Blockchain Instant Messenger Specification *v1* Artefact

Jasper Parish

February 2020

This represents *v1* of the final result (artefact) of my EPQ design process.

Contents

1 Blockchain Architecture

The following table shows all the data types which are to be used for the platform.

Data Type	Description
uint_8	8 bit unsigned integer.
uint_16	16 bit unsigned integer.
uint_32	32 bit unsigned integer.
uint_64	64 bit unsigned integer.
uint_128	128 bit unsigned integer.
(u)int_256	256 bit (un)signed integer.
var_(u)int	Variably sized, (un)signed integer.
(u)int_xx[]	Array of xx-bit integers. (prepended with length)
tx	Message transaction.
encrypted_payload	Encrypted field in a tx.
msg	Message object contained by encrypted_payload.
block	Block. (with headers)
blockheaders	Block headers.

1.1 tx

The following table shows the structure of a **tx** (message transaction).

Data Type	Field	Description
uint_32	version	Version of platform used by sender.
uint_256	chat_id	ChatID of chat.
uint_256	tx_id	PoW value for tx. $H(H(nonce + encrypted\_payload))$
uint_32	lock_time	The block number at which this transaction is unlocked.
uint_64	nonce	PoW nonce value.
uint_128	iv	Initialisation vector used for AES-256-CBC.
uint_256	x	x component of ephemeral public key R. (signed)
uint_256	y	y component of ephemeral public key R. (signed)
uint_256	mac	Message authentication code.
uint_8[]	encrypted_payload	Encrypted payload.

Where:

- *chat_id* (ChatID) is a cryptographically secure 256-bit random number. This must be exchanged between users before they can communicate properly.
- *lock_time* is the first block in which this transaction can be included.
- *nonce* is a random value incremented until the tx_id is below a target value.

Encrypting and Decrypting a tx The steps to **encrypt** are as follows (identical to Bitmessage):

1. The destination public key is called K.
2. Generate 16 random bytes using a cryptographically secure random number generator. Call them *iv*.
3. Generate a new random EC key pair with private key called r and public key called R.
4. Do an EC point multiply with public key K and private key r. This gives you public key P.
5. Use the x component of public key P and calculate the **SHA512** hash H.
6. The first 32 bytes of H are called *key_e* and the last 32 bytes are called *key_m*.
7. Pad the input text[1] to a multiple of 16 bytes, in accordance to PKCS7.
8. Encrypt the data with **AES-256-CBC**, using *iv* as initialization vector, *key_e* as encryption key and the padded input text as payload. Call the output *cipher_text*.
9. Calculate a 32 byte MAC with **HMACSHA256**, using *key_m* as salt and *iv* + R + *cipher_text* as data. Call the output *mac*.

Thus the steps to **decrypt** are as follows (again identical to Bitmessage):

1. The private key used to decrypt is called k.
2. Do an EC point multiply with private key k and public key R. This gives you public key P.
3. Use the x component of public key P and calculate the **SHA512** hash H.
4. The first 32 bytes of H are called *key_e* and the last 32 bytes are called *key_m*.
5. Calculate MAC' with **HMACSHA256**, using *key_m* as salt and *iv* + R + *cipher_text* as data.

[1]This just refers to the input payload (ie. the fields that will later become *encrypted_payload*) in bytes.

6. Compare MAC with MAC'. If not equal, decryption will fail.

7. Decrypt the cipher text with **AES-256-CBC**, using *iv* as initialization vector, *key_e* as decryption key and the *cipher_text* as payload. The output is the padded input text.

Here, the *cipher_text* is the *encrypted_payload*.

Proof-of-Work The target value for a transaction's proof-of-work is calculated as follows.

$$target = \frac{2^{64}}{payloadLength + payloadExtraBytes}$$

payloadExtraBytes is set to 1000 as a network minimum. Its purpose is to increase the difficulty of small messages. *payloadLength* refers to the length in bytes of *encrypted_payload*.

The use of proof of work on each transaction increases the computational cost for a node to spam the network.

1.2 encrypted_payload

The following table shows the fields present on an **encrypted_payload**.

Data Type	Field	Description
var_uint	msg_id	msg_id of message. (Used for acknowledgements and reactions)
uint_256	x_tx_from	x component of sender's public key (0 for anonymous).
uint_256	y_tx_from	x component of sender's public key (0 for anonymous).
uint_256	x_tx_to	x component of intended recipient's public key (0 for open)
uint_256	y_tx_to	y component of intended recipient's public key (0 for open)
uint_8	msg_type	Type of message. (text, acknowledgment, sticker...)
msg	msg	Message object. (has different subfields depending on msg_type)

1.3 msg

A **msg** object can be of several different types.

text

To send a simple message (text only), the following message is sent.

Data Type	Field	Description
uint_16	encoding	Encoding standard for decoding bytes to text. (eg. `UTF-8`)
uint_8[]	data	Simple plain-text message.

acknowledgement

An *acknowledgement* msg is sent in order to indicate that a message has been read.

Data Type	Field	Description
var_uint	rmsg_id	The msg_id of the message which is being acknowledged.

sticker

A *sticker* message is a way of sending a picture. It is equivalent to the stickers included in all major, modern messaging services today.

Data Type	Field	Description
uint_16	sticker_pack	The sticker pack which contains the chosen sticker.
uint_8	sticker_index	The index of the chosen sticker in its sticker pack.

reaction

The *reaction* message is sent to indicate a reply to a message. If the reply is a single unicode character long (eg. an emoji), then a client should treat this as a reaction (as in Facebook Messenger).

Data Type	Field	Description
var_uint	rmsg_id	The message to be responded to.
uint_16	encoding	Encoding standard for decoding bytes to text.
uint_8[]	data	Plain-text message data (could be single Unicode character or string reply).

file

The *file* message is self-explanatory. It allows for one file to be sent along with file name and file type.

Data Type	Field	Description
uint_8[]	file_name	Name of file.
uint_8[]	file_type	Type of file (eg. PDF, PNG, JPEG).
uint_8[]	data	Binary file data.

1.4 block

The following table indicates the fields which should be present on a block.

Data Type	Field	Description
uint_32	version	Version of platform used by block creator.
uint_32	no	Block Number.
uint_256	block_id	The Merkle root hash of the current block. (used for PoW)
uint_256	prev	The BlockID of the previous block in the blockchain.
uint_64	nonce	PoW nonce value for the block.
uint_256	author_x	Author's public key x component.
uint_256	author_y	Author's public key y component.
uint_256	author_sig	Block signature.
uint_32	ts	Timestamp of block when created.
tx[]	tx_l	A list of transactions.

Where:

- *no* (block number) is the index of a block in the blockchain (ie. 1^{st} block $\rightarrow$ 0, 2^{nd} block $\rightarrow$ 1, 3^{rd} block $\rightarrow$ 2...);
- *block_id* is calculated with $H(nonce + prev + merkle)$ ($H \coloneqq$ `SHA256` hash function, $+ \coloneqq$ concatenation);
- *nonce* is a random integer that is incremented until the BlockID is below a certain difficulty value;
- *author_x*, *author_y* are the components of the block creator's public key;
- *author_sig* is the BlockID signed with the block author's private key;
- *ts* is the UNIX timestamp of the block's creation time.

Proof-of-Work The difficulty for the next block in the chain is calculated as so.

$$difficulty_{new} = difficulty_{old} + \left\lfloor \frac{difficulty_{old}}{2048} \right\rfloor \times \max\left(1 - \left\lfloor \frac{block\_time}{10} \right\rfloor,\ -99\right)$$

The new target value is calculated as follows.

$$target_{new} = \frac{target_{old}}{difficulty_{new}}$$

2 Network

This platform uses a decentralised peer-to-peer network of nodes who relay information between each other. All communication is done over **TCP** (Transmission Control Protocol).

The default port for use of this platform's network protocol is **8855**.

The 'ip_addr' type has to be defined for the network protocol. This consists of the following structure.

Data Type	Field	Description
uint_128	address	IPv6 address or IPv4-mapped IPv6 address.
uint_16	port	Port number.

2.1 request

The standard structure for a network request is as follows.

Data Type	Field	Description
uint_32	magic	Magic value indicating origin network.
uint_8	command	Identifies packet content (ie. encodes the purpose of the message).
uint_32	checksum	Checksum hash of request payload.
request_payload	payload	Actual request payload (ie. content of request).

Where:

- *magic* is the magic value of the network being used ($\texttt{DDE6AE4C}_{16}$ and $\texttt{CF713BF3}_{16}$ for main and test networks respectively);
- *command* indicates the type of message to be sent (ie. 0 = *text*, 1 = *acknowledgement*, ...);
- *checksum* is the first four bytes of $H(request\_payload)$ ($H \coloneqq$ `SHA256` hash function);
- *request_payload* is the main request itself.

2.2 request_payload

A **request_payload** can be any of the following types.

version

The **version** request is how all communications on the network are initiated. Each node sends a 'version' request to the other, who returns with a 'verack'.

Data Type	Field	Description
uint_8	version	Version of platform being used.
uint_64	nonce	Random nonce used to detect connections to self.

verack

The **verack** request is made in response to a 'version' request. It contains no fields and thus the payload is empty.

addr

The **addr** request is made when a node wants to discover new nodes on the network. This request should be made in most interactions between two nodes in order to keep the network online.

Data Type	Field	Description
ip_addr[]	addr_l	Array of shared node addresses. (max 1000)

inv

The **inv** request is sent once 'verack's have been exchanged. This is so that each node knows the exact data available to them.

Data Type	Field	Description
uint_32[]	blocks	List of block numbers of blocks to share.
uint_32[]	blockheaders	List of block numbers of block headers to share.
uint_8[]	bloomfilters	List of block numbers for which a bloom filter is possessed.

get_blocks

The **get_blocks** request is sent in order to ask for a given set of blocks.

Data Type	Field	Description
uint_32[]	block	List of block numbers of blocks wanted.

get_blockheaders

The **get_blockheaders** request is sent in order to ask for a given set of blockheaders.

Data Type	Field	Description
uint_32[]	blockheaders	List of block numbers of block headers wanted.

blocks

The **blocks** request is sent in order to return a set of blocks asked for by another node. If the sender doesn't possess one or more the blocks asked for, then those blocks are ignored. An empty 'blocks' request indicates that none of the requested blocks were possessed.

Data Type	Field	Description
block[]	blocks	List of blocks to share.

blockheaders

The **blockheaders** request is sent in order to return a set of blockheaders asked for by another node. If the sender doesn't possess one or more of the block headers asked for, then those block headers are ignored. An empty 'blockheaders' request indicated that none of the requested blocks were possessed as in the 'blocks' request above.

Data Type	Field	Description
blockheader[]	blockheaders	List of block headers to share.

get_newtx

The **get_newtx** request is sent in order to ask for all transactions that have yet to be included in the blockchain. It contains no fields; thus the payload is empty. Non-mining nodes will never need to send this request but they may receive a 'get_newtx' request.

newtx

The **newtx** request is sent in response to a 'get_newtx' request. It contains one field: a list of transactions.

Data Type	Field	Description
tx[]	tx_l	List of new transactions to share. (max 1000)

check_blooms

The **check_blooms** request is sent in order to check to see if messages on given chats have been included in any of a given set of blocks. It contains two fields: a list of ChatIDs to filter for and a list of block numbers corresponding to blocks which should be checked.

Data Type	Field	Description
uint_256[]	chat_ids	ChatIDs for transactions being looked for.
uint_32[]	blocks	Block Numbers to check.

blooms_checked

The **blooms_checked** request is sent in response to a 'check_blooms' request and provides the results of a bloom filter check.

Data Type	Field	Description
uint_32[]	blocks	Block Numbers which gave positive result.