Identity Management System Based on Blockchain Technology

Mohammad Shabib 19290116
Zaid Ibaisi 19290199
Mazen Houran 19290097

I. BACKGROUND

In this section, we are going to present the required blockchain key concepts and prerequisites the reader must have before proceeding to understand our solution.

1. **Blockchain technology**: blockchain can be thought of as a decentralized database in which every stakeholder in that database has a copy of it(more on stakeholders later).
2. **Blockchain as a database**: blockchain can be thought of as a linked list that is called a **ledger** in which there is a set of nodes or blocks connected to each other using a certain mechanism thus the name blockchain it’s a chain of blocks, where each block or node is connected to previous blocks using cryptography, in a way that tampering or altering a node will require to change all previous nodes.

3. **Stakeholders**: stakeholders of the blockchain are anyone who has an interest in the blockchain, let it be users, owners, or just observers of the transactions on the ledger.
4. **Transactions**: a transaction is any interaction between two nodes or blocks on the blockchain.
5. **Public blockchains**: public blockchains are blockchains in which anyone on any side of the world can join the network.
6. **Private blockchains**: private blockchains are blockchains that only a select number of users can join the network, for example, a private business, in which private blockchains control who is allowed to join the network.
7. **Permissioned blockchain**: permissioned blockchains are blockchains that require users to acquire authorization to join the network.
8. **Consensus**: Since there is no central authority that controls the blockchain and it’s a decentralized network, one could ask how do I know that I have the right copy of the blockchain? The answer to that question would be consensus mechanisms. So consensus mechanisms ensure that the ledger one could have is true and honest.
9. **Smart contracts**: Smart contracts are the equivalent of a traditional contract in the digital world of blockchain, where a smart contract is stored on the

blockchain as a computer program. But how does it work? let’s say that we want to deploy a blockchain for raising funds to donate to people in need, where we have a specific fund goal we want to reach, in a traditional system we would donate the money and trust the 3rd party we are donating to deliver the donations, but smart contracts solve the problem of having a 3rd party and trusting it. For example, it would hold all the received funds until the goal is reached, so the supporters can transfer their money to the smart contract, and then the smart contract would deliver the money to the required destination, this way we eliminate the need for the 3rd party and trusting it, and no one is in complete control of the funds, and since the smart contract is stored on a blockchain it automatically inherits the properties of it, such that, it is immutable and it is distributed, so everything is ensured to be done in the way it is intended to be.

II. CONSTRUCTION

In this section, we will show the architecture of the Identity Management platform.

● Blockchain Type

The Platform will be built on top of a **Public permissioned blockchain**, the blockchain will be based on two projects Hyperledger Indy & Sovrin Network, which provide tools, libraries, and reusable components for providing digital self-sovereign identities on blockchains. The reason for choosing a **public blockchain** is to enable anyone to read the blockchain without authorization, and choosing a **permissioned blockchain** since we don't want anyone without authorization to join the blockchain and write to the ledger.

● Platform Entities

The Identity Management System will consist of five entities: **identity owners, identity auditors, identity verifiers - Service Provider -, registrar, and a platform.**

The identity owner can store their credentials in their personal identity wallet and use them later to prove statements about

his/her identity to a third party (the verifier). The owner will register to the platform off-chain.

The identity Auditor, The auditor will check the PII provided by the user and will attest to the validity of The PII.
Auditors will have access to a government database that issued the documents, therefore they must be governmental employees, and they will make a transaction on the ledger indicating that this person's documents have been verified which in return will increase the person's trust score. The auditor can be a Governmental Certification Authority (CA), Auditors will be added by registrars off-chain.

The identity verifier (Service Provider), the verifier is a 3rd party user who tries to check the correctness of the identity owner's PII in a transaction. A verifier can be a business entity or organization. We can say every verifier is an owner.

The registrar will be responsible for the registration of both the owners, verifiers, and auditors, the Register can be Governmental Entity (civil registry department, Immigration department), and registrars will be added in the genesis block.

A Platform, The platform will act as an environment to monitor the activities of the user, verifier, and auditor.

- **Registration**

The registration process will be handled off-chain, and the entity responsible for the registration is none other than a government-issued entity that has access to databases to verify user identity in order to register to the system. Also, the auditor will be treated in the same way and the necessary hiring protocols will be applied so that only trustworthy auditors may audit the system.

- **Protocols & Standards**

The platform will use **Plenum**, we can say that it's a collection of standards and protocols used in the blockchain, it uses RBFT, as a consensus protocol, Zk-SNARK as a Zero-knowledge proof, Digital signature, Public-key, DPKI, and DIDs.

The **Redundant-BFT** is an enhanced version of practical-BFT, practical-BF has 4 phases as shown in Figure 1. (Propagate phase is added for Redundant-BFT).

- The client sends a request to the primary node.
- The primary node broadcasts the request to all the secondary(backup) nodes.
- The nodes(primary and secondaries) perform the service requested and then send back a reply to the client.
- The request is served successfully when the client receives 'f+1' replies from different nodes in the network with the

same result, where f is the maximum number of faulty nodes allowed.

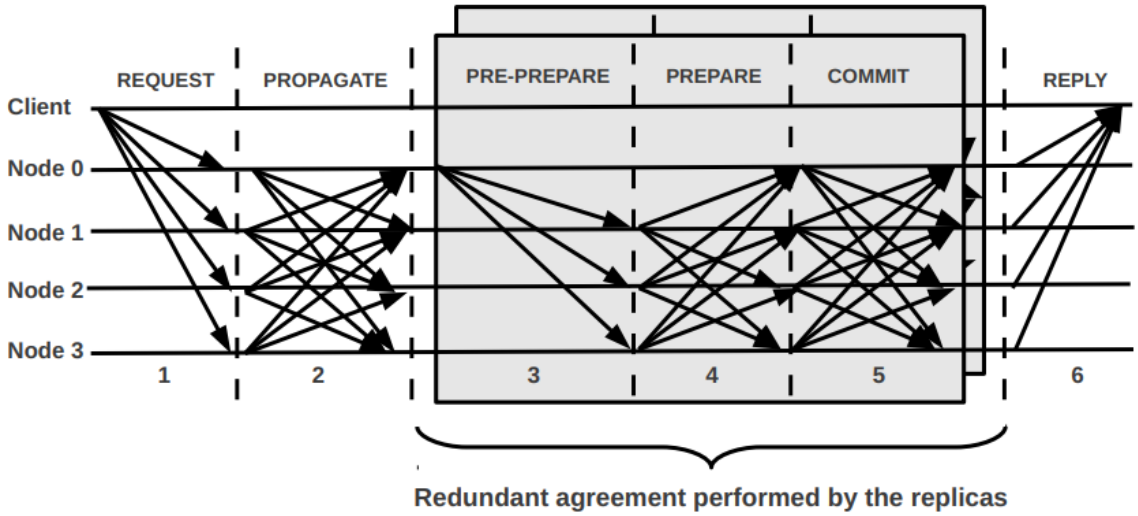

Figure 1. Redundant Byzantine Fault Tolerance protocol steps

The issue with practical-BF is that a primary can be smartly malicious. Despite efforts from other replicas to control that it behaves correctly, it can slow the performance down to the detection threshold, without being caught. **Redundant-BFT** solves this issue, In RBFT, multiple instances of a PBFT protocol are executed in parallel as shown in Figure 2. Each instance has a primary replica. The various primary replicas are all executed on different machines. While all protocol instances order requests, only one instance (called the master instance) effectively executes them. Other instances (called backup instances) order requests in order to compare the throughput they achieve to that achieved by the master instance. If the master instance is slower, the

primary of the master instance is considered malicious and The replicas elect a new primary, at each protocol instance.

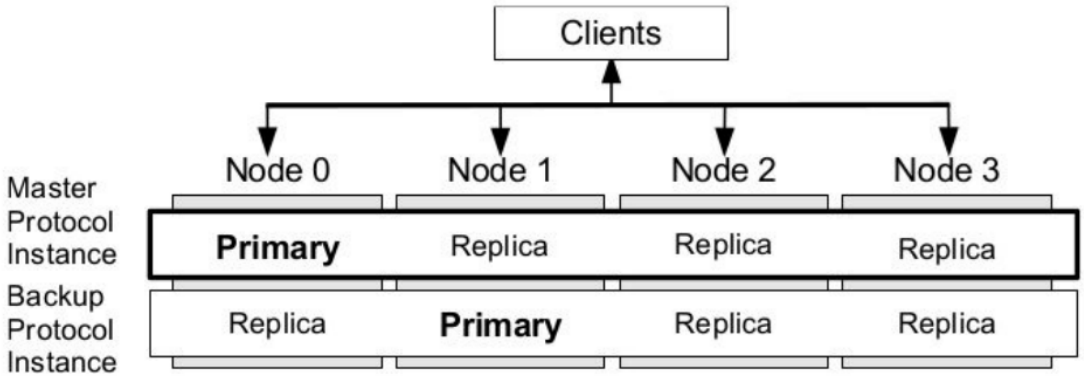

Figure 2. Redundant Byzantine Fault Tolerance Overview

Redundant-BFT Properties:

- Tolerance against Byzantine Faults
- Tolerance against malicious primary that can slow the performance down to the detection threshold (F).
  Compared to PBFT
  |R| = 3F + 1 where F is the maximum number of replicas that may be faulty and R is the total number of nodes.

● Security Issues

Security Against a Malicious Identity Auditor. A malicious auditor may try to verify a PII without checking its validity. In this system, when a user wants to upload a document, three random auditors will be responsible for verifying the uploaded document. If one auditor out of these three rejects the document then the document is invalidated.

Security Against Malicious identity verifier. A malicious verifier may try to use users' PII for other reasons rather than verification (e.g selling users' PII). Security, in this case, can be achieved by using Zero-Knowledge proofs, and non-reusable proofs as shown in Figure 3.

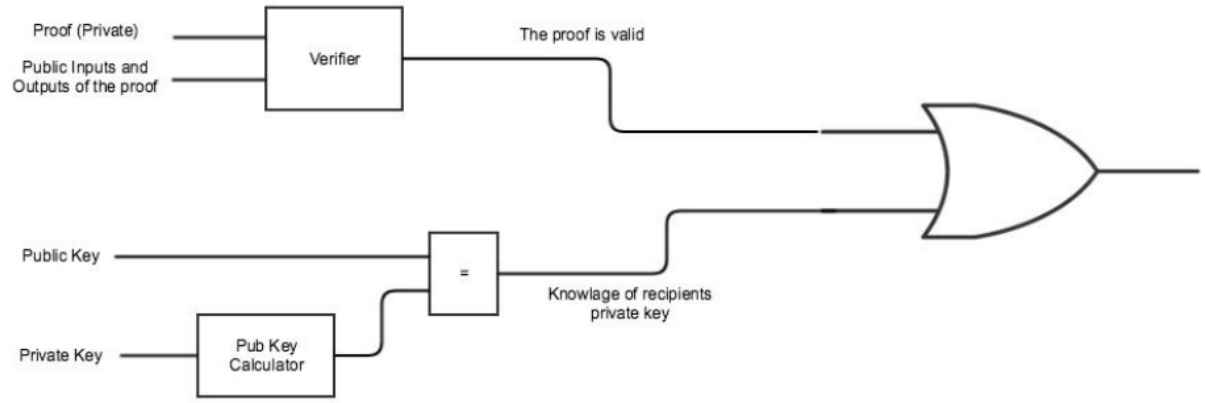

Figure 3. non-reusable proofs mechanism

- **System challenges**

- **Who will pay the transaction fees?**

The government might pay so that the cost of centralized servers is reduced. The mutual relationship between two entities that pay to each other is similar to the government-citizen relationship in which the citizen pays taxes and the government offers services.

- **Malicious Identity Auditor in Real Life**

  A malicious identity auditor may try to share your documents without consent when you submit your document.

  We could use a watermark on the document so if the auditor tries to share someone’s document, the malicious auditor will be identified and punished. This issue still exists now in real life, for example, a corrupt government employee may try to share someone's passport document when he/she issues the document. More reliable access control methods could be implemented, like monitoring the activities on the screen, or in the room, and implementing reliable hiring policies to hire the most trustworthy.

III. Implementation

The following will illustrate how the important functionalities attributed to the smart contract work for our system.

```
    //audit requests for the auditors

    struct AuditionRequest{

        address owner; // identity owner id

        address documentID; // document id

        address auditor; // auditor id

    }
```

This is the audition request, as soon as someone uploads a document 3 structs are made

For three different auditors to audit, It contains 3 addresses:

1- address owner: it’s the id of the person who uploaded the document. It will be the same for all three audition requests.

2- documentID: it’s the id of the document uploaded.

3- auditor: it’s the address of the auditor who is supposed to audit the document.

```
    //'to-be-verified' requests for the verifier(service
provider)
```

```
    struct VerificationRequest{

        address owner;

        address verifier; // verifier id

        bool isVerified; // To check if a document is truly
verified

        bool isVerificationRequired; // to check if a document
needs verification

    }
```

This is the verification request,

It contains 2 addresses and 2 Boolean type variables:

1- Owner: the id of the person who needs to be verified.

2- Verifier: the service provider that is requesting to verify the owner.

3- IsVerified: this is a Boolean which indicates if the id is verified or not.

4- isVerificationRequired: this is a Boolean which indicates if the id needs to be verified or not, it is used to block unrequested verifications and thus improving the security in the smart contract.

the variables are initialized (a new struct is created) once a service provider calls the function verificationRequest.

```
    // List of documents uploaded by the owner

    struct Document{

        bool isAudited; // Check if document is audited by the
auditor

        bool isDeclined; // Check if the document uploaded is
declined

        string title; // Title of the document uploaded e.g
passport

    }
```

This structure represents each document in the network

It contains 2 addresses and a string:

1,2- isAudited, isDeclined: this is initialized as false and when any of the auditors verifies the document there is 2 options: 1- he approves the document and it switches into true 2- he/she can reject the document and it will become false and the isDeclined switches to true and in this situation even if other auditors try to approve it, they will not be able to since it is rejected by at least 1 auditor.

3- title: it’s the title of the document for example passport,driver’s license.

```
    struct Owner{

  // A score to check the trustworthiness of an identity owner

        int256 trustScore;

        mapping(address=>Document) documents;

    }
```

It represents any regular person in the system, It contains an integer that represents the trust score and mapping functions which act as an array for the documents that the person has uploaded.

```
    function checkAuditor(address sender) internal view
returns(bool){

        for(uint256 i=0;i<auditors.length;++i)

            if( sender == auditors[i])

                return true;

        return false;

    }
```

This function goes through the auditors to check if the address passed in is an auditor. So that only a person who is an auditor is able to go through the transaction.

```
    function checkAdmin(address sender) internal view
returns(bool){

        for(uint256 i=0;i<admins.length;++i)

            if(sender == admins[i])

                return true;

        return false;

    }
```

This function goes through the admins to check if the address passed in is an admin. Just like the function before it. It will check if the person is an admin.

The above two functions are used in the modifier parameters.

```
    //To check if the person auditing is an actual auditor

    modifier isAuditor(){

        require(checkAuditor(msg.sender),"Caller is not an
auditor!");

        _;    }
```

The modifier uses the function checkAuditor to make sure that the caller of any function with this modifier applied is an auditor and blocks any other addresses from calling the same function.

```
    modifier isAdmin(){

        require(checkAdmin(msg.sender),"Caller is not an
admin!");

        _;

    }
```

The modifier uses the function checkAdmin to make sure that the caller of any function with this modifier applied is an admin and blocks any other addresses from calling the same function.

```
    // upload a document request by the identity owner/user

    function uploadDocument(address _documentID , string memory
_title)           public{

// initially the document is neither audited nor declined by
//an auditor

bool _isAudited = false;

bool _isDeclined = false;
owners[msg.sender].documents[_documentID]=Document(_isAudited,
```

```
_isDeclined,_title);

        owners[msg.sender].trustScore = 0; // initially
//trustScore is zero



        //Request an audit request from 3 random auditors

        auditRequest(msg.sender,_documentID);

    }
```

This function can be called by anyone; it adds a document into the system and automatically initializes the variables of the document by the parameters sent by the caller.

The document id and the title also set the owner of the document as the address of the caller.And sends 3 audition requests to three random auditors.

```
    // Audit request for 3 random auditors

    function auditRequest(address _ownerID ,address _documentID) internal {

        for(uint256 i=0;i<3;i++){

            auditionList.push(AuditionRequest(_ownerID,_documentID,random()));
// Pick a random auditor to check the uploaded document        }    }
```

This function can only be called in the above function; it sends 3 audition requests to 3 random auditors by adding them into a list of auditions stored in the smart contract.

```
    function audit(address _documentID , address _ownerID , bool isApprove,              int256 _feedback) public isAuditor{

        //If request is not approved

        if(!isApprove){
owners[_ownerID].documents[_documentID].isDeclined=true;

        }
owners[_ownerID].documents[_documentID].isAudited=true;

        owners[_ownerID].trustScore +=_feedback;

    }
```

This function is called by an auditor to audit a document its parameters are 4

1- _documentID: the id of the document to be audited.

2- _ownerID: the owner of the document that needs to be audited.

3- isApprove: true or false to approve or reject the document.

4- _feedback: the value added to the owner's trust score the auditor can remove trustscore by using negative values.

```
    function verificationRequest(address _ownerID) public
returns (uint256){

        //Algorithm to verify if owner is who claims to be

        uint256 verifierNum =
uint256(uint160(address(msg.sender)));

        uint256 senderNum = uint256(uint160(address(_ownerID)));

        uint256 _specialNum=verifierNum + senderNum; // Result-a
special number- is physically/manually handed to the identity
owner

        // Add information to verificationList

        verificationList[_specialNum].owner= _ownerID;

        verificationList[_specialNum].verifier= msg.sender;

        verificationList[_specialNum].isVerified= false;

        verificationList[_specialNum].isVerificationRequired=
true;
```

```
        return _specialNum;

    }
```

This function is called by a service provider that needs to verify an owner.

It converts the verifier's id into an int and the id of the person also into an int and adds the two values creating a special number that will be returned by the function and will be given to the person that needs to be verified physically, the person will use that number later to verify himself as the real owner of the id.

```
    function verifyOwnership(uint256 spcnum) public{

        if(verificationList[spcnum].isVerificationRequired &&
verificationList[spcnum].owner == msg.sender ){

            verificationList[spcnum].isVerified=true;

        }

    }
```

This function is used by the person who needs to be verified; it takes the special number as a parameter.

Removes the int value of the senders and converts the result to an address that supposedly will represent the address is the verifier now it checks if the verifier is needing verification of the person who is calling this function and if so, the verification process will be done and the person will be verified by the particular service provider.

```
    function getVerificationStatus(uint256 spcnum) public view
returns (bool){

        return verificationList[spcnum].isVerified;    }
```

This function is used by the service provider that was previously called a verification request; it takes the special number created by a verification request as a parameter.

And checks if the person is verified and then returns the result.

```
// Add auditor- only admins can do that
function addAuditor(address _auditorID) public isAdmin{
    if(!checkAuditor(_auditorID))
        auditors.push(_auditorID);
    else
        revert();
}
```

This function is used only by an admin to add a new auditor to the system.

REFERENCES

- BLOCKCHAIN IDENTITY MANAGEMENT: ENABLING CONTROL OVER IDENTITY in leewayhertz.com

- Actors in Identity Management with Blockchain in tykn.tech

- Yang Liua, Debiao Hea, Mohammad S., Neeraj Kumarf , Muhammad Khurram Khang , Kim-Kwang Raymond Chooh “Blockchain-Based Identity Management Systems: A Review” in sciencedirect.com

- Sovrin’s White Paper

- Indy Plenum’s documentation

- Hyperledger Indy Custom Network with Indy Node & Plenum

- RBFT: Redundant Byzantine Fault Tolerance

- iden3.io