Table of Contents

Overview

Key Components

Smart Contract

This is an application on a public blockchain platform that manages the recording of messages by requesters and responders, allowing for communication that can later be proven and verified publicly.

For our implementation, we are deploying our smart contract on an Ethereum Virtual Machine(EVM)-based blockchain network.

File System

This is a platform where files are stored. To aid the public verification feature, the file system must allow for files to be publicly accessed.

For our implementation, we are storing our files on the Inter-Planetary File System (IPFS), along with the Pinata Pinning Service to ensure the availbility of files.

Requesters

These are users that submit requests to the Messaging smart contract and wait for a response.

For our implementation, a requester is an instance of the MessageUser class, written in Python, that encapsulates methods to send requests and receive responses.

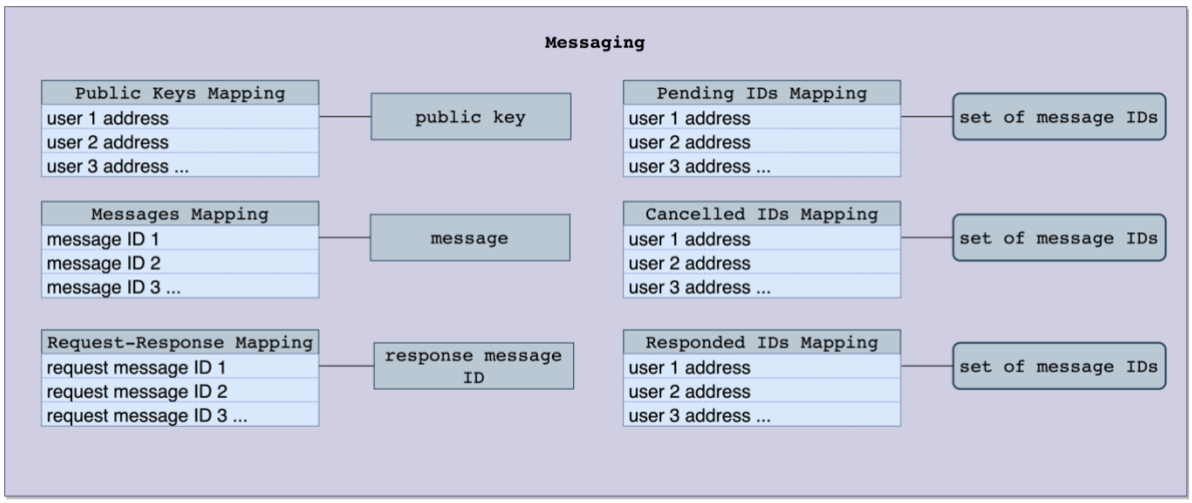
Responders

These are users that submit responses on the Messaging smart contract and wait for a response.

For our implementation, a responder is an instance of the MessageUser class, written in Python, that encapsulates methods to receive requests and send responses.

Smart Contract Usage

Data Layout



Public Keys Mapping
Maps a wallet address to its associated public key. This public key is required for sending messages to the associated wallet address.

Messages Mapping
This is where messages, which can either be a request or a response, are stored. Each message has a unique ID, which is an incremental counter. Message IDs begin from 1.
0 is reserved.

Request-Response Mapping
Maps the message ID of a request to the message ID of its response. A response ID of 0 indicates that the request has not been responded to yet.

Pending IDs Mapping
Maps a responder wallet address to a set of request IDs awaiting responses from it.

Cancelled IDs Mapping
Maps a responder wallet address to a set of request IDs that it has failed to construct a proper response for. Response construction may fail for a variety of reasons, including when the responder is not able to decrypt the original request and as well as when the signature verification of the original request fails.

Responded IDs Mapping
Maps a responder wallet address to a set of request IDs that it has successfully responded to.

Setting Up

The smart contract assigns overall admin rights to the deployer at the point of creation. This deployer can then assign requester rights to selected wallet addresses. In other words, requests are restricted to a selected set of users. This is to prevent spam from potentially malicious actors.

We also need the public key from both requesters and responders to be made available. Users participating in this protocol must send a transaction to upload their public key to the smart contract.

The smart contract will enforce that the public key uploaded is linked only to the sender of the transaction. In other words, malicious actors cannot manipulate the public key of other users. If user A uploads a public key, this public key will be be linked to user A. If user B uploads a public key, this will be linked to user B and cannot be linked to user A. The `setPublicKey` method takes only 1 input – the public key in bytes format, and then stores and links it to the sender of the transaction.

Note that the transaction itself is also cryptographically secured by the blockchain protocol – the "sender" of a transaction must sign the transaction using the wallet private key associated with the specified "sender". So, user B cannot send a transaction and manipulate the "sender" of the transaction to be user A. (without having somehow stolen user A's wallet private key via social engineering, phishing, or other means. This is an issue for all existing cryptographic systems and is beyond the scope of our protocol)

Message ID

The message ID is an incrementing counter beginning from 1.
Every message, whether request or response, is given a unique message ID that is 1 more than the last recorded message.
Every message is either a request or a response.

Message

Each message contains 8 fields.

Field Name	Description
mid	The message ID of this message.
sender	Wallet address of the message sender.
receiver	Wallet address of the intended receiver of the message.
fileReference	Identifier from which the encrypted message file can be retrieved from the file system.
encryptedSymmetricKey	Symmetric key encrypted with the receiver's public key. This symmetric key is the one used for encrypting the message file, which can also be used to decrypt the message file.

signedHash	Hash of the unecnrypted message file signed using the sender's wallet private key.
pid	The message ID of the request message being responded to. "0" if this message is a request.
decryptedSymmetricKey	Unencrypted form of the symmetric key in the "encryptedSymmetricKey" field. This is meant to be empty when the message is first created. Later, the "sender" may record the decrypted symmetric key here so that anyone who wants to check and verify the contents of the encrypted file (accessed via the file reference) may do so.

Making a Request

Requests are made to the smart contract via the `request` method.
The `request` method takes 4 inputs:

1. The wallet addresses of the intended responder.
2. The file reference of the encrypted request file. This is an ID to be used for retrieving the encrypted request file from the file system.
3. The encrypted symmetric key. This is the symmetric key used to encrypt the request file, which is then encrypted with the public key of the intended responder.
4. The signed hash of the request file. This is the hash of the unencrypted request file, signed with the wallet private key of the requester.

The `request` method creates a message as follows:

Field Name	Value
mid	1 more than the last recorded message ID.
sender	Wallet address of the requester. The smart contract enforces that this is the wallet address of the sender of this `request` transaction.
receiver	Wallet address of the intended responder.
fileReference	Identifier from which the encrypted request file can be retrieved from the file system.
encryptedSymmetricKey	Symmetric key encrypted with the responder's public key. This symmetric key is the one used for encrypting the request file, which can also be used to decrypt the request file.
signedHash	Hash of the unencrypted request file signed using the requester's wallet private key.
pid	0 (This indicates that this message is a request and not a response.)

decryptedSymmetricKey	Empty Value

The smart contract enforces that the sender and receiver fields cannot be the same.

The message ID of this message is added to the set of pending IDs for the receiver of this message.

Making a Response

Responses are made to the smart contract via the `response` method.
The `response` method takes 5 inputs:

1. The wallet addresses of the intended recipient. This should be the "sender" of the request message that this response is meant for. (ie. the requester)
2. The file reference of the encrypted response file. This is an ID to be used for retrieving the encrypted response file from the file system.
3. The encrypted symmetric key. This is the symmetric key used to encrypt the response file, which is then encrypted with the public key of the requester.
4. The signed hash of the response file. This is the hash of the unencrypted response file, signed with the wallet private key of the responder.
5. The message ID of the request message that this response is meant for.

The `response` method creates a message as follows:

Field Name	Value
mid	1 more than the last recorded message ID.
sender	Wallet address of the responder. The smart contract enforces that this is the wallet address of the sender of this `response` transaction.
receiver	Wallet address of the requester.
fileReference	Identifier from which the encrypted response file can be retrieved from the file system.
encryptedSymmetricKey	Symmetric key encrypted with the requester's public key. This symmetric key is the one used for encrypting the response file, which can also be used to decrypt the response file.
signedHash	Hash of the unencrypted response file signed using the responder's wallet private key.
pid	Message ID of request message that this response is meant for.
decryptedSymmetricKey	Empty Value

The smart contract enforces that:

- The sender and receiver fields cannot be the same.
- The sender must be the receiver of the message referenced by the "pid".

    - ie. The responder must be the intended responder specified in the original request message.
- The receiver must be the sender of the message referenced by the “pid”.
    - ie. The receiver of this response must be the requester of the original request message.
- The “pid” must exist in the sender’s set of pending IDs.

The request-response mapping is updated to map the message ID of the request message to the message ID of this response message.

The “pid” (original request ID) is removed from the sender’s (responder) set of pending IDs.
If the file reference is of a 0 value, this indicates that there was a problem with constructing a response, and the “pid” is added to the sender’s set of cancelled IDs.
Otherwise, the “pid” is added to the sender’s set of successfully responded IDs.

Publishing a Symmetric Key

At some later time, the symmetric key that was used to encrypt a particular message, can be recorded in the smart contract, so that anyone one who wants to check or perform some verification on the contents of the message can do so.
This can be done via the `updateDecryptedSymmetricKey` method.
The message ID and the unencrypted symmetric key must be provided as input.
The smart contract will enforce that the sender of the transaction must be the “sender” of the message referenced by the message ID. It will then proceed to record this symmetric key under the “decryptedSymmetricKey” field of the message referenced by the message ID.

IPFS Usage

IPFS is a decentralised file storage system where users that retrieve a file can also serve it for other future retrievers. To ensure that our files are continuously served and available, we need to serve it via a continously running and connected machine.
For our use case we use a service (Pinata) that takes care of the details of serving a file to the IPFS network once we upload it. This act of uploading a file and to be continuously served is known as “pinning”.

Python Script Usage

Message User

Each `MessageUser` instance consists of an instance of the `Sender` class as well as an instance of the `Receiver` class. This encapsulates what each user needs to do to interact with the protocol. A requester will need to send requests and receive responses, while a responder will need to receive requests and send responses.

Sender

The `Sender` class consists of the following methods.

Get Public Key

The `get_public_key` method retrieves the public key of the specified recipient address from the Messaging smart contract. This is used for encryption, so that the recipient, and only the recipient, can read the message being sent to it.

Send Request

The `send_request` method sends a blockchain transaction that calls the `request` method of the Messaging smart contract. This method requires a file reference, encrypted symmetric key and signature that can be obtained via the `prepare_message` method.
There is an internal `unresponded_requests` list, which records the message IDs of the requests sent, that helps to keep track of which requests this user is still waiting on a response for.

Send Response

The `send_response` method sends a blockchain transaction that calls the `response` method of the Messaging smart contract. This method is similar to the `send_request` method, except that it also requires a "pid", which is the message ID of the request this response is meant for.

Prepare Message

The `prepare_message` method is the most elaborate of the methods in the `Sender` class and takes the necessary actions to prepare a message to be sent to the smart contract.

It first generates a symmetric key. This key will be used to both encrypt and decrypt the message file.
In our implementation this a random byte string of length 16 generated by the cryptographically secure random number generator offered by the PyCryptodome library.

Next, it encrypts the message file using the symmetric key.
In our implementation we use the Advanced Encryption Standard (AES)[1] in Galois/Counter Mode (GCM)[2] offered by the PyCryptodome library.

We then pin this encrypted file to IPFS via the Pinata service, obtaining a file reference. This file reference, a base58-encoded string, can be used to download the encrypted file from the public IPFS network.

The symmetric key is then encrypted using the receiver's public key.
In our implementation we use the RSAES-OAEP[3] cipher offered by the PyCryptodome library.

We then obtain a hash from the unencrypted message file.

---

[1] Dworkin, Barker, Nechvatal, Foti, Bassham, Roback and Dray Jr, "Advanced Encryption Standard (AES)", NIST, November 2001, https://www.nist.gov/publications/advanced-encryption-standard-aes.
[2] Dworkin, "Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC", NIST, November 2007, https://csrc.nist.gov/pubs/sp/800/38/d/final.
[3] Moriarty, Kaliski, Jonsson and Rusch, "RFC 8017 - PKCS #1: RSA Cryptography Specifications Version 2.2", IETF, November 2016, https://datatracker.ietf.org/doc/html/rfc8017.

In our implementation we perform a SHA-1[4] hash on the unencrypted message file, offered by the Hashlib library.

We then take this hash and sign it using the wallet private key of the sender, producing a signature in the EIP-191[5] format. Signing is done using the ECDSA[6] algorithm.

We return the file reference, encrypted symmetric key, signature for use by either `send_request` or `send_response`. We also return the unencrypted symmetric key to be saved and used by the `publish_symmetric_key` method.

Publish Symmetric Key

The `publish_symmetric_key` method sends a blockchain transaction that calls the `updateDecryptedSymmetricKey` method of the Messaging smart contract. This records the provided symmetric key to the message in the smart contract referenced by the specified message ID.

Receiver

Load Private Key

The `load_private_key` method is used to read the private key of the receiver, stored on its local machine.
Public-private key pairs are generated in our implementation using the RSA algorithm[7], offered by the PyCryptodome library, using 2048 bits for the RSA modulus.

Set Own Public Key

The `set_own_public_key` method sends a blockchain transaction to publish the specified public key on the Messaging smart contract and link it with the receiver.

Get Messages

The `get_messages` method reads from the Messaging smart contract to get the set of pending, cancelled, or responded messages for the receiver.

Get Response Message

The `get_response_message` method reads from the Messaging smart contract to get the response message to the specified request message ID. A “0” indicates that the request has not been responded to yet.

---

[4] NIST, “FIPS 180-4: Secure Hash Standard (SHS)”, August 2015, NIST, https://csrc.nist.gov/pubs/fips/180-4/upd1/final.
[5] Swende and Johnson, “ERC-191: Signed Data Standard”, Ethereum Improvement Proposals, no. 191, January 2016, https://eips.ethereum.org/EIPS/eip-191.
[6] Johnson, Menezes and Vanstone, The Elliptic Curve Digital Signature Algorithm (ECDSA), Certicom Research, 2001, https://web.archive.org/web/20170921160141/http://cs.ucsb.edu/~koc/ccs130h/notes/ecdsa-cert.pdf.
[7] Rivest, Shamir, Adleman, “A Method for Obtaining Digital Signatures and Public-Key Cryptosystems”, Communications of the ACM, Vol 21 #2, February 1978, https://dl.acm.org/doi/10.1145/359340.359342.

Retrieve Message

The `retrieve_message` method is the most elaborate of the methods in the `Receiver` class and takes the necessary actions to retrieve the contents sent to the receiver.

Given a message, that has been read via the `get_message` method, the file reference is used to download the encrypted message file from IPFS.

The encrypted symmetric key is then decrypted using the receiver's private key.
In our implementation we use the RSAES-OAEP cipher offered by the PyCryptodome library.

We now have the symmetric key, which we use to decrypt the encrypted message file.
In our implementation we use AES in GCM.

We now have the unencrypted message file, on which we perform a hash.
In our implementation we use the SHA-1 hash.

We then verify this hash against the signature that was sent along in the message, by checking that the wallet address recovered by the hash (using ECDSA) and the signature is indeed the wallet address of the sender.

Delay Analysis

Cryptography

Tests performed on an Apple M1 Machine with 16 GB Ram.

	Median Delay (ns)	**Mean Delay (ns)**
Generate symmetric key	1,000	1,096
Encrypt csv file (39MB)	282,471,500	282,858,600
Encrypt csv file (390MB)	3,074,220,500	3,229,708,130
Encrypt symmetric key	605,000	609,638
Hash csv file (39MB)	25,035,000	27,803,200
Hash csv file (390MB)	193,684,500	215,080,800
Sign Hash	4,762,000	4,850,152
Decrypt symmetric key	3,590,000	3,609,043
Decrypt csv file (39MB)	288,447,000	287,356,400
Decrypt csv file (390MB)	3,316,888,500	3,333,414,100
Verify Hash	7,379,000	7,521,713

Network

Network times should be treated as a rough guide as these can vary with factors such as the user machine's network setup, location, internet connection speed as well as overall network conditions.

	Median Delay (ns)	Mean Delay (ns)
Pin File to IPFS (39MB)	13,469,971,500	13,928,348,500
Pin File to IPFS (390MB)	66,980,130,000	79,855,493,666
Retrieve File from IPFS (39MB)	5,742,185,500	7,064,324,000
Retrieve File from IPFS (390MB)	25,347,311,000	65,699,283,666
Read one message from blockchain	722,070,000	702,579,000
Send one message to the blockchain	12,000,000,000*	12,000,000,000*

**: This is based on the minimum network propagation delay of 12 seconds for a blockchain network to be fully decentralised[8]. Certain blockchain networks may offer delays with a centralisation tradeoff. These blockchain networks may be utilised if their level of centralisation is within acceptable bounds.*

Communication Overhead

Suppose we are making a request with a file of around 390MB and expecting an output file of forecasts of around 39MB.

Our communication overhead will be as follows.

Send Request (83 seconds)

Action	Median Delay (ns)
Generate symmetric key	1,000
Encrypt csv file (390MB)	3,074,220,500
Encrypt symmetric key	605,000
Hash csv file (390MB)	193,684,500
Pin File to IPFS (390MB)	66,980,130,000
Sign Hash	4,762,000
Send one message to the blockchain	12,000,000,000
TOTAL	82,253,403,000

Receive Request (30 seconds)

Action	Median Delay (ns)
Read one message from blockchain	722,070,000
Decrypt symmetric key	3,590,000
Retrieve File from IPFS (390MB)	25,347,311,000

[8] Buterin, "Toward a 12-second Block Time", Ethereum Foundation, July 2014, https://blog.ethereum.org/2014/07/11/toward-a-12-second-block-time.

Decrypt csv file (390MB)	3,316,888,500
Hash csv file (390MB)	193,684,500
Verify Hash	7,379,000
TOTAL	29,590,923,000

Send Response (26 seconds)

Action	**Median Delay (ns)**
Generate symmetric key	1,000
Encrypt csv file (39MB)	282,471,500
Encrypt symmetric key	605,000
Hash csv file (39MB)	25,035,000
Pin File to IPFS (39MB)	13,469,971,500
Sign Hash	4,762,000
Send one message to the blockchain	12,000,000,000
TOTAL	25,782,846,000

Receive Response (7 seconds)

Action	**Median Delay (ns)**
Read one message from blockchain	722,070,000
Decrypt symmetric key	3,590,000
Retrieve File from IPFS (39MB)	5,742,185,500
Decrypt csv file (39MB)	288,447,000
Hash csv file (39MB)	25,035,000
Verify Hash	7,379,000
TOTAL	6,788,706,500

This gives us a total round trip overhead, from sending a request to receiving a response, of 146 seconds.

Future Works

Public-Private Key Rotation

It is good practice to change cryptographic keys regularly. The smart contract can be re-organised such that it can accommodate the update of public keys associated with a wallet address, along with the ability for message receivers to identify which of their public keys has been used in the preparation of the message (and thus which of their private keys to use to retrieve the original message).

Compression

In this implementation, we have not considered compression. Compression may be applied at various steps to lighten the payload and decrease the delay overhead. For example, the

original message file might be compressed before encryption, or the encrypted file might be compressed before uploading it to the file system. More analysis is required to further understand the benefits of compression to this system.

More Efficient Cryptographic Libraries

Cryptographic libraries, possibly available only outside of the Python language, may be written in a way that takes bigger advantage of the available computing hardware and can complete the cryptographic computations in less time. These should be explored and applied if beneficial.