Mokachain: Blockchain As A Service API – Version 1.0.0 – 2018.08.30

**Blockchain As A Service**

J.D Neemuth, Madeven Pyanee

**Abstract**: A decentralized framework that would allow companies or users to share and verify sensitive information with each other without having to disclose non-essential information; and to ensure that only users having the right key can have access to the data published on the blockchain.

Company data is highly sensitive and is protected by the many data protection laws in countries. There has been widespread concern over data privacy issues lately on the internet. Besides; many systems are still using legacy systems to keep account of existing data. The proposed framework would be built in such a way as to facilitate integration with existing data warehouses and achieve an acceptable level of trust when sensitive data is being shared. This whitepaper discusses the problems that existing data transfers involve and how the framework resolves them.

**Keywords.** Data Protection, Encryption, Wired Transfer

**1. Introduction**

Under the original Bitcoin whitepaper, the transactions effected through the network

were used to act as a store for value to depict cryptocurrency. This principle was replicated

in all subsequent developments which resulted in the blockchain ecosystem transferring tokens

between nodes with no real value in itself.

Several frameworks have been created in the past to bridge the communication gap between

different computer systems to exchange information either through Webservices (SOAP/REST)

or XML. In that perspective; we have identified the blockchain as an efficient way to store confidential data which can be accessed by only authorized parties without going through a broker.

The sensitive data would be encrypted on the blockchain and can be accessed by different clients (web or mobile based).

Under the mokachain framework, the way the information is stored, shared and validated is completely decentralized, making it perfectly compatible with the nature of blockchain.

This whitepaper addresses the construction of the BAAS as follows:

1. classifying digital data

**2. Classifying Digital Data**

Digital data plays an essential part in the framework. It is upon the consumer to built up their sensitive data specific to their business and sends their requests to the mokachain network. The advantage of this approach is that their data would be available to a bigger audience (mobile app, web page or be consumed by other clients). The data would also be replicated on multiple nodes reducing the single point of failure.

The decentralization of data needs to be prioritized, starting with how we aim to protect that data from unauthorized access.

Within the mokachain framework, we will explain; how we construct a data exchange message, how we encrypt sensitive info for identification and storage on the blockchain; and how we make used of the consensus system to achieve transparency and always available data.

2.1 Creating a data exchange message

TEXT

XML

XMLdfdfdfdf

JSON

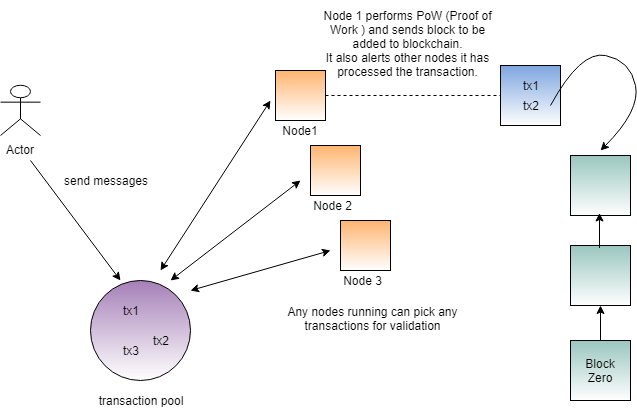
Fig. 1. Sources of Data

Digital data can be represented as different sources. We are going to concentrate only on the textual data in our infrastructure as the most commonly data exchange format used between systems.

To construct a digital data; we must provide textual data, xml or json; and hashed identifier that supports that the data belongs to that identifier. After mapping that identifier to the processed message, a decision is reached as to whether the owner of the encrypted data can be ascertained.

ARCHITECTURE

The system architecture is provided below:



1. User sends a message using the client; the message is wrapped into a transaction using encryption.
2. The transaction is sent to a transaction pool.
3. Any nodes ( call them miners ) in waiting can pick up the transactions in the pool; and start processing.

A complex mathematical proof must be solved so that the transaction can be added to a block.

1. If ever a node succeeds in assembling a whole block; it broadcasts to other nodes to validate the block and the black is finally added to the blockchain.
2. The other nodes stop any mining operations for that particular transaction.
3. The node which has performed the successful task is awarded with a number of tokens.

*mokachain does not contain any schemes like profit distribution as-is; but plans are underway to rewards miners either through tokens or by monetary means for their computing efforts.*

1. All the nodes are updated with the latest state of the blockchain.

First Steps…

Our Java-based blockchain was kept simply deliberately and so, from a users point of view, it only has the functions of sending messages and viewing them in the blockchain, quite similar to a public chatroom. You basically have to differentiate between users (who, for example, exchange money on the Bitcoin network) and network participants who provide infrastructure, store data, and also provide communication for between the participants. Different things have to be done, which depends upon what you are: either a user or a participant.

Since we also need the infrastructure to communicate with other users we have to start up a node. This command here shows how it works:

java -jar node/target/node-0.0.1-SNAPSHOT.jar

You will need a private key and a public key, as well as user name to communicate with the other participants. These are required. The key and the username will provide a unique public address later on, which can identify our messages. This command here creates the pair of keys:

java -jar client/target/client-0.0.1-SNAPSHOT.jar –keypair

There are two files key.priv and key.pub. Next up is the unique public address, which must be generated:

java -jar client/target/client-0.0.1-SNAPSHOT.jar --address –-node "[http://localhost:8080](http://localhost:8080/)" --name "Testing Joe" --publickey key.pub

This requires the address of a network node, the public key and a freely selectable name. The call returns the unique public address for the user. Check that the resource *http://localhost:8080/address* to see that they have been created in the system.

In order to send a message to the system these are required: The address, which was created in the previous step, and the private key. Listing 1 shows how the message “Hello World” is sent. Under the resource http://localhost:8080/transaction the message can now be viewed until the network node has written it into the actual blockchain. Fixed messages are then available at the resource under http://localhost:8080/block.

Listing 1: Send-Hello-World-Message

java -jar client/target/client-0.0.1-SNAPSHOT.jar --transaction --node "[http://localhost:8080](http://localhost:8080/)" --sender "Tdz0bKDfca3QjFAe5Ccuj9Noy6ah8n+R8DnZznvjic4=" --message "Hello World" --privatekey key.priv

ESTABLISHING NODE COMMUNICATION

The network nodes must communicate with each other so that everyone has the same state of the blockchain. The peer-to-peer approach has established itself to ensure that this works with a large number of participants. With this approach, all network nodes have the same status and communicate with each other without a central control authority. We use a simple communication via HTTP in our example instead of the peer-to-peer approach. As soon as a network node receives new information, such as a new transaction or a new block, it then sends the information to all other network nodes (broadcast all). For example: In listing 2 the *AddressController* implements a method with which a new address can be added, if it does not exist already. With the optional parameter *publish*, the node can be instructed to inform all other nodes about the new address.

Listing 2: The AddressController

@RestController()

@RequestMapping("address")

public class **AddressController** {

  private final AddressService addressService;

  private final NodeService nodeService;

  @RequestMapping(method = RequestMethod.PUT)

  void addAddress(@RequestBody Address address, @RequestParam(required = false) Boolean publish, HttpServletResponse response) {

    if (addressService.getByHash(address.getHash()) == null) {

      addressService.add(address);

      if (publish != null && publish) {

        nodeService.broadcastPut("address", address);

      }

      response.setStatus(HttpServletResponse.SC\_ACCEPTED);

    } else {

      response.setStatus(HttpServletResponse.SC\_NOT\_ACCEPTABLE);

    }

  }

}

The implementation of the *broadcastPut* method of NodeService just sends put-requests to all known network nodes in parallel. For the sake of simplicity we deliberately assume that the nodes are always accessible and also process the requests.

public void broadcastPut(String endpoint, Object data) {

  knownNodes.parallelStream().forEach(

    node -> restTemplate.put(node.getAddress() + "/" + endpoint, data));

  }

When a network node is started, the following initial actions are performed to update the local data to the current state of the network:

A request from the master network node to download the following data: all nodes that the master node knows, all client addresses, the blockchain (database data) and the complete transaction pool (temporary data that has not yet been written to the blockchain)

* Broadcast to all nodes that there is a new network node in the system
* The node will be ready then and can be addressed by users.

### SENDING AND VERIFYING MESSAGES

Previously we showed how to send a message with the help of the client. Under the hood, the message is packed in a transaction ( **listing 3**). The attribute *hash* forms the identifier of the transaction and is formed by hashing all attributes together. This makes it possible to uniquely identify a transaction: If the hash is the same, the content must also be the same. The message is stored in the field text and the *senderHash* references the unique sender address. The time of transaction creation is selected as the *timestamp*. The signature created in **listing 4** is stored in the *signature* attribute.

Listing 3: Package message in transmission

public class **Transaction** {

  private byte[] hash;

  private String text;

/\*\* meta data info \*/

private MetaDataVO metaDataContent;

/\*\* sealed content \*/

private SealedByteVO sealedContent;

  private byte[] senderHash;

  private long timestamp;

  private byte[] signature;

}

The *metadataContent* attribute contains all the relevant details to that transaction e.g. transaction hash, business or company.

The *sealedContent attribute* contains the content that has been encrypted with RSA & DES algorithm.

The signature is created from the message text or sealed content and the sender’s private key. Since the private key is only known to the sender, anyone can then confirm that the message was actually sent from the address behind the *senderhash*.

Listing 4: Signature creation

byte[] signature = SignatureUtils.sign(text.getBytes(), Files.readAllBytes(privateKey));

Transaction transaction = new Transaction(text, senderHash, signature);

The transaction is then sent to a network node.

Listing 5 shows how a transaction is accepted by the node and if it is sufficient for verification, how it enters the transaction pool. The pool is a buffer for transactions that are not yet anchored in the blockchain.

Listing 5: Node accepts transaction

public synchronized boolean add(Transaction transaction) {

  if (verify(transaction)) {

    transactionPool.add(transaction);

    return true;

  }

  return false;

}

The verify-method in **listing 6** first checks whether the sender of the transaction is known at all. At the address, the public key is accessible to everyone, so that the authenticity of the message can be confirmed together with the message text (getSignableData) and the signature. Finally, the system checks whether the transmitted hash of the transaction was calculated correctly.

Listing 6: verify method

private boolean ***verify***(Transaction transaction) {

  // known address

  Address sender = addressService.getByHash(transaction.getSenderHash());

  if (sender == null) {

    return false;

  }

  // correct signature

  if (!SignatureUtils.verify(transaction.getSignableData(), transaction.getSignature(), sender.getPublicKey())) {

    return false;

  }

  // correct hash

  if (!Arrays.equals(transaction.getHash(), transaction.calculateHash())) {

    return false;

  }

  return true;

}

### TRANSACTION POOL AND MINING

The messages would be stored in a relational database via an INSERT. Transaction properties (ACID) and transaction level (READ UNCOMMITTED or READ COMMITTED) ensure that parallel write and read accesses meet certain requirements and define how secure they should be. In this way, a transaction is opened before the INSERT and, if no errors occur, written into the database using a COMMIT.

Since the data is not stored centrally in the blockchain, but a copy of all data is stored on any number of network nodes and any numbers of users want to store data in parallel, we need a different mechanism to obtain transaction security. This is where the transaction pool and mining come in. There are basically two areas in the blockchain in which data resides. On the one hand, this is the transaction pool in which the data still to be written is located, and on the other hand the blockchain itself with the data no longer to be changed (**fig. 1**). To prevent the network nodes from writing transactions into the blockchain at the same time, we need to overcome a mathematical challenge. You take any freely selectable transactions from the transaction pool and generate a hash from them. This hash must now begin with one, two, or three zeros, depending on the level of difficulty.

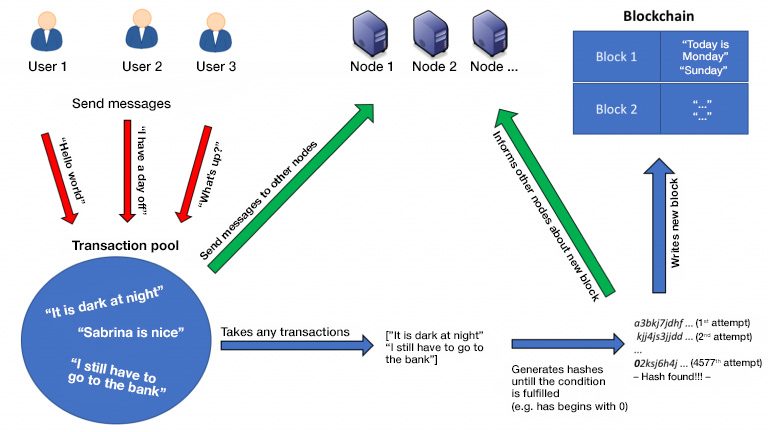


Fig. 1: The two areas of the blockchain where data resides: the transaction pool and the blockchain itself

The difficulty depends upon how much mining capacity is actually available. Simply put: Few network nodes mean that there is a small number of zeros, many network nodes mean that there is a large number of zeros. Here it is the goal that a suitable hash is only found after a certain amount of time, e.g. every five minutes. **Listing 7** shows how a suitable block is searched for by using a brute force method. A new block object is constructed, so long as the miner is active. The miner references the last block, which is already anchored in the blockchain and contains the previously selected transactions. In addition, each block has the attribute tries, which is a freely definable number and which is also used in the calculation of the block hash. If the hash of the newly created block does not have enough leading zeros, then tries is simply increased by one and a new block is created.

Listing 7: Using a brute force methodology to find a suitable block

long tries = 0;

while (runMiner.get()) { // atomicBoolean for controlling from other threads

  Block block = new Block(previousBlockHash, transactions, tries);

  if (block.getLeadingZerosCount() >= Config.DIFFICULTY) {

    return block;

  }

  tries++;

}

### CURRENT LIMITATIONS AND EXPANSION OPTIONS

The *[blockchain](https://github.com/neozo-software/jblockchain" \t "_blank) implementation* is kept very simple, because the understanding of the technology should be the main focus here. Many concepts are missing for a productive usage. Here are some examples:

* The all-broadcast does not scale for a large number of network nodes. Here we have need for a smarter message distribution.
* At least some master nodes must exist in the network (for example: statically stored) to which the users can connect.
* The naive consideration of a distributed system without concurrences, message loss etc. will quickly lead to problems.
* In general, it should be considered whether the peer-to-peer component should be replaced by a framework such as [GNUnet](https://gnunet.org/" \t "_blank).
* The mathematical challenge grows along with the number of network nodes. This is currently configured statically.
* The purpose of the application would have to be specified in more detail, because this could result in additional requirements. For example: the transactions may not be combined into a block, but certain transactions are of higher priority (e.g. according to date of receipt).

CONCLUSION

The blockchain is a highly interesting and exciting technology which will be with us for a long time, but there some factors to ponder upon before adopting the blockchain model.

A decision maker would also have a lot to put up with when it comes to the subject of concrete projects: Little control over the number of participants, high fluctuation of participants, complex programming model, difficult connection of external interfaces as well as incentives and marketing must be created for the system. These points make a realistic assessment of a project very difficult, as most companies expect reliable figures and a concrete result.

With our project, we are aiming to make the process of understanding and integrating into the block chain technology for developers much easier. We are still at our first iteration of our project; and this holds a lot of opportunities to rethink oh how applications are build.