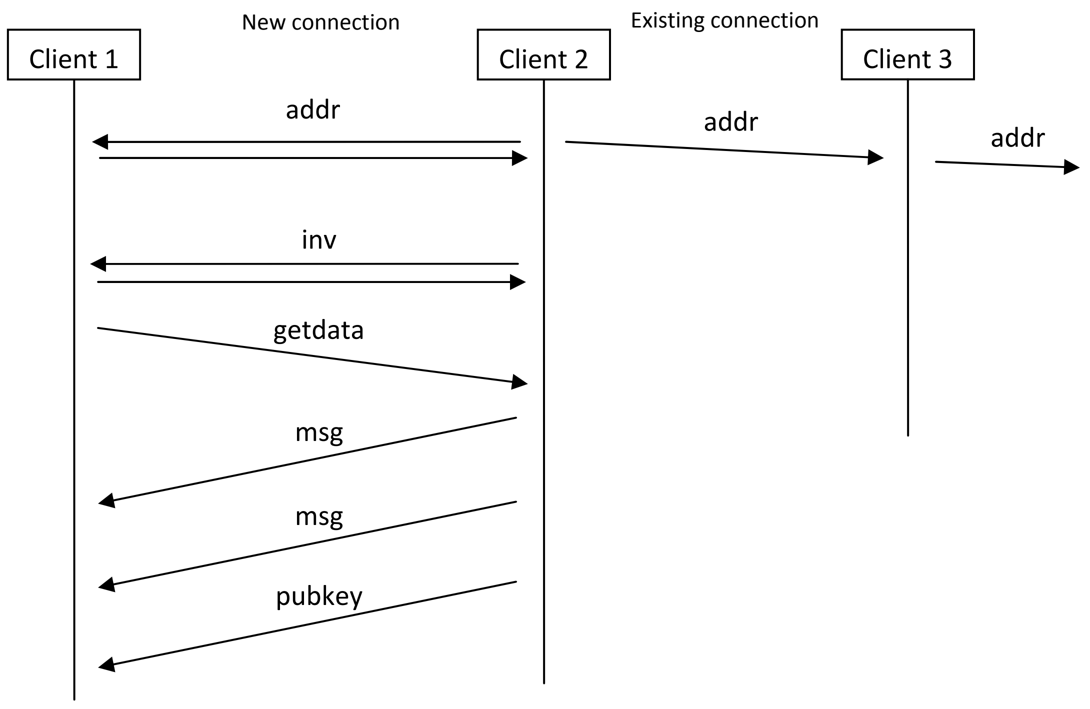
Field Size	Description	Data type	Comments
32	hash	char[32]	Hash of the object

getdata is used in response to an **inv** message to retrieve the content of a specific object after filtering known elements.
Payload (maximum payload length: 50000 entries)

getdata

Field Size	Description	Data type	Comments
?	count	var_int	Number of inventory entries
32x?	inventory	inv_vect[]	Inventory vectors

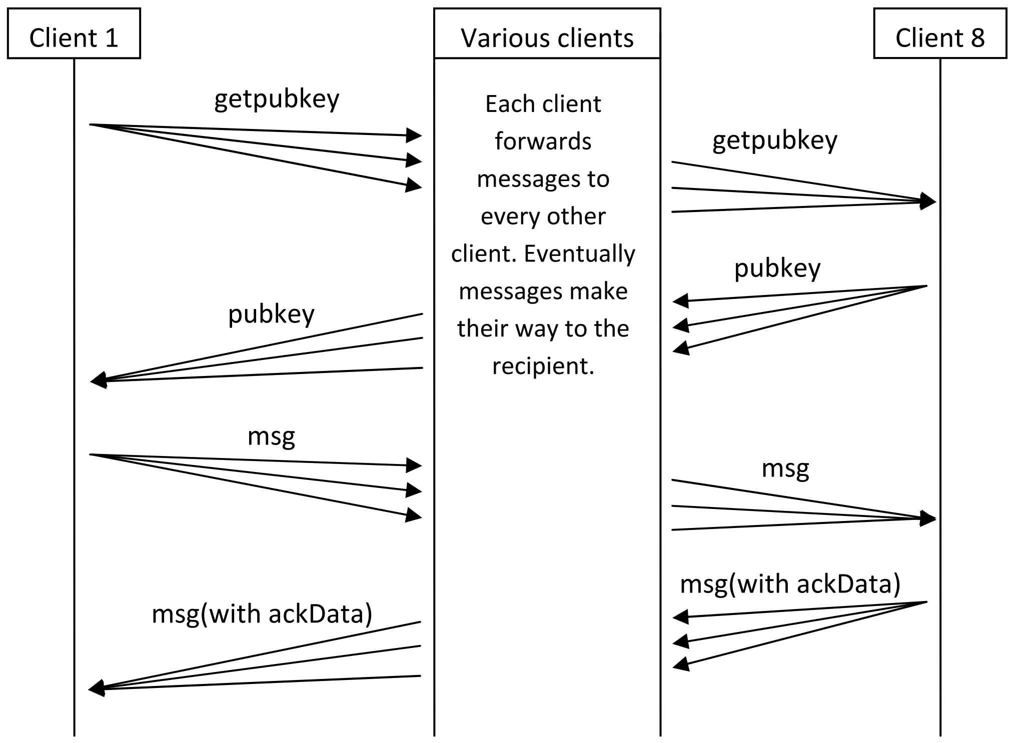
The peer should then send the requested object in a **getpubkey**, **pubkey**, **msg**, or **broadcast** message depending on the type of the requested object.



Using this model, Client 1 downloads all objects stored by Client 2. We propose that each client store objects for about two days and then delete them to reclaim disk space. If the receiver of a message is not online to receive it during the two-day window, the sender will notice that he never received an acknowledgement and will resend the message after waiting an exponentially increasing amount of time.

5. Sending a message

(Pseudocode is below)


Client 1
Various clients
Client 8
getpubkey
Each client forwards messages to every other client. Eventually messages make their way to the recipient.
getpubkey
pubkey
pubkey
msg
msg
msg(with ackData)
msg(with ackData)

Informal description:

For Alice to send a message to Bob, Bob must use a trusted medium to give her his address which is a base58-encoded RIPEMD160 hash of two secp256k1 public keys, one used for signing and the other used for encryption. Alice broadcasts out a request to get the public keys. Upon seeing the request, Bob broadcasts out his public keys which are then stored by all nodes in case they also want to send a message to Bob or if another node requests them. When Alice receives the public keys, she uses her private signing key to sign a message to Bob, then uses Bob's public encryption key to encrypt the message. She then broadcasts the message throughout the Bitmessage stream. Each node attempts to decrypt the message with each of their private encryption keys and also passes the message to peers. Bob decrypts the message then checks the signature using the public signing key included in the message. Finally, he hashes the public key in order to create the base58 address to display in the user interface. By default, he will also send an acknowledgement which is also included in the message from Alice.

Sending a broadcast is simpler: Alice signs a message and broadcasts out her public keys, the message, and the signature in a **broadcast** message. Any nodes that wish to display it may do so.

The formats of each of the four objects types are as follows:

getpubkey

Field Size	Description	Data type	Comments
8	POW nonce	uint64_t	Random nonce used for the Proof Of Work
4	time	uint32_t	The time that this message was generated and broadcast
1+	address version	var_int	The address' version
1+	stream number	var_int	The address' stream number
20	pub key hash	uchar[]	The ripemd hash of the public key

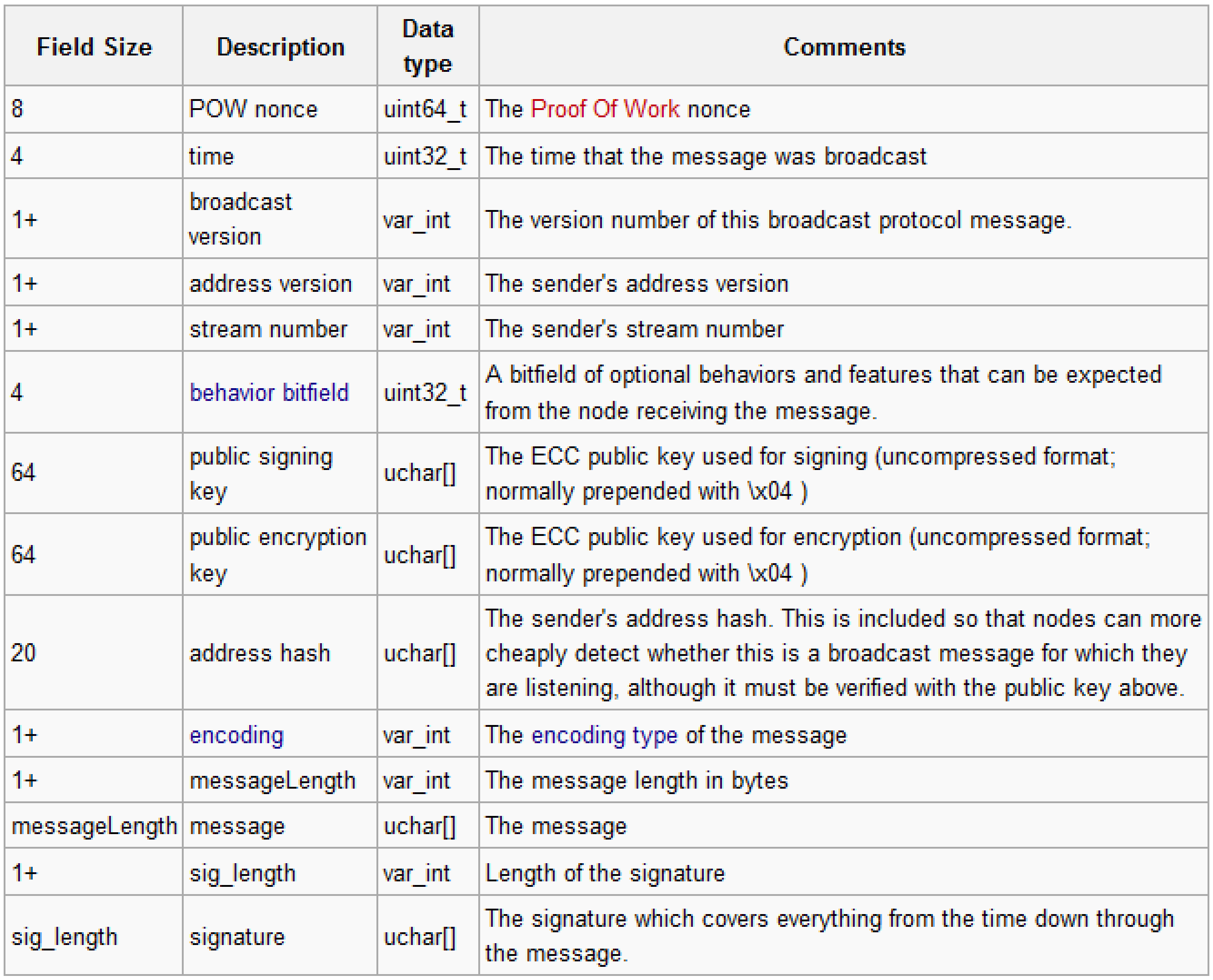
pubkey

Field Size	Description	Data type	Comments
8	POW nonce	uint64_t	Random nonce used for the Proof Of Work
4	time	uint32_t	The time that this message was generated and broadcast.*
1+	address version	var_int	The address' version
1+	stream number	var_int	The address' stream number
4	behavior bitfield	uint32_t	A bitfield of optional behaviors and features that can be expected from the node receiving the message.
64	public signing key	uchar[]	The ECC public key used for signing (uncompressed format; normally prepended with \x04)
64	public encryption key	uchar[]	The ECC public key used for encryption (uncompressed format; normally prepended with \x04)

msg

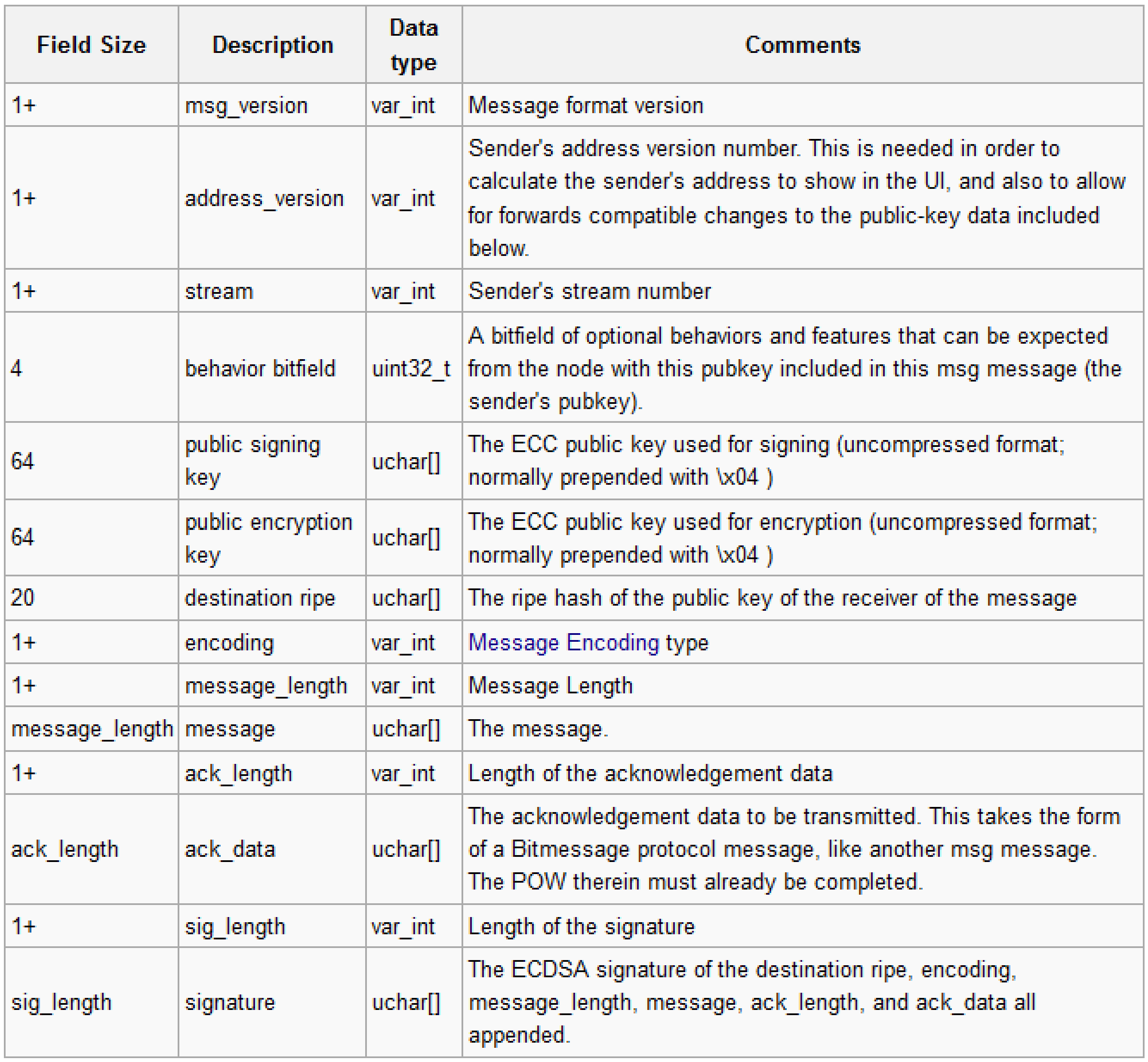
Field Size	Description	Data type	Comments
8	POW nonce	uint64_t	Random nonce used for the Proof Of Work
4	time	uint32_t	The time that this message was generated and broadcast
1+	streamNumber	var_int	The stream number of the destination address.
?	encrypted	uchar[]	Encrypted data. See also Unencrypted Message Data Format

broadcast



Field Size	Description	Data type	Comments
8	POW nonce	uint64_t	The Proof Of Work nonce
4	time	uint32_t	The time that the message was broadcast
1+	broadcast version	var_int	The version number of this broadcast protocol message.
1+	address version	var_int	The sender's address version
1+	stream number	var_int	The sender's stream number
4	behavior bitfield	uint32_t	A bitfield of optional behaviors and features that can be expected from the node receiving the message.
64	public signing key	uchar[]	The ECC public key used for signing (uncompressed format; normally prepended with \x04)
64	public encryption key	uchar[]	The ECC public key used for encryption (uncompressed format; normally prepended with \x04)
20	address hash	uchar[]	The sender's address hash. This is included so that nodes can more cheaply detect whether this is a broadcast message for which they are listening, although it must be verified with the public key above.
1+	encoding	var_int	The encoding type of the message
1+	messageLength	var_int	The message length in bytes
messageLength	message	uchar[]	The message
1+	sig_length	var_int	Length of the signature
sig_length	signature	uchar[]	The signature which covers everything from the time down through the message.

msg messages contain encrypted data. The unencrypted data format is:



Field Size	Description	Data type	Comments
1+	msg_version	var_int	Message format version
1+	address_version	var_int	Sender's address version number. This is needed in order to calculate the sender's address to show in the UI, and also to allow for forwards compatible changes to the public-key data included below.
1+	stream	var_int	Sender's stream number
4	behavior bitfield	uint32_t	A bitfield of optional behaviors and features that can be expected from the node with this pubkey included in this msg message (the sender's pubkey).
64	public signing key	uchar[]	The ECC public key used for signing (uncompressed format; normally prepended with \x04)
64	public encryption key	uchar[]	The ECC public key used for encryption (uncompressed format; normally prepended with \x04)
20	destination ripe	uchar[]	The ripe hash of the public key of the receiver of the message
1+	encoding	var_int	Message Encoding type
1+	message_length	var_int	Message Length
message_length	message	uchar[]	The message.
1+	ack_length	var_int	Length of the acknowledgement data
ack_length	ack_data	uchar[]	The acknowledgement data to be transmitted. This takes the form of a Bitmessage protocol message, like another msg message. The POW therein must already be completed.
1+	sig_length	var_int	Length of the signature
sig_length	signature	uchar[]	The ECDSA signature of the destination ripe, encoding, message_length, message, ack_length, and ack_data all appended.

Pseudocode

To calculate an address:

```
    If using a PRNG:
        private_signing_key = random 32 byte string
        private_encryption_key = random 32 byte string
    Else if calculating an address deterministically using a passphrase:
        private_signing_key = first 32 bytes of SHA512 ( passphrase || "\x00" )
        private_encryption_key = first 32 bytes of SHA512 ( passphrase || "\x01" )
    public_signing_key = calculate public key from private_signing_key
    public_encryption_key = calculate public key from private_encryption_key
    hash = RIPEMD160 ( SHA512 (public_signing_key || public_encryption_key )
    checksum = first four bytes of SHA512 ( SHA512 ( address version || stream number || hash ) )
    address = base58encode ( address version || stream number || hash || checksum )
```

```
To send a message:
    Check that the destination address is valid by checking the checksum
    Check if we already have the recipient's public key. If we do not:
        assemble a getpubkey message and  do the necessary proof-of-work
        for each peer to whom we are connected:
            wait a random amount of time from 0 to 10 seconds
            send the getpubkey message
    Wait for Bob to respond with his public key. After 4 days, make the request again (then again in 8 days, then 16 days….)
    After you receive Bob's public key:
        payload = time || stream number ||  32 bytes of random data
        complete the proof of work for this payload and attach the nonce and msg header to the front of the payload. This payload is the acknowledgement data
        store the payload in a data structure (this will later be checked by the receive_msg function in order to detect when the acknowledgement arrives.)
        Assemble together a new msg message
        Sign the message with your private signing key using ECDH and attach the signature (actual Python code below for this signing step)
        Encrypt the message with the receiver's public encryption key using ECIES (actual Python code below for this step)
        Complete the necessary proof-of-work
        For each peer to whom we are connected,
            wait a random amount of time between 0 and 10 seconds
            send the msg

For each msg message we receive:
    Check the proof-of-work. Abort if insufficient (and move on to the next message if there is one).
    Check the time. Abort if more than 2.5 days in the past or three hours in the future.
    Check stream number. Abort if it is not the stream associated with this connection.
    For each of our peers:
        Wait a random amount of time between 0 and 10 seconds
        Broadcast the msg message
    Check whether this is an acknowledgement bound for us by looking up the msg data in our awaiting_ack_data data structure.
    For each of our private encryption keys:
        Try to decrypt the data. (Actual Python code below.) If decryption is successful:
            Verify that the message version = 1. Abort if not.
            Verify that the sender's address version is one we understand. Abort if not.
            Verify that the destination_ripe matches the ripe hash of our public keys. Abort if not.
            Verify the validity of the signature with the public signing key included in the message. The signature covers the ripe hash of the receiver's public keys, encoding type, message_length, message, ack_length, and ackData, all appended. Abort if signature check fails.
            Store the sender's public keys for optional later use.
            If we have not received this message before, display the message.
            Check that then length of the ackData is greater than 24. Abort if not.
            Check that the magic bytes on the ackData message are correct. Abort if not.
```

Check that the encoded payload length in the ackData message matches the actual length. Abort if not.

Add the ackData to a list of messages to process once all other messages from this peer have been processed. We will process it by following this same function. By default, we, of course, won't be able to decrypt it because it is 32 bytes of random data, but users may use actual objects (msg, broadcast, getpubkey, or pubkey messages) as ackData.

Else if the decryption is not successful:

Sleep for a calculated amount of time (0.3 to several seconds depending on the size of the message and the speed of your computer). Thus decrypting or failing to decrypt the message will take the same amount of time.

Algorithm to calculate the proof-of work:

```
nonce = 0
averageProofOfWorkNonceTrialsPerByte = 320
extraBytes = 14000
trialValue = 99999999999999999999
target = $2^{64}$ / ((length of the payload + extraBytes ) * averageProofOfWorkNonceTrialsPerByte)
initialHash = SHA512 (payload)
while trialValue > target:
    nonce += 1
    trialValue  = first 8 bytes of SHA512( SHA512( nonce byte string + initialHash) )
        interpreted as int
prepend the nonce byte string to the payload
```

6. Encryption Source Code

To implement ECIES and ECDH, we rely on a Python wrapper for OpenSSL.
The highest level functions are here:
https://github.com/Atheros1/pyelliptic/blob/master/pyelliptic/bitcoin.py

This requires several other files:

openssl.py
https://github.com/Atheros1/pyelliptic/blob/master/pyelliptic/openssl.py

ecc.py
https://github.com/Atheros1/pyelliptic/blob/master/pyelliptic/ecc.py

cipher.py
https://github.com/Atheros1/pyelliptic/blob/master/pyelliptic/cipher.py

hash.py
https://github.com/Atheros1/pyelliptic/blob/master/pyelliptic/hash.py

arithmetic.py
https://github.com/Atheros1/pyelliptic/blob/master/pyelliptic/arithmetic.py

7. Attackers

There are several attackers against whom we hope to defend:

- Chuck is malicious but has no abilities above normal Internet nodes.
- NSA may eavesdrop on Internet backbones but not individual Internet connections.
- Eve may eavesdrop on a particular Internet connection but not all individual Internet connections.
- Mallory is malicious, has the abilities of Eve, and may also modify and drop packets at-will.

Alice sends a message to Bob. Both are honest.

Concerning the first five goals of the project (decentralized, trustless, encrypted, authenticated, and simple exchange of identity information), we contend that they all can be accomplished using the methods above although when Mallory is the attacker, he can prevent individuals from communicating by blocking all messages to and from the target node. Exchanging hashes of public keys guarantees that users receive the correct public keys, and at that point the security of the system should be no different than with other public key systems.

The last goal, hiding the sender and receiver of individual messages, is much more difficult.

8. Attacks

Eavesdropping attack

We contend that NSA, under the definition above, cannot identify the sender or receiver of a message but can identify the rough geographic location of Alice and Bob with a reasonable probability. To find the locations or, perhaps, the real life identities of Alice or Bob, he would need to monitor all individual Internet connections in order to detect which node first responds to an acknowledgement. None of our attackers have this ability although Eve may perform the attack on individual nodes that she suspects may harbor Alice or Bob. This attack is viable.

We finally contend that Mallory is no more an able attacker than Eve except that he may block messages. Someday, when the bitfield_features field is used, that field might need to be signed within the pubkey message if it can be imagined that flipping bits in the field would be of any particular benefit or nuisance to anyone.

Proposed solution

The eavesdropping attack can be thwarted if Bob, being a paranoid individual, instead of sending an acknowledgement out through his own Internet connection, waits a random number of minutes and then packages it up within another message and sends it to yet another user, Charlie, a public figure whom he trusts not to be colluding with Eve. The message Bob sends to Charlie may even be a normal important message (rather than a blank message). When Charlie broadcasts the acknowledgement, it would simultaneously acknowledge the message both from Alice to Bob and from Bob to Charlie. If Charlie and Bob do not personally know each other, Bob may send a blank message which is not shown in Charlie's user interface. Bob may also distribute his public keys or send msg or broadcast messages in this manner. Charlie is happy to provide this service because the marginal cost of sending an acknowledgement is small and because it gives Charlie plausible deniability for the acknowledgement messages which truly *are* his which emanate from his Internet connection. This proposed solution is an attempt to accomplish one goal (hiding message meta-data from Eve and Mallory) but comes at the cost of another (trustlessness).

Other suggestions are welcome.

Timing attack
Naively implemented, a Bitmessage client would be vulnerable to a timing attack where Chuck sends a series of messages to node1 and node2 in an attempt to locate Alice. Chuck sends a hundred msg messages bound for Alice to node1. Let us suppose that it takes a typical node one second to process an msg message if the message is *not* bound for any address owned by this node but two seconds if it *is* as the node must decrypt and display it. Node1 would reach and request the last msg message after 100 seconds if Alice is not at the node or 200 seconds if she is. This reveals Alice's location.
Proposed Solution
We propose that nodes measure the length of time it takes to successfully decrypt and display messages of various sizes and also the time it takes to unsuccessfully decrypt messages of various sizes. The difference is the amount of time a node should sleep if it is unable to decrypt a message. Nodes can come pre-programmed with reasonable default values and can adjust based on their own timings.

9. Extensions

Instant Messaging
Using the protocol above, an IM interface would not be functional because the proof-of-work would take several minutes to complete for each message. We propose an extension to the above protocol where a special operating mode is used for IM and for sending large attachments. The idea is that nodes agree to directly connect to one-another or to a third party if they are blocked by firewalls. They then forgo the POW requirement and are able to send instant messages or files of any size. Using this option, the last goal of the project is abandoned: both the third party and NSA would be able to tell which two nodes are communicating. Users would, however, be free to use some Bitmessage addresses privately- never using the IM operating mode- and others publicly.
Other ideas on how to implement instant messaging are welcome.

10. Conclusion

We have presented our plan for how a Bitmessage protocol might work. We invite researchers to review our chosen cryptographic implementation and review our rough specification looking for potential problems. We are confident that users worldwide would benefit from a protocol like this and we aim to acquire input and make changes early in the design process to help the project be as successful as it can be.