K. J. Somaiya College of Engineering, Mumbai-77
(Autonomous College Affiliated to University of Mumbai)

University of Mumbai

Hierarchical Distributed Ledger for Ensuring Reliable and Authentic data Collection in IoT with a Peer to Peer Ethereum Blockchain Network

Submitted in partial fulfillment of requirements

For the degree of

Bachelor of Technology

by

Pranav Kalambe
1913023

Shinjini Bhattacharya
1913066

Himani Dave
1913074

Nikshita Shetty
1913117

Guide
Prof. Shilpa Vatkar

Department of Electronics and Telecommunication Engineering
K. J. Somaiya College of Engineering, Mumbai-77
(Autonomous College Affiliated to University of Mumbai)

Batch 2019 -2023

Certificate

This is to certify that the dissertation report entitled **Hierarchical Distributed Ledger for Ensuring Reliable and Authentic data Collection in IoT with a Peer to Peer Ethereum Blockchain Network** is bona fide record of the dissertation work done by **Pranav Kalambe, Shinjini Bhattacharya, Himani Dave, Nikshita Shetty** in the year 2022-23 under the guidance of **Prof. Shilpa Vatkar** of Department of Electronics and Telecommunication Engineering in partial fulfillment of requirement for the Bachelor of Technology degree in Electronics and Telecommunication Engineering of University of Mumbai.

Guide

Head of the Department

Principal

Date:

Place: Mumbai-77

Certificate of Approval of Examiners

We certify that this dissertation report entitled **Hierarchical Distributed Ledger for Ensuring Reliable and Authentic data Collection in IoT with a Peer to Peer Ethereum Blockchain Network** is a bona fide record of project work done by **Pranav Kalambe, Shinjini Bhattacharya, Himani Dave, Nikshita Shetty**. This project is approved for the award of Bachelor of Technology Degree in Electronics and Telecommunication Engineering of University of Mumbai.

____

Internal Examiner

____

External Examiner

Date:

Place: Mumbai-77

DECLARATION

We declare that this written thesis submission represents the work done based on our and / or others' ideas with adequately cited and referenced the original source. We also declare that we have adhered to all principles of intellectual property, academic honesty and integrity as we have not misinterpreted or fabricated or falsified any idea/data/fact/source/original work/ matter in my submission.

We understand that any violation of the above will be cause for disciplinary action by the college and may evoke the penal action from the sources which have not been properly cited or from whom proper permission is not sought.

Signature of the Student **Roll No. 1913023**	**Signature of the Student** **Roll No. 1913066**
Signature of the Student **Roll No. 1913074**	**Signature of the Student** **Roll No. 1913117**

Date:

Place: Mumbai-77

Abstract

This project aims to address the challenges associated with data transactions between data collectors and third-party buyers by leveraging the power of blockchain technology. As IoT devices generate an enormous amount of data, companies that require data often purchase it from companies that generate it. However, third-party buyers of data often face challenges in trusting the authenticity of the data they purchase. Additionally, the risk of data security breaches during data transmission poses a serious threat to data integrity and confidentiality. To overcome these challenges, we have created a decentralized application (DApp) that utilizes smart contracts and distributed ledger technology to facilitate the secure buying and selling of data. Our blockchain-based DApp offers enhanced data security, transparency, trust, privacy, and authenticity, while also empowering users to maintain control over their own data. Through this project, we aim to enable seamless and secure data transactions.

Keywords - Blockchain, Smart contracts, Web3, Ethereum, DApp, Marketplace, IoT data.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

INTRODUCTION

This chapter provides a concise overview of the project, including its purpose and underlying concept. It also highlights the motivation behind the project and outlines its scope.

1.1 Background

In today's growing world, "Data is the new fuel" and "Connected devices are growing at an exponential rate". Numerous companies generate IoT data as they possess IoT devices and the necessary infrastructure to collect data, they are called data collectors. On the other hand, there are also numerous companies that rely on raw data to conduct analysis and processing, in order to derive useful insights or outcomes that can aid their business. Data being a valuable resource, companies that require data often purchase it from companies that generate it. But the third-party buyers of data often face challenges in trusting the authenticity of the data they purchase from data-generating companies. It can be difficult for them to verify the accuracy and reliability of the data, and ensure that it is not just random or fabricated information. Another challenge in the transaction of data between data collectors and third-party buyers is the risk of data security breaches. Hackers and intruders can potentially intercept, modify, steal, or delete data during transmission, posing a serious threat to the integrity and confidentiality of the data

1.2 Motivation

Looking at the need for a fair and transparent system to facilitate the buying and selling of authentic IoT sensor data. We wanted to create a decentralized platform that enables data owners or managers to monetize the data generated through their sensors while also keeping in mind the importance of authenticity and integrity of the data being traded. We chose to leverage Blockchain technology as it provides a decentralized and transparent ledger that can enhance trust between buyers and sellers. It ensures that data transactions are recorded in an immutable and auditable manner, reducing the potential for fraud or manipulation. We also wish to tackle problems related to financial incentives with the help of Smart Contracts as they can automatically execute payments based on predefined conditions, ensuring a transparent and equitable compensation system. Our motivation is to build a solution that offers benefits to both data providers and buyers, creating a more reliable and sustainable market for IoT sensor data.

1.3 Scope of the project

Our aim is to develop a Decentralized Application (DApp) that harnesses the potential of blockchain technology for facilitating secure transactions related to data buying and selling. By utilizing distributed ledger technology and smart contracts, our DApp provides improved data security, transparency, and trust, while addressing the existing challenges associated with data transactions. Our blockchain-based solution also enhances data privacy and authenticity, and enabling users to retain full control over their data at the same time.

1.4 Organization of report

This report follows a structured approach and consists of five chapters. The introduction chapter sets the context and motivation for the project, while the literature survey chapter provides an overview of existing research and technologies related to the project, including an analysis of similar projects and their limitations. The product design chapter defines the problem statement and proposes a system to solve it, detailing the architecture, functionality, and features of the proposed system. The implementation chapter describes the steps taken to create the proposed system, while the conclusion chapter summarizes the limitations of the project and outlines potential future developments.

Chapter 2

LITERATURE SURVEY

This chapter deals with the literature survey done on similar topics and explores various proposed solutions discussed in the research papers.

In this paper, “Monetization of loT data using smart contracts” [1] the proposed system for the monetization of IoT data with automated payment in a way that involves no intermediary has 4 entities – the owner of the IOT device, the MQTT broker, the device contract and the customer. The data generated by the devices is aggregated and collected by an MQTT broker hosted on the cloud. The MQTT broker allows the device to publish the data on a specific topic and allows access to the data by public users through subscribing messages. The MQTT broker also authenticates valid customers using smart contracts. The IoT device is equipped with a regular MQTT API to connect with the broker. The device publishes data to its listed topics on the broker, which are then published by the broker to the subscribed customers. The broker has an added communication interface to interact with the smart contract using JS Services. Upon successful data access, an automatic payment in ether (the cryptocurrency token of Ethereum) based on use time is deducted from the customer balance and is sent to the owner of the IoT device. The customer calls the function ‘subscribe’ in the smart contract with the arguments ‘amount’ and the topic id to which it wants to subscribe. After successful subscription, the customer can request for access function of the contract with the specific topic id. This returns the calculated access time depending on the deposit amount by the customer and the token to access the MQTT broker. Further, the customer can directly request the Broker for data and the broker publishes data to the customer for the time interval previously set. After disconnection, the broker updates the subscription time (total time elapsed) in the contract and the owner of the device is paid in ethers by the smart contract. The paper used Ethereum based smart contracts, Remix IDE to interact with the functions and debug the outputs.

This paper, “A Decentralized Solution for IoT Data Trusted Exchange Based-on Blockchain” [3] proposed platform uses blockchain technology to ensure secure and transparent data exchange between different IoT devices without the need for intermediaries. The architecture includes four layers: data layer, network layer, protocol layer, and interaction layer. The data layer collects and stores data generated by IoT devices, while the network layer manages the exchange of data using blockchain technology. The protocol layer ensures secure and efficient communication between devices and the network, and the interaction layer provides a user-friendly interface for participants to access and exchange data. The major component design of the solution includes three types of smart contracts: exchange management contracts, data management contracts, and user management contracts. The exchange management contracts handle the exchange of data between IoT devices on the blockchain network, the data management contracts handle the storage and retrieval of data on the network, and the user management contracts manage the identities and permissions of users on the network. Overall, the proposed solution provides a secure, decentralized, and efficient way to exchange data in the IoT ecosystem, with potential applications in areas such as smart cities, healthcare, and manufacturing.

This paper, “IDMoB: IoT Data Marketplace on Blockchain” [4] proposes a decentralized system that stores IoT data and creates a searchable data marketplace for AI/ML companies, using blockchain technology and a secondary decentralized file storage layer. The proposed data marketplace targets non-real time and noncritical IoT systems that push monitoring data to the data backend in large time intervals (>30 mins). To achieve this, the IoT device data is uploaded from the gateway to a secondary decentralized file storage layer, such as IPFS or Swarm, in an encrypted form. The Swarm client returns a file handle, which is a unique identifier and address of the data, and is visible on the blockchain. To prevent unauthorized access, the IoT data uploaded to Swarm is encrypted with a symmetric key before uploading, and the payload metadata should include the name of the encryption scheme and a key index. The proposed system also outlines guidelines and rules for the implementation and governance of the data marketplace, which aims to provide secure and reliable data storage and sharing for non-real time and noncritical IoT systems. Additionally, the system provides steps for accessing the data using a smart contract function, including the encryption and decryption of the symmetric key.

This paper, “An Implementation of a Blockchain-based Data Marketplace using Geth” [5] explores the implementation of a blockchain-based data marketplace utilizing the Go Ethereum (Geth) library. The implementation consists of an IoT node powered by a raspberry pi zero W, which is utilized to collect data from the environment and store it in an InterPlanetary File System (IPFS) external server, a web page that displays the marketplace, and a private blockchain that records transactions. The proposed system consists of an IoT node powered by a raspberry pi zero W and a DHT11 temperature/humidity sensor. This node connects to an InterPlanetary File System (IFPS) private server in order to store the collected data which is deployed in the Amazon Web Services (AWS) cloud. A private Ethereum network was configured using Geth (Go Ethereum) and involves 2 smart contracts. One of the contracts deals with recording information about the data and to allow others to see what data is stored in the marketplace. The second contract is to allow sellers to white/blacklist buyers’ access to the data. The process begins with the data being uploaded to the IPFS server, after which the hash of the file is returned. This IPFS hash, together with other metadata is sent to the smart contract. The web interface can query the blockchain and display the data in the marketplace. Interested buyers can then search for and purchase data using a smart contract, which verifies the buyer's payment and records the transaction in the blockchain. Once access is granted, the buyer can download the data from an IPFS server. This system described in the paper provides a secure and transparent way for buyers and sellers to exchange data.

Chapter 3

PROJECT DESIGN

This chapter defines problem statement and objectives of the project. It provides a brief overview of the proposed system, including its web interface and the technologies required to build it.

3.1 Problem Statement

To ensure authentic and secure transaction of data from vendors to buyer, a transparent system is needed that eliminates the requirement for third-party brokers and resolves issues related to data confidentiality, integrity, and authenticity. The objective is to create a tamper-proof and transparent data transfer process while ensuring the privacy of the shared data. This will leverage the benefits of blockchain technology to develop an efficient and dependable system for secure data transfer.

3.2 Objectives

- To create an end-to-end system which facilitates secure transactions between buyers and sellers of data.
- To ensure data authenticity, integrity and confidentiality.
- To design an interface that is intuitive for the user to view and manage data according to their preferences.
- To setup a private blockchain network and maintain a distributed ledger.
- To eliminate third party brokers and provide a cost-efficient solution.

3.3 System Architecture and Working

The architecture for our system consists of two levels of blockchain network.

- ➢ Level one is a local blockchain that is implemented individually for each data generator or manager. Each manager has their own local blockchain network, which includes all the sensor nodes owned and managed by that particular manager, along with its local identity.

  The local blockchain network is responsible for collecting IoT data generated by the sensor nodes and transferring it securely from the sensor nodes to the manager. The local blockchain network also ensures data privacy and confidentiality, and is secured through encryption and validation mechanisms.

- Level two is a global blockchain network that connects all the local blockchain networks created by the data generators or managers. This global blockchain network consists of all the managers and entities who want to buy the data (bidders), and it includes the global identity of managers.

  The main aim of the global network is to facilitate the buying and selling of data. Managers who have collected data via their sensor nodes through the local blockchain network can put their data up for sale on the global network, and bidders can bid on this data to purchase it. The global blockchain network ensures data authenticity and transparency, and enables secure data transactions through the use of smart contracts and distributed ledger technology.

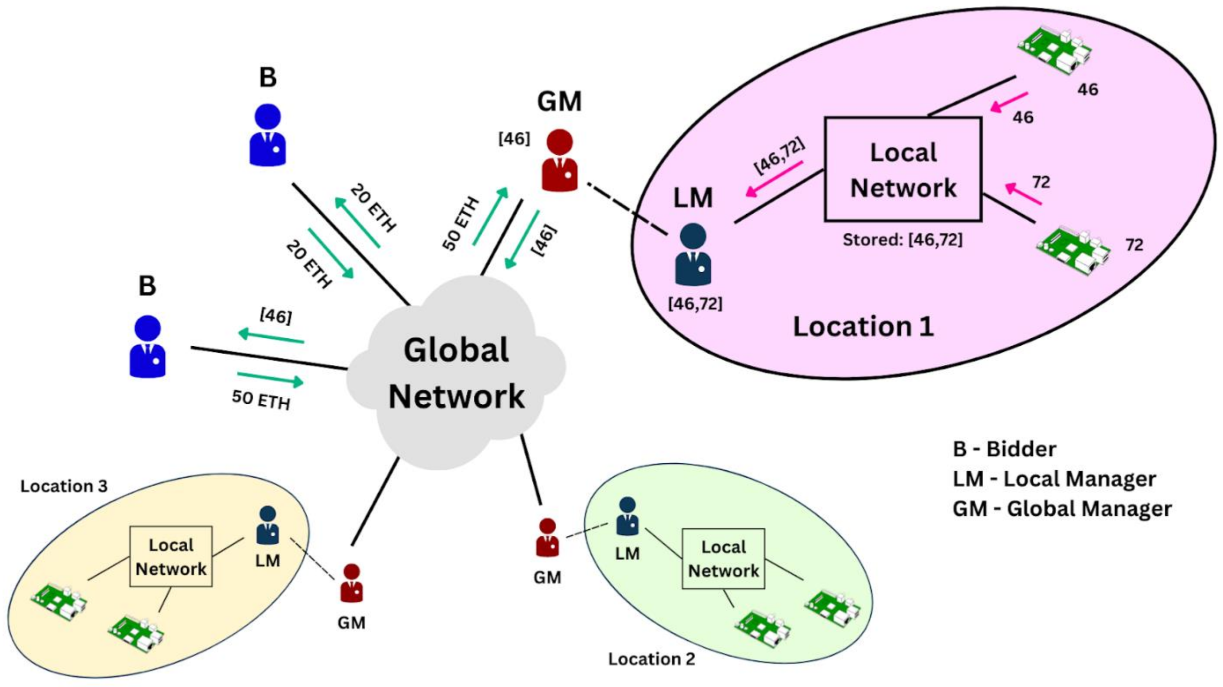


Fig. 1. System Architecture

The overview of the working step-by-step is:

1. Data Generation: The sensor nodes owned by the data manager generate IoT data.
2. Local Blockchain: The data generated by the sensor nodes is sent to the local blockchain network, which is implemented individually for each data manager. The local blockchain network is responsible for securely transferring the data from the sensor nodes to the manager.
3. Data Management: The local instance of manager—which is a part of the local network, fetches the data from the local blockchain network. The manager further analyzes the data and decides which data to put up for sale on the global blockchain network.
4. Global Blockchain: The selected data is uploaded to the global blockchain network, by the global instance of manager. The global network allows different entities, such as bidders or buyers, to bid on the data put up for sale.
5. Bidding Process: Bidders can place their bids on the data, and the highest bidder will ultimately be able to purchase the data. The global blockchain network ensures that the bidding process is secure and transparent by using smart contracts and distributed ledger technology.
6. Transfer of Ownership: Once the bidding process is closed, the ownership of the data will be transferred to the highest bidder, who will be able to access and use the data.
7. Payment Settlement: The global instance of manager, who put the data up for sale, will receive the amount of the highest bid in their wallet. The amount will be in the form of Ether-- which is the cryptocurrency used on the Ethereum blockchain. The other bidders who placed lower bids will have their money refunded to their wallets.

3.4 Proposed System

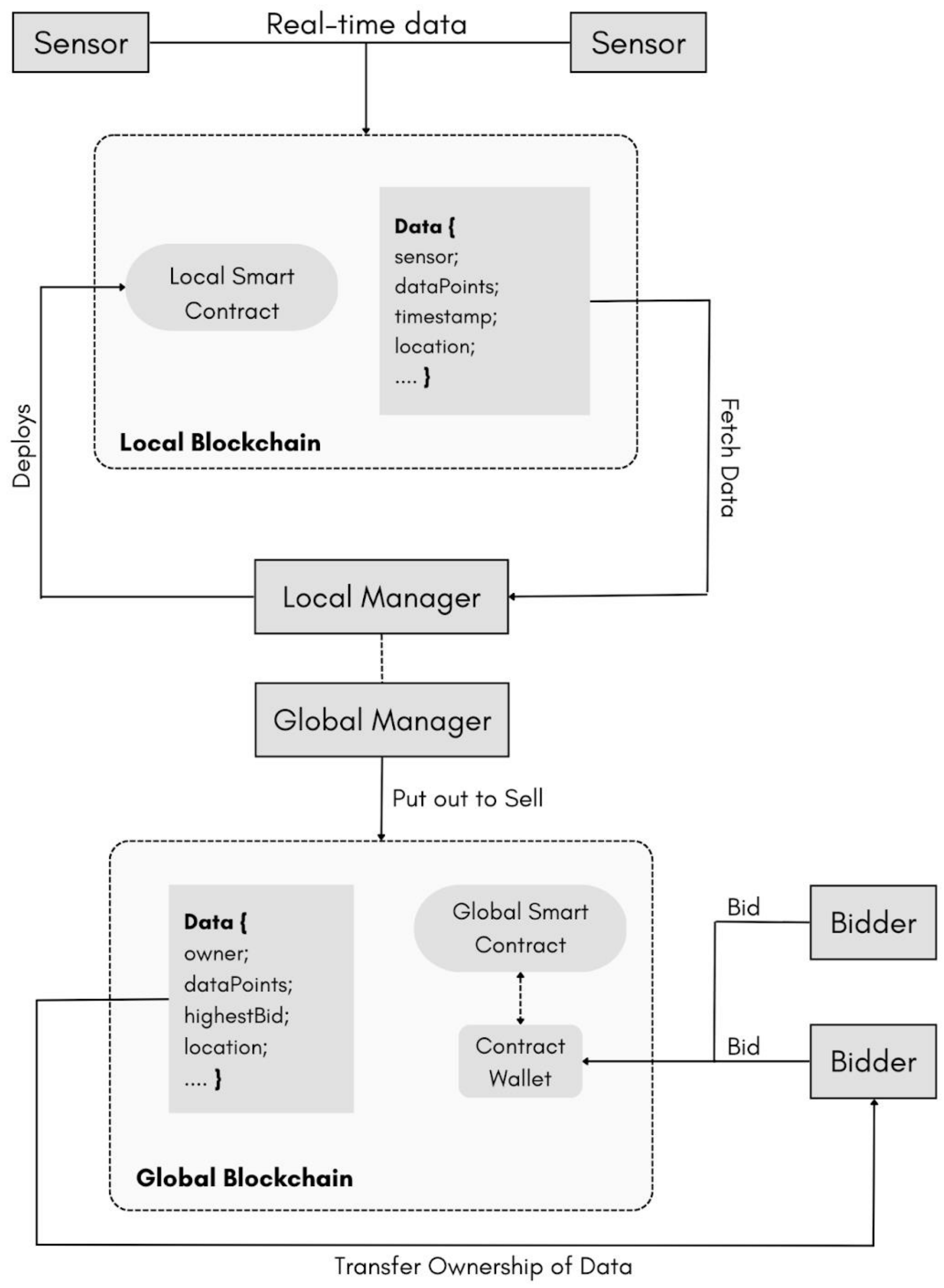


Fig. 2. Proposed System

3.5 Web Interface Structure

The web app consists of two main parts:

1. Manager Setup:

   The process of setting up the data collection system is a one-time step that does not need to be repeated. This initial setup involves the data collector manager entering various details, such as their global and local network addresses, the number of sensor nodes, and the location of the nodes. Additionally, the manager needs to add the addresses of each sensor into the system. After the system is set up, the manager can activate the sensor nodes using a web interface. Once the sensor is activated, they will automatically start sending data over the local network without requiring any further manual intervention from the manager.

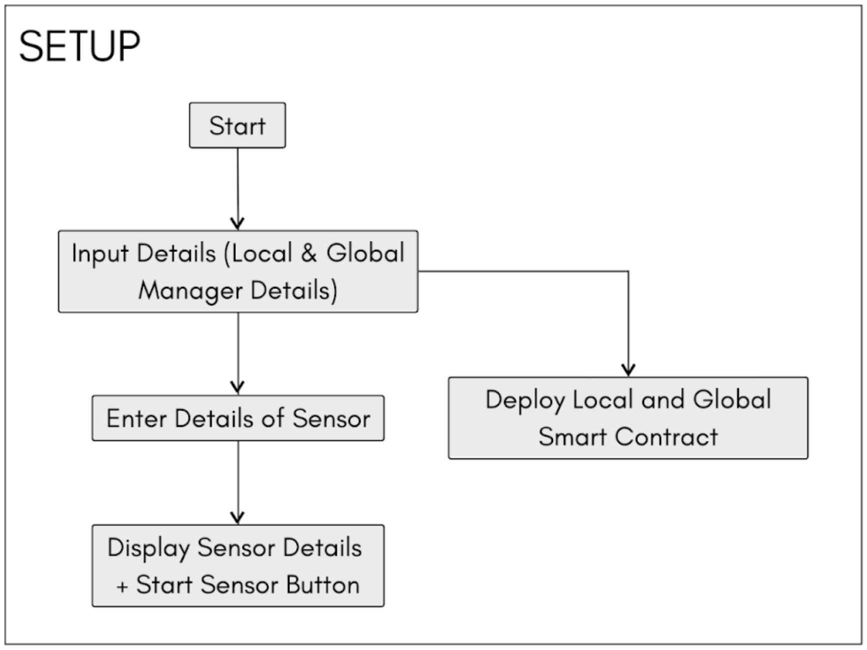

SETUP
Start
Input Details (Local & Global Manager Details)
Enter Details of Sensor
Deploy Local and Global Smart Contract
Display Sensor Details + Start Sensor Button

Fig. 3. Web App Setup Flow

2. Global Platform:

   The system provides a primary interface that allows for the transaction of data on the global network. This interface includes various options, such as a portfolio section that displays all the data collected by the sensor nodes and transferred over the local network to that particular manager.

   Using the data ID, the manager can select specific data from their portfolio and put it up for bidding in the bidding space. Other entities that are part of the global network can view the data available for bidding and place their bids accordingly.

   After the bidding period has ended, the owner of the data can close the bid, and the entity with the highest bid receives the data in their data procured section. Meanwhile, the manager who put the data up for bidding receives the bid amount in their wallet, which can be viewed in the navigation bar.

   This bidding process allows for a streamlined and efficient way of selling and buying data, enabling entities to access valuable insights and information that can benefit their business or research. The interface provides a secure and transparent method of data transaction, allowing for easy tracking of bids and ensuring the confidentiality and privacy of sensitive information

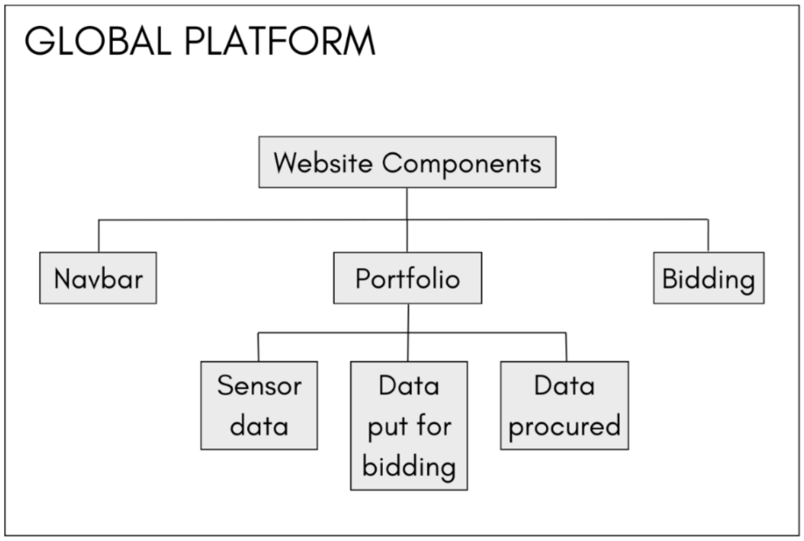


Fig. 4. Web App Components

3.6 Technologies Used

3.6.1 Blockchain

Blockchain is a system of recording information in a way that makes it difficult or impossible to change, hack, or cheat the system. A blockchain is essentially a digital ledger of transactions that is duplicated and distributed across the entire network of computer systems on the blockchain. Each block in the chain contains a number of transactions, and every time a new transaction occurs on the blockchain, a record of that transaction is added to every participant's ledger. The decentralized database managed by multiple participants is known as Distributed Ledger Technology (DLT).

3.6.2 Ethereum

Ethereum is a decentralized blockchain platform that establishes a peer-to-peer network that securely executes and verifies application code, called smart contracts. Smart contracts allow participants to transact with each other without a trusted central authority.

3.6.3 Smart Contract

A smart contract is a self-executing contract with the terms of the agreement between buyer and seller being directly written into lines of code. The code and the agreements contained therein exist across a distributed, decentralized blockchain network. The code controls the execution, and transactions are trackable and irreversible. Smart contracts permit trusted transactions and agreements to be carried out among disparate, anonymous parties without the need for a central authority, legal system, or external enforcement mechanism. Once a smart contract is published to Ethereum, it will be online and operational for as long as Ethereum exists. Not even the author can take it down. Since smart contracts are automated, they do not discriminate against any user and are always ready to use.

3.6.4 Geth (Go Ethereum)

Go Ethereum is one of the three original implementations (along with C++ and Python) of the Ethereum protocol. It is written in Go, fully open source and licensed under the GNU LGPL v3. A golang implementation of Ethereum blockchain in the form of a standalone client software. Geth is one of the preferred alternatives for running, setting up nodes, and interacting with Ethereum blockchains due to its ease of use.

Geth allows the following advantages for users:

- Customizable: Geth provides extensive customization options for configuring a private blockchain network.
- Secure: Geth is designed with security in mind, with features like account management and encryption.
- Efficient: Geth is built to be fast and efficient, making it a good choice for private blockchain networks that require high throughput and low latency.

3.6.5 Web3

Web3, also known as the decentralized web, is a next-generation web architecture that enables the creation of decentralized applications (dApps) and peer-to-peer interactions without the need for intermediaries. This is made possible by blockchain technology, which provides a secure and transparent mechanism for storing and exchanging data.

We have leveraged the Web3 library in JavaScript – which is a popular tool for interacting with blockchain networks such as Ethereum. It provides a simple and convenient way to write scripts and applications that can send and receive data from the blockchain.

Some of the most common use cases for the Web3 library in JavaScript include:

- Accessing Blockchain Data: Web3 can be used to access data from the blockchain, including transaction history, account balances, and more.
- Interacting with Smart Contracts.
- Building DApps.

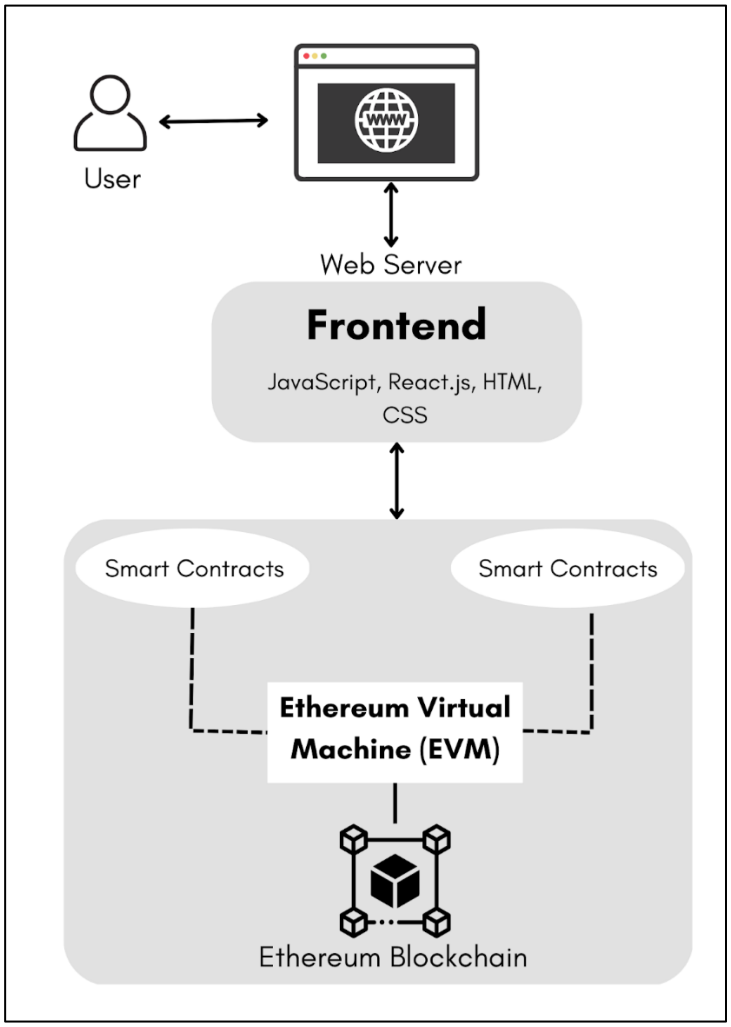


Fig. 5. Web3 Architecture

3.6.6 React

React (also known as React.js or ReactJS) is a free and open-source front-end JavaScript library for building user interfaces based on UI components. It is maintained by Meta (formerly Facebook) and a community of individual developers and companies. React can be used as a base in the development of single-page, mobile, or server-rendered applications with frameworks like Next.js.

There are multiple advantages of using React.js in developing applications such as:

- Reusability: React's reusable components reduce code duplication and improve development efficiency by allowing developers to reuse them across multiple applications.
- Performance: React is known for its high performance and efficient rendering. It uses a virtual DOM (Document Object Model) to minimize the number of DOM manipulations required, resulting in faster and smoother user interfaces.
- Declarative approach: React uses a declarative approach to programming, which makes it easier to reason about and debug code.

3.7 Software Description

3.7.1 EVM (Ethereum Virtual Machine)

The Ethereum Virtual Machine or EVM is a piece of software that executes smart contracts and computes the state of the Ethereum network after each new block is added to the chain. The EVM sits on top of Ethereum's hardware and node network layer. Its main purpose is to compute the network's state and to run and compile various types of smart contract code into a readable format called 'Bytecode.'

3.7.2 Bytecode & ABI

As Ethereum uses EVM (Ethereum Virtual Machine) as a core component of the network, smart contract code written in high-level languages needs to be compiled into EVM bytecode to be run. Bytecode is an executable code on EVM and Contract ABI is an interface to interact with EVM bytecode. For example, if a function has to be called in a smart contract with the JavaScript code, ABI plays a role as an intermediary between the JavaScript code and EVM bytecode to interact with each other. The diagram below shows the architecture of Contract ABI, EVM bytecode and outside components (DApp and network). The left side is a process of compiling and the right side is interacting.

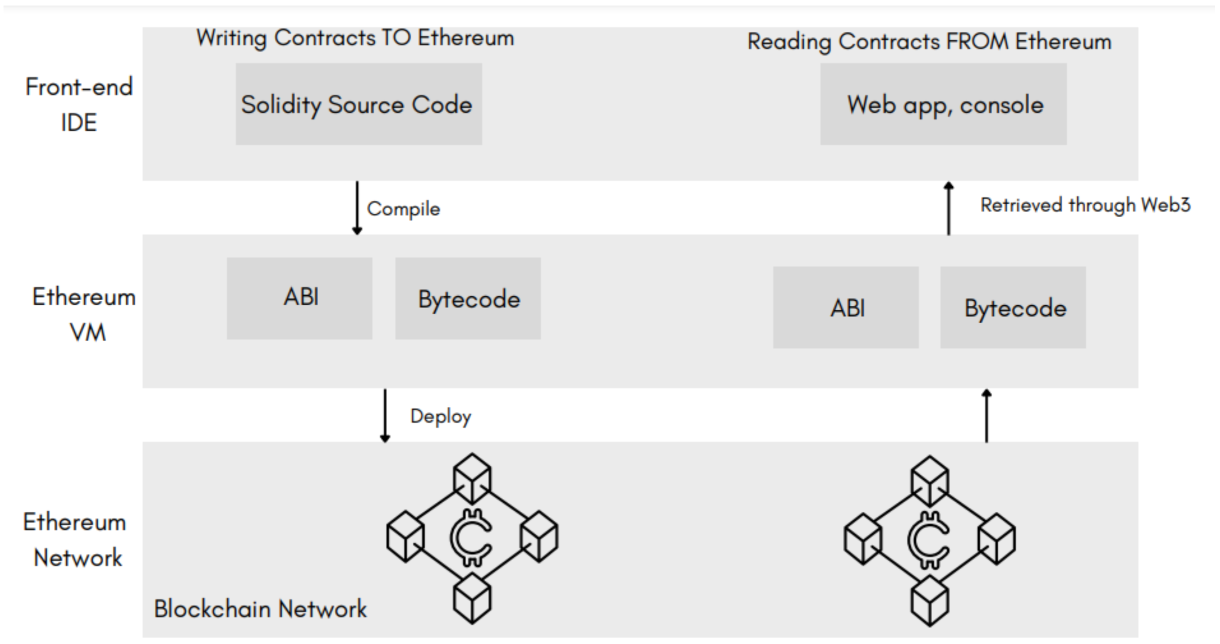


Fig. 6. EVM Working

EVM bytecode is a low-level programming language which is compiled from a high-level programming language such as solidity.
ABI (Application Binary Interface) In Ethereum, Contract ABI is an interface that defines a standard scheme of how to call functions in a smart contract and get data back. Contract ABI is designed for external use to enable application-to-contract and contract-to-contract interaction.

3.7.3 Geth in Private Network

An Ethereum network is private if the nodes are not connected to the main network. A network is considered private if it is composed of multiple Ethereum nodes that can only connect to each other. In order to run multiple nodes locally, each one requires a separate data directory (--datadir). The nodes should also know about each other and be able to exchange information, share an initial state and a common consensus algorithm.

Geth is a command-line interface (CLI) tool that is widely used as an Ethereum client to interact with Ethereum blockchain networks. Geth can be particularly useful for private blockchain networks for specific use cases. Here are a few reasons why:

- Customization: Geth can be customized to meet the specific requirements of a private blockchain network, such as modifying the consensus mechanism or the gas limit. This makes it easier to tailor the network.
- Security: Geth supports various security features, such as private key management and encryption, to help ensure the security and integrity of the network.
- Control: In a private blockchain network, the network operators have more control over the network's configuration and governance.

General Steps to enable a private network:

- ➢ Choosing a Network ID
- ➢ Choosing a Consensus algorithm (Ethash, Clique)
- ➢ Creating the Genesis block
- ➢ Initializing the Geth database
- ➢ Setting IP networking
- ➢ Running member nodes
- ➢ Adding other nodes as peers in the network
- ➢ Running a miner

These steps are further implemented in the implementation part of the report.

3.7.4 Ethereum

Ethereum is a decentralized blockchain-based software that has smart contract functionality. Ethereum is open source and used primarily to support the second-largest cryptocurrency in the world known as Ether. Ethereum enables the smart contracts and applications built on its blockchain to run smoothly without fraud, downtime, control, or any third-party interference. Ethereum is a popular platform for decentralized application (dApp) development due to several reasons:

- ❖ Smart contracts: Ethereum enables developers to write and deploy smart contracts, which are self-executing code. This allows for the creation of decentralized applications that can run without the need for intermediaries.

- ❖ Security: Ethereum is a secure platform due to its use of a public blockchain, which is maintained by a decentralized network of nodes.
- ❖ Community: Ethereum has a large and active developer community that is constantly working to improve the platform and create new tools and applications.

Chapter 4

IMPLEMENTATION AND EXPERIMENTATION

This chapter provides a detailed explanation of the step-by-step process for implementing the complete system. It includes relevant images, code snips and output images to understand the process more clearly.

4.1 Overall Steps

I. Set up a private Ethereum network: To set up a private Ethereum network, Geth needs to be installed and configured on each node, as per the instructions given in the official documentation of Geth. The network can be customized by assigning unique IP addresses and ports to each node.

II. Create a smart contract: A smart contract needs to be created next, which will determine the rules and data structure for transmitting data from sensors to local managers via the private network. Solidity, the programming language used for Ethereum smart contracts, can be used for this purpose. The smart contract can be customized according to the specific requirements of each local manager.

III. Deploy the smart contract: The smart contract created above needs to be deployed on the local private network using the Geth console, and assigned to a specific address so that it can be accessed by the local managers.

IV. Automate the smart contract: Automating the smart contract using Javascript is recommended so that it can be easily accessed and used by the local managers. The web3.js library can be used to interact with the smart contract.

V. Set up Global Network: For creating a multi-node private network consisting of global managers and bidders/buyers, similar steps can be repeated.

VI. Develop global smart contract: A global smart contract needs to be created that determines the rules and data structure for the transaction of data from the seller to the bidder with the highest bid via the global private network.

VII. Deployment of Global Contract: Once the global smart contract has been created, it needs to be deployed on the global private network using the Geth console.

VIII. Create a web interface: Finally, a web interface can be created using HTML, CSS, and Javascript to interact with the blockchain network. The web3.js library can be used to connect to the network and perform transactions on the smart contracts.

4.2 Setting up private blockchain using Geth

Objectives of the implementation:

- Setup a blockchain with multiple nodes
- Setup mining nodes
- Connect the multiple nodes and setup the blockchain network
- Test the blockchain network by mining blocks and verifying that blocks are propagated to all the nodes
- Verify that the local copy of blockchain on all the nodes is updated

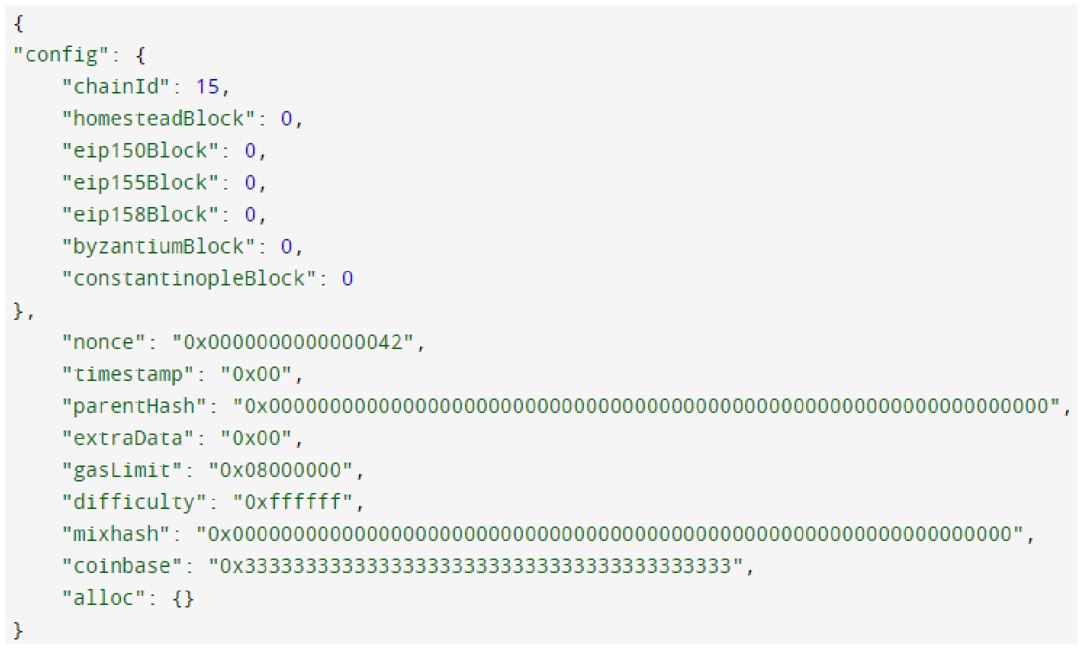
Initial Setup:

- Install geth from its official website – https://geth.ethereum.org/downloads/
- Create a project directory – which will have the block data of the nodes.
- Configure the Genesis Block

4.2.1 Genesis Block

It is the starting point of every blockchain – that is, a block "zero" or the very first block of the chain. It will be the only block without a predecessor which is hard coded.

The genesis block for our global network is as follows:



```
{
"config": {
    "chainId": 15,
    "homesteadBlock": 0,
    "eip150Block": 0,
    "eip155Block": 0,
    "eip158Block": 0,
    "byzantiumBlock": 0,
    "constantinopleBlock": 0
},
    "nonce": "0x0000000000000042",
    "timestamp": "0x00",
    "parentHash": "0x0000000000000000000000000000000000000000000000000000000000000000",
    "extraData": "0x00",
    "gasLimit": "0x08000000",
    "difficulty": "0xffffff",
    "mixhash": "0x0000000000000000000000000000000000000000000000000000000000000000",
    "coinbase": "0x3333333333333333333333333333333333333333",
    "alloc": {}
}
```

Fig. 7. Genesis Block

Some of the parameters in the genesis file above is described as following:

- config: This block defines the settings for your custom chain.
- chainId: This identifies your Blockchain, and because the main ethereum network has its own, you need to configure your own unique value for your private chain.
- homesteadBlock: Defines the version and protocol of the ethereum platform.
- eip155Block / eip158Block: These fields add support for non-backward-compatible protocol changes to the Homestead version used. For the purposes of our project, we won't be leveraging these, so they are set to "0".
- difficulty: This value controls block generation time of the blockchain. The higher the value, the more calculations a miner must perform to discover a valid block.
- gasLimit: Gas is ethereum's fuel spent during transactions.
- Nonce: The nonce is an arbitrary variable that a miner continually changes in order to be awarded the right to add a block to the blockchain.

- parentHash: Each block refers to the previous block called the parent block by including the parent block's hash in a special field of its header. since this is a genesis block, it will not have any parent– therefore the hash value is 0**.**

4.2.2 Setting up a node

TERMINAL commands

Initialize a data directory

geth --datadir managerLocal init genesis.json

Configuring the node in the blockchain with the given parameters.

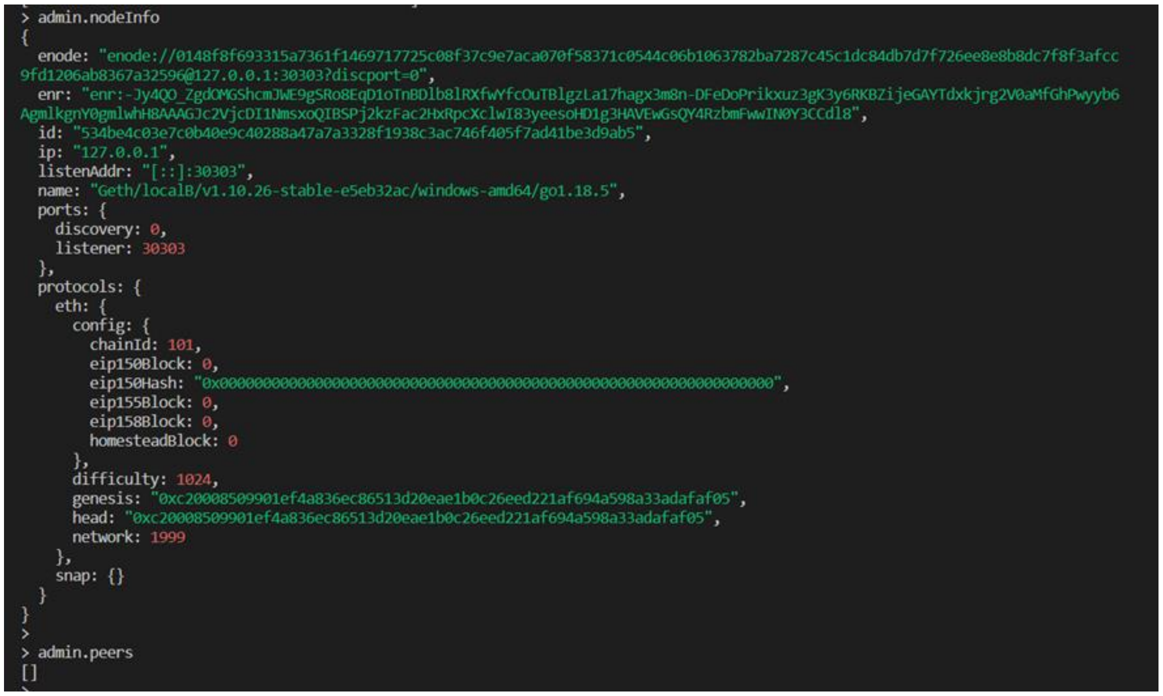
geth --identity "localB" --http --http.port 8041 --http.corsdomain "" --http.api "db,eth,net,web3,personal" --datadir managerLocal --port "30301" --authrpc.port 8530 --nodiscover --networkid 100 --ipcdisable console --allow-insecure-unlock*

The parameters of this command:

- --*datadir managerLocal*: specifies the data directory of the blockchain. If we do not specify this, it will default to the main Ethereum blockchain.
- --*networkid1999*: identity of our Ethereum network, other peer nodes will also have the same network identifier. The network id can be any random integer value.
- --*allow-insecure-unlock*: This flag can be used when running Geth to allow unlocking an account with an insecure password over an HTTP connection as in certain situations, like in testing or debugging, it may be necessary to unlock an account with an insecure password over an HTTP connection.
- --*http*: this flag enables the HTTP-RPC server, which allows users to interact with Geth using HTTP requests.
- *authrpc.port:* this flag is used, it enables the authenticated RPC server on the specified port number.

4.2.3 View Node

Command: *admin.nodeInfo*



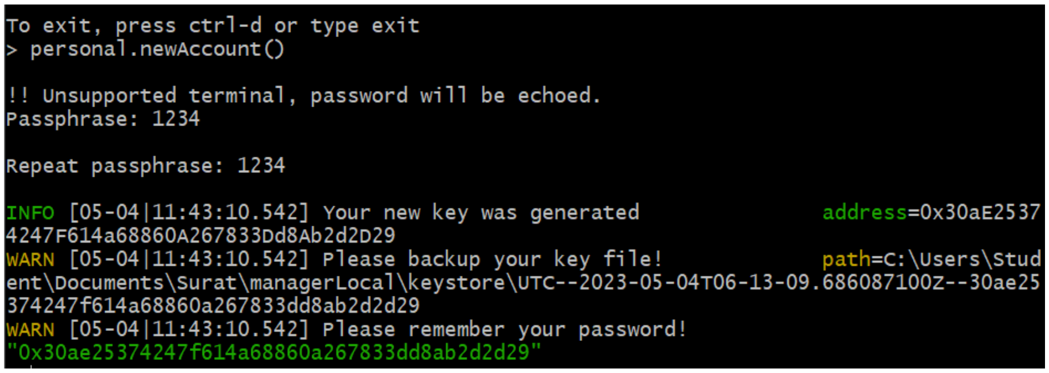
```
> admin.nodeInfo
{
  enode: "enode://0148f8f693315a7361f1469717725c08f37c9e7aca070f58371c0544c06b1063782ba7287c45c1dc84db7d7f726ee8e8b8dc7f8f3afcc
9fd1206ab8367a32596@127.0.0.1:30303?discport=0",
  enr: "enr:-Jy4QO_ZgdOMGShcmJWE9gSRo8EqD1oTnBDlb8lRXfwYfcOuTBlgzLa17hagx3m8n-DFeDoPrikxuz3gK3y6RKBZijeGAYTdxkjrg2V0aMfGhPwyyb6
AgmlkgnY0gmlwhH8AAAGJc2VjcDI1NmsxoQIBSPj2kzFac2HxRpcXclwI83yeesoHD1g3HAVEwGsQY4RzbmFwwIN0Y3CCdl8",
  id: "534be4c03e7c0b40e9c40288a47a7a3328f1938c3ac746f405f7ad41be3d9ab5",
  ip: "127.0.0.1",
  listenAddr: "[::]:30303",
  name: "Geth/localB/v1.10.26-stable-e5eb32ac/windows-amd64/go1.18.5",
  ports: {
    discovery: 0,
    listener: 30303
  },
  protocols: {
    eth: {
      config: {
        chainId: 101,
        eip150Block: 0,
        eip150Hash: "0x0000000000000000000000000000000000000000000000000000000000000000",
        eip155Block: 0,
        eip158Block: 0,
        homesteadBlock: 0
      },
      difficulty: 1024,
      genesis: "0xc20008509901ef4a836ec86513d20eae1b0c26eed221af694a598a33adafaf05",
      head: "0xc20008509901ef4a836ec86513d20eae1b0c26eed221af694a598a33adafaf05",
      network: 1999
    },
    snap: {}
  }
}
>
> admin.peers
[]
```

Fig. 8. Node Info

The enode id acts like an unique identifier of the particular node. This enode id is required when a node has to be added as a peer to another node in the network.

4.2.4 Create Ethereum Account

Command: *personal.newAccount()*



```
To exit, press ctrl-d or type exit
> personal.newAccount()

!! Unsupported terminal, password will be echoed.
Passphrase: 1234

Repeat passphrase: 1234

INFO [05-04|11:43:10.542] Your new key was generated             address=0x30aE2537
4247F614a68860A267833Dd8Ab2d2D29
WARN [05-04|11:43:10.542] Please backup your key file!           path=C:\Users\Stud
ent\Documents\Surat\managerLocal\keystore\UTC--2023-05-04T06-13-09.686087100Z--30ae25
374247f614a68860a267833dd8ab2d2d29
WARN [05-04|11:43:10.542] Please remember your password!
"0x30ae25374247f614a68860a267833dd8ab2d2d29"
```

Fig. 9. New Ethereum Account

4.3 Connecting Multiple Local Nodes

Post all the nodes are set up and configured, we proceed to add one as a peer to another node in order to create a peer-to-peer (p2p) network. If both the nodes are on same local device the simple steps are

1. Add a Peer:

   Command: *admin.addPeer("//enode id")*

   where 'enode id' is the node id of the other node.

2. Check Peers:

   A peer refers to a node that is connected to the Ethereum network and is capable of communicating with other nodes on the network to synchronize data, propagate transactions and blocks, and participate in the consensus mechanism. To ensure that the local network is properly connected, we can check the peer count and peer details to verify.

   Command: *admin.peers()*

4.4 Connecting Multiple PCs to form a global network

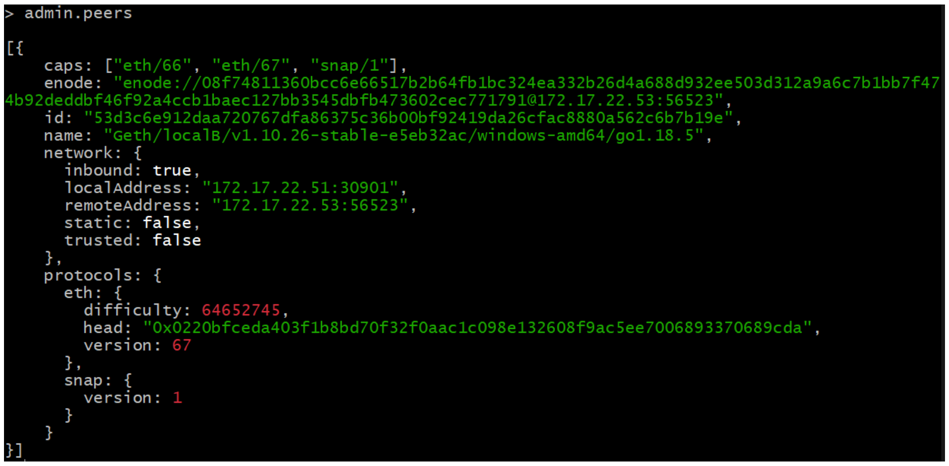
To connect multiple PCs using Geth in a private network with the help of static nodes, we followed the following steps:

1. The steps to set up a Geth private network and create a new account on each computer that we want to connect to the network should be followed at every PC.
2. The "enode" URL for each node in the network should be noted. The URL can be obtained by running the command *"admin.nodeInfo.enode"* in the Geth console. The URL for each node should be copied.
3. A new file named "static-nodes.json" needs to be created in the Geth data directory for each node. The "enode" URLs for each node in the network should be added to this file, with each URL added on a separate line.
4. It should be noted that the enode URL obtained from each PC should contain the IP address of that PC and not the local host address. For example, "enode:.@127.0.0.1:30901?discport=0" should be changed to "enode:.@{ip-address-pc}:30901?discport=0."
5. The static nodes.json file should be placed in the data directory, and the geth terminal should be restarted with the same command.
6. Once Geth is running on each computer, they will automatically connect to each other

using the "static-nodes.json" file. The status of the network can be checked by running the following command-

Command: *admin.peers*.

This command returns an array of nodes which are currently connected to the Ethereum network.



```
> admin.peers

[{
    caps: ["eth/66", "eth/67", "snap/1"],
    enode: "enode://08f74811360bcc6e66517b2b64fb1bc324ea332b26d4a688d932ee503d312a9a6c7b1bb7f47
4b92deddbf46f92a4ccb1baec127bb3545dbfb473602cec771791@172.17.22.53:56523",
    id: "53d3c6e912daa720767dfa86375c36b00bf92419da26cfac8880a562c6b7b19e",
    name: "Geth/localB/v1.10.26-stable-e5eb32ac/windows-amd64/go1.18.5",
    network: {
      inbound: true,
      localAddress: "172.17.22.51:30901",
      remoteAddress: "172.17.22.53:56523",
      static: false,
      trusted: false
    },
    protocols: {
      eth: {
        difficulty: 64652745,
        head: "0x0220bfceda403f1b8bd70f32f0aac1c098e132608f9ac5ee7006893370689cda",
        version: 67
      },
      snap: {
        version: 1
      }
    }
}]
```

Fig. 10. Node Peers

The configuration information for our network is given in the table below:

GLOBAL BLOCKCHAIN NETWORK			
NETWORK ID - 1999			
Entity	**IP Address**	**Http Port**	**Node Port**
Mumbai Global Manager	172.17.22.51	9091	30901
Pune Global Manager	172.17.22.52	9092	30902
Nagpur Global Manager	172.17.22.53	9093	30903

Table 1. Global Network Configuration

MUMBAI LOCAL BLOCKCHAIN NETWORK		
NETWORK ID - 101		
Entity	**Http Port**	**Node Port**
Mumbai Local Manager	8041	30401
Sensor Node	8042	30402

Table 2. Mumbai Local Network Configuration

PUNE LOCAL BLOCKCHAIN NETWORK		
NETWORK ID - 202		
Entity	**Http Port**	**Node Port**
Pune Local Manager	8051	30501
Sensor Node	8052	30502

Table 3. Pune Local Network Configuration

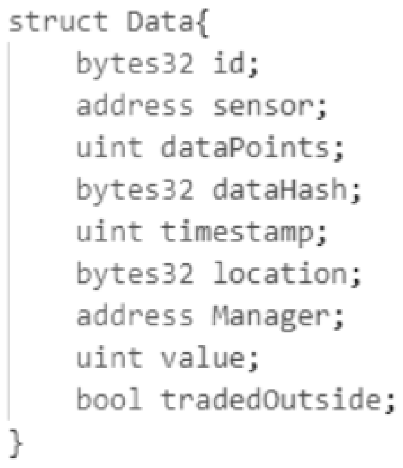
NAGPUR LOCAL BLOCKCHAIN NETWORK		
NETWORK ID - 303		
Entity	**Http Port**	**Node Port**
Nagpur Local Manager	8061	30601
Sensor Node	8062	30602

Table 4. Nagpur Local Network Configuration

4.5 Implementation of Local Smart Contract

4.5.1 Data Structures

The contract defines a struct called "Data" that represents a single data point which includes details about the data such as the data ID, the address of the sensor that collected it, the number of data points, the hash of the data, the timestamp, the location, etc. The objective of this contract is to facilitate communication between managers and their sensors on the local blockchain network.



```
struct Data{
    bytes32 id;
    address sensor;
    uint dataPoints;
    bytes32 dataHash;
    uint timestamp;
    bytes32 location;
    address Manager;
    uint value;
    bool tradedOutside;
}
```

Fig. 11. Local Data Structure

Data types in Solidity used to defined the structure:

- Bytes32: The bytes value type in Solidity is a dynamically sized byte array. It is provided for storing information in binary format. Since the array is dynamic, its length can grow or shrink. Solidity provides a wide range — from bytes1 to bytes32.
- Address: An address value type is specifically designed to hold up to 20B, or 160 bits, which is the size of an Ethereum address.
- Unsigned integers (uint): An unsigned integer, declared with the uint keyword, is a value data type that must be non-negative; that is, its value is greater than or equal to zero. 'uint' is an abbreviation for uint256.
- Mappings: Mapping in Solidity acts like a hashtable or dictionary in any other language. These are used to store the data in the form of key-value pairs, a key can be any of the built-in data types but reference types are not allowed while the value can be of any type.

We have created 2 mappings in the code, namely:

A. *mapping(address=>mapping(bytes32=>Data)) sensorData*

This is a nested mapping where the outer mapping maps an Ethereum address (sensor node) to an inner mapping, which maps a bytes32 value (data ID) to the Data struct defined above.. This mapping essentially stores all the sensor data sent to the network along with their associated metadata. This is basically a matrix of all sensorData.

B. *mapping(address=>bytes32[]) dataIdArray*

This is a mapping that associates an array of bytes32 data IDs with each address. It keeps track of all the data IDs that have been sent by a particular address (i.e. sensor) to the contract. It allows for easy retrieval of all the data IDs associated with a particular sensor.

4.5.2 Functions:

The objective of developing the below-mentioned functions in the smart contract is to enable our web application to interact with the blockchain via the contract, using web3.js. web3.js is a collection of libraries that allow us to interact with a local or remote ethereum node using HTTP, IPC or WebSocket. The main functionality of sending data to the manager, putting the data up for selling etc is carried out by the code in the smart contract.

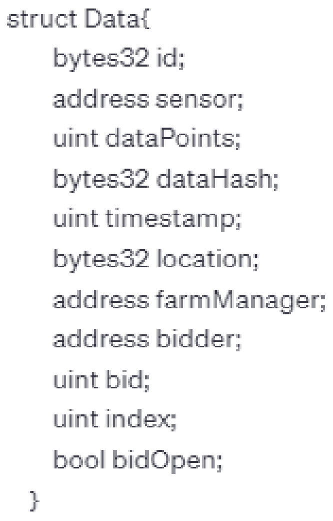
Some of the important functions in the smart contract

- ❖ sendSensorDataToManager: this function is used by the sensor nodes to send the data to their manager.
- ❖ getSensorDataCount: this function returns the length of the data for a particular address from the mapping(address=>bytes32[]).
- ❖ putOutToSell: this function enables the node to put up a particular data for trading (bidding).
- ❖ getSensorDataInfoById: this function returns the data (includes hash, timestamp, location etc) of a particular data.
- ❖ Istraded: this function returns a boolean true/false depending on whether a particular data was put up for trading or not.

4.6 Implementation of Global Smart Contract

The objective of this contract is to facilitate the exchange of data in a transparent and secure way, where managers can sell the data they obtained to interested buyers at the highest price possible.

4.6.1 Data Structure



```
struct Data{
    bytes32 id;
    address sensor;
    uint dataPoints;
    bytes32 dataHash;
    uint timestamp;
    bytes32 location;
    address farmManager;
    address bidder;
    uint bid;
    uint index;
    bool bidOpen;
  }
```

Fig. 12. Global Data Structure

This Solidity struct called "Data" defines the properties of data records related to a particular sensor node. The properties include:

- id: A unique identifier for the data record.
- sensor: The address of the sensor node that produced the data.
- dataPoints: The number of data points in the record.
- dataHash: A hash of the data.
- timestamp: The time at which the data was produced.
- location: A hash of the location where the data was produced.
- Manager: The address of the manager under which the sensor node is located.
- bidder: The address of the current highest bidder (if bidding is open).
- bid: The current highest bid (if bidding is open).
- index: The index of the data record in an array of data records.
- bidOpen: A boolean indicating whether bidding is currently open for the data record.

We created 2 mappings in the code:

A. *mapping(bytes32=>Data) data;*

   The ‘data’ mapping maps a bytes32 id to a Data struct. This mapping is used to store all the data records that are added to the contract. The Data struct contains various attributes such as the data ID, sensor address, location, manager address, bid amount, etc. The data mapping is used to store and retrieve data records based on their ID.

B. *mapping(address=>bytes32[]) dataOwned;*

   The mapping called ‘dataOwned’ maps an Ethereum address to an array of bytes32 ids. This mapping is used to keep track of the data records that are owned by each address. Each address can own multiple data records. The ‘dataOwned’ mapping is used to retrieve the list of data IDs owned by a particular address, and to add new data record IDs to an address's list when a bid is closed and ownership is transferred.

4.6.2 Functions

The functions in the contract are as follows:

❖ addNewData: This function allows a user to add a new data record to the contract. It takes in parameters such as the data ID, sensor address, timestamp, and location, etc. The function creates a new "Data" struct object and adds it to the "data" mapping. It also adds the data ID to the array of data IDs owned by the sender in the "dataOwned" mapping.

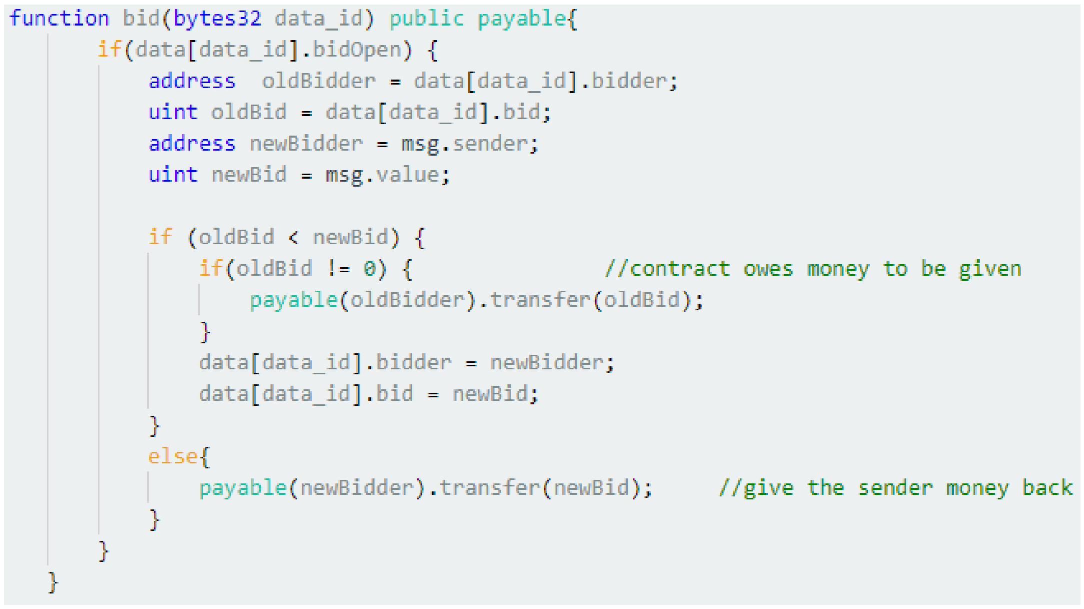
❖ bid: This function allows a user to place a bid on a specific data record. It takes in the data ID and the bid amount. If the bid is higher than the current highest bid on the record, the function updates the bid amount and bidder address in the "Data" struct object. If the previous bidder had a non-zero bid, the contract returns the previous bid to the bidder using the ‘transfer’ function.

❖ getHighestBid: This function returns the highest bid on a specific data record. It takes in the data ID and returns the bid amount stored in the corresponding "Data" struct object.

❖ closeBid: This function allows the owner of a specific data record to close the bidding and transfer ownership of the record to the highest bidder. It transfers the bid amount to the owner using the "transfer" function and transfers ownership to the highest bidder.

- whoOwns: This function returns the address of the current owner of a specific data record.
- countData: This function returns the number of data records owned by a specific address.
- getDataDetails and getDataDetailsById: These functions return the details of a specific data record.

One of the most important functions of the global contract – 'bid' – which enables the whole bidding process is mentioned below:



```
function bid(bytes32 data_id) public payable{
        if(data[data_id].bidOpen) {
            address  oldBidder = data[data_id].bidder;
            uint oldBid = data[data_id].bid;
            address newBidder = msg.sender;
            uint newBid = msg.value;

            if (oldBid < newBid) {
                if(oldBid != 0) {                //contract owes money to be given
                    payable(oldBidder).transfer(oldBid);
                }
                data[data_id].bidder = newBidder;
                data[data_id].bid = newBid;
            }
            else{
                payable(newBidder).transfer(newBid);     //give the sender money back
            }
        }
    }
```

Fig. 13. Bidding Function

The bid function is a smart contract function that can be called by anyone willing to bid on a piece of data. The function takes a *data_id* parameter, which is the identifier of the data being bid on and requires a payment in the form of Ether.

When the function is called, it first checks if the bidding for the given *data_id* is still open. If the bidding is still open, the function retrieves the current highest bid and the bidder who made that bid. Then it compares the new bid being made with the current highest bid. If the new bid is higher than the current highest bid, then the function transfers the previous highest bid

amount back to the previous bidder. The function then updates the highest bid amount and bidder to the new values. If the new bid is not higher than the current highest bid, then the function does not update the highest bid and transfers the Ether back to the sender.
In this way, the bid function ensures fair and competitive bidding on the data by only accepting the highest bid and returning the money to the previous bidder if they are outbid.

4.7 Function Calling

4.7.1 Mining

Once all the nodes are connected in a private Ethereum network, mining is typically used to create new blocks and add transactions to the blockchain. Mining is important when we are trying to add transactions to the blockchain. Mining has to be enabled so that transactions are verified and added to the network. Whenever a function is called which modifies or adds anything in the network a transaction is created which is executed once mined.
Command: *miner.start()*

4.7.2 Function Call Sequence

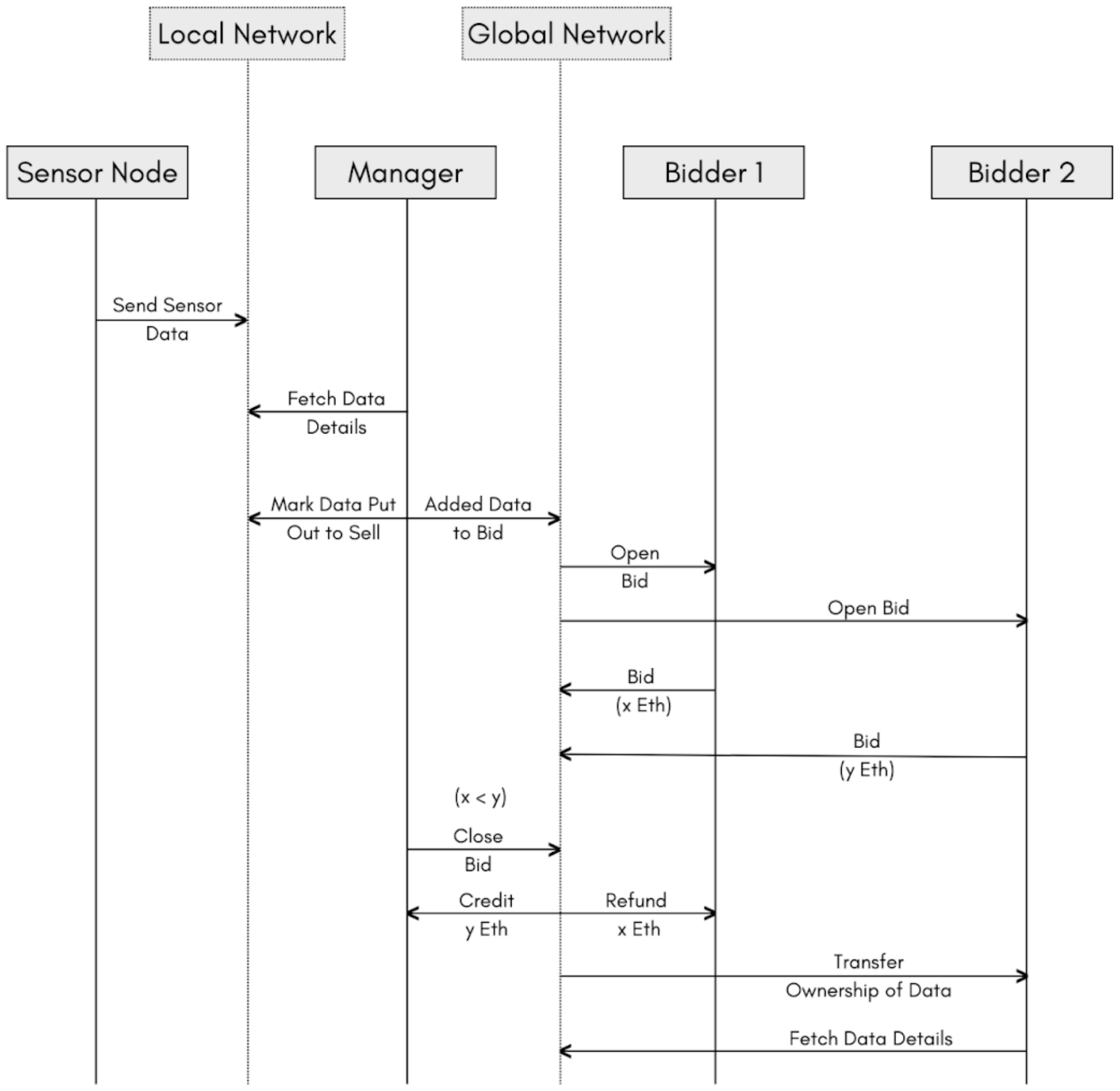
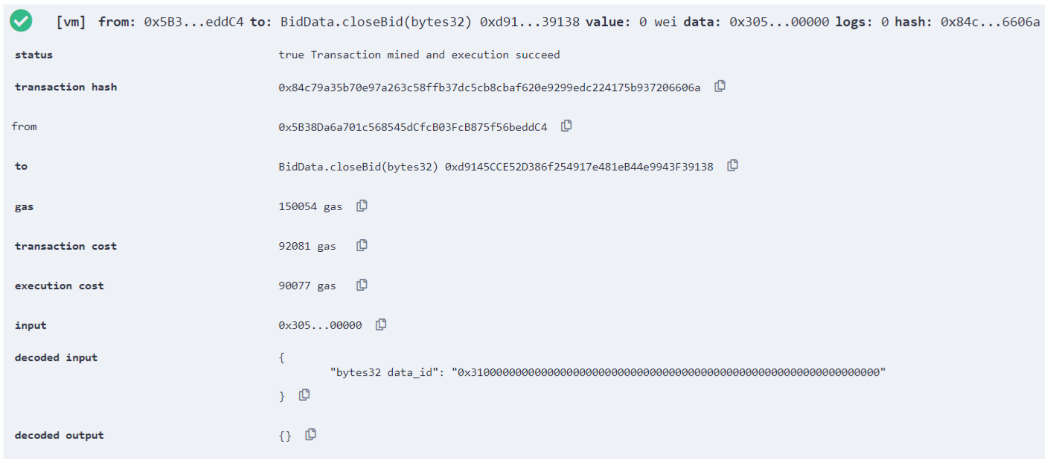


Fig. 14. Function Call

4.7.3 Transaction Details



```
[vm] from: 0x5B3...eddC4 to: BidData.closeBid(bytes32) 0xd91...39138 value: 0 wei data: 0x305...00000 logs: 0 hash: 0x84c...6606a
status                  true Transaction mined and execution succeed
transaction hash        0x84c79a35b70e97a263c58ffb37dc5cb8cbaf620e9299edc224175b937206606a
from                    0x5B38Da6a701c568545dCfcB03FcB875f56beddC4
to                      BidData.closeBid(bytes32) 0xd9145CCE52D386f254917e481eB44e9943F39138
gas                     150054 gas
transaction cost        92081 gas
execution cost          90077 gas
input                   0x305...00000
decoded input           {
                                "bytes32 data_id": "0x3100000000000000000000000000000000000000000000000000000000000000"
                        }
decoded output          {}
```

Fig. 15. Transaction Details

On examining a transaction from our web application in Remix IDE closely, we obtain the details of that transaction, for example the hash of the transaction, the account address from which the contract was called, the transaction cost, the amount of gas used. Every operation that is performed by a contract requires a certain amount of gas, which is determined by the complexity of the operation and the resources that are required to perform it. It also displays the decoded output. The output of a transaction is usually returned in its raw form, which is a hexadecimal string which is why the output must be decoded into a human-readable format such as strings, numbers, etc.

4.8 Deployment of Smart Contract

Smart contracts need to be deployed to a blockchain network in order to become active and execute the pre-defined functions. When a smart contract is deployed, it creates an instance (contract account) on the network. One can create multiple instances of a smart contract on the network or multiple networks. Deployment of a smart contract is done by sending a transaction to the network with bytecode. Deploying a smart contract also establishes ownership of that contract, meaning that only the owner of the contract can update or modify its code.

The steps followed to deploy the smart contract:

1) Set up the development environment
2) This is done by installing the necessary software and dependencies, such as *Node.js, npm, and web3.js*, etc.

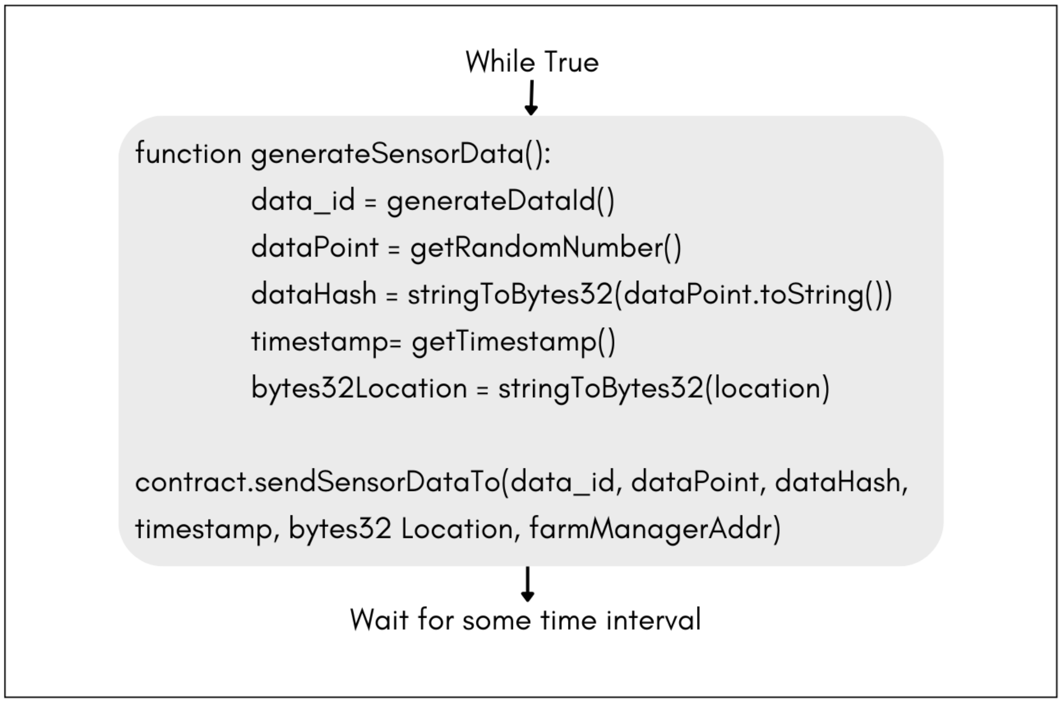
3) Write the smart contract
4) Develop the logic of the functions and write the code in Solidity and save it as a .sol file.
5) Compile the contract
6) Compile the contract using a Solidity compiler such as Solc, which generates the contract's bytecode and ABI (Application Binary Interface).
7) Command: *solc MyContract.sol --bin --abi --optimize -o build.*
8) Write the deployment script
9) Using *web3.js*, connect to the blockchain network using a Web3 provider.
10) Read the compiled contract file and extract the contract's bytecode and ABI
11) Use the *'deploy'* function of the contract instance along with parameters such as from (the account address from which the contract is being deployed), gasLimit, gasPrice.
12) Run the deployment script in a terminal using Node.js to initiate the deployment process and confirm that the contract is successfully deployed.

4.9 Emulating a sensor to send real-time data

We have tried to emulate a sensor node that continuously sends random data points to a smart contract deployed on the Ethereum blockchain. This simulation is run as a loop which generates data points after some predefined delay to mimic the behaviour of a real sensor. This function enables the manager to start a particular sensor's emulation.

The function takes in parameters such as the location, address of the selected sensor node which has to be emulated, address of the manager and sends sensor data to the smart contract. This function generates a new data ID every time and sends a transaction to the smart contract using the *sendSensorDataTo* function of the contract. The conversion of data points to a bytes32 data type before sending it as a transaction to the network, along with other conversions like datetime formatting is carried out by utility functions.

Pseudo Code:



```
While True
↓
function generateSensorData():
        data_id = generateDataId()
        dataPoint = getRandomNumber()
        dataHash = stringToBytes32(dataPoint.toString())
        timestamp= getTimestamp()
        bytes32Location = stringToBytes32(location)

contract.sendSensorDataTo(data_id, dataPoint, dataHash,
timestamp, bytes32 Location, farmManagerAddr)
↓
Wait for some time interval
```

Fig. 16. Psuedo code for emulating the sensor

4.10 Steps for creating the dynamic User Interface

4.10.1 Overview and requirements

1) Install React Framework.
2) Install react-bootstrap library which provides native bootstrap components as pure react components.
3) Install react-router-dom package for routing purposes.
4) Create a separate component for each page.
5) Use functional based components and states for easy handling of data.

The website has the following components:

- Portfolio
- Bidding

Portfolio is divided into three sub-components:

1. Data Available for Portfolio
2. Data put up for bidding
3. Data Procured

4.10.2 Data Available For Portfolio

- This is a React component that displays data available for portfolio. It fetches data from a smart contract and displays it in a table. Manager can select a data ID and click a button to put it up for bidding.

- The `useState` hook is used to define the `sensorData` state variable, which is an empty array by default. When the component mounts, the `useEffect` hook is called, which fetches data from the smart contract and updates the `sensorData` state variable with the fetched data.

- The `handleChange` function is called when the user selects a data ID from the dropdown menu. It updates the `data_id` variable with the selected data ID.

- The `putForBidding` function is called when the user clicks the "Put for Bid" button. It retrieves the sensor address associated with the user's account, encodes a transaction to put the selected data up for bidding, and sends the transaction to the smart contract. It then retrieves the details of the selected data from the smart contract and adds the data to the bidding contract by calling its `addNewData` function.

- The component renders a Bootstrap `Card` component containing a `Table` component. The `sensorData` state variable is mapped to create a row in the table for each data point fetched from the smart contract. Each row displays the data ID, data points, timestamp, hash, location, and whether or not the data has been traded.

4.10.3 Data Put for Bidding

- This is a React component that displays a table of data that has been put up for bidding, and provides a way for users to close a bid.

- When the component is mounted, it makes a call to the `bidContract` to get the count of data that has been put up for bidding, and then loops through each of the data items to get their details, which are stored in the `bdata` array. The `bdata` array is then set as the state using the `setBidData` function. The table in the UI is populated with the data in the `bidData` state.

- When the user clicks the "Close Bid" button, the `handleChange` function is called to get the value of the input field, which represents the data ID that the user wants to close the bid for. The value is stored in the `data_id` variable.

- The `closeBid` function is then called, which first converts the `data_id` string to a `bytes32` hex string using the `stringToBytes32` function. This hex string is then used to call the `closeBid` method of the `bidContract`, and the resulting encoded ABI is stored in the `closebid` variable.

- Finally, a transaction is sent to the `bidContract` to close the bid. If the transaction is successful, the transaction ID is logged to the console. If the `data_id` variable is empty, an alert is shown to the user.

4.10.4 Data Procured

- This is a React component that displays data that has been procured by the manager. The component uses the useState and useEffect hooks to manage and update the state of the bidData variable, which is an array of objects that contains information about the data that has been procured.
- The useEffect hook is used to fetch data from the Ethereum blockchain using the web3_inter and bidContract objects. The countData method is called to get the number of data items that have been procured by the manager. Then, a for loop is used to iterate over the data items and call the getDataDetails method for each item to retrieve the details of the data. The retrieved data is then added to the bdata array, and the setBidData function is called to update the state of the bidData variable with the new data.
- The component returns a Card component that contains a Table component. The Table component displays the data in a tabular format. The data is mapped over and rendered as rows in the table using the map method.

The columns of the table are as follows:

- Data_ID: The ID of the data item.
- Sensor: The type of sensor that was used to capture the data.
- Bid: The location where the data was captured.
- Data Captured: The timestamp when the data was captured.
- Manager: The address of the manager who captured the data.

Overall, this component provides a way to display the data that has been procured by a specific Manager in a clear and organized manner.

4.10.5 Bidding Process

- This is a React functional component that allows users to bid on data. The useState hook is used to manage component state, and the useEffect hook is used to fetch data when the component mounts.

- The handleAmount and handleDataID functions are used to update the amount and dataID when the user enters input values, respectively. The stringToBytes32 function converts a string to bytes32 format, which is used by the bidContract contract.

- The Bid function is called when the user clicks on the Bid button. It retrieves the data associated with the dataID using the getDataDetailsById function by interacting with the global smart contract and displays it in a modal. The bidTrans function is called when the user submits a bid, and it sends the bid transaction to the blockchain. It interacts with the smart contract method “bid” which takes data_id as the input. SendTransaction method is used for sending the transaction. Upon successful transaction it will return a transaction hash in an alert.

4.11 Web App

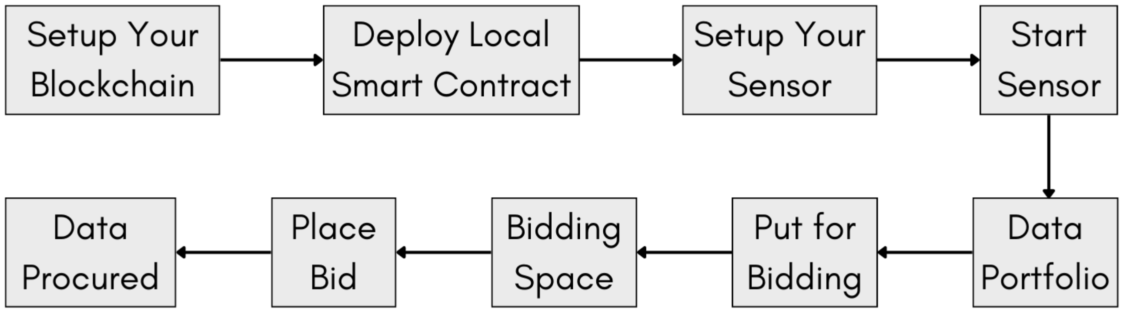
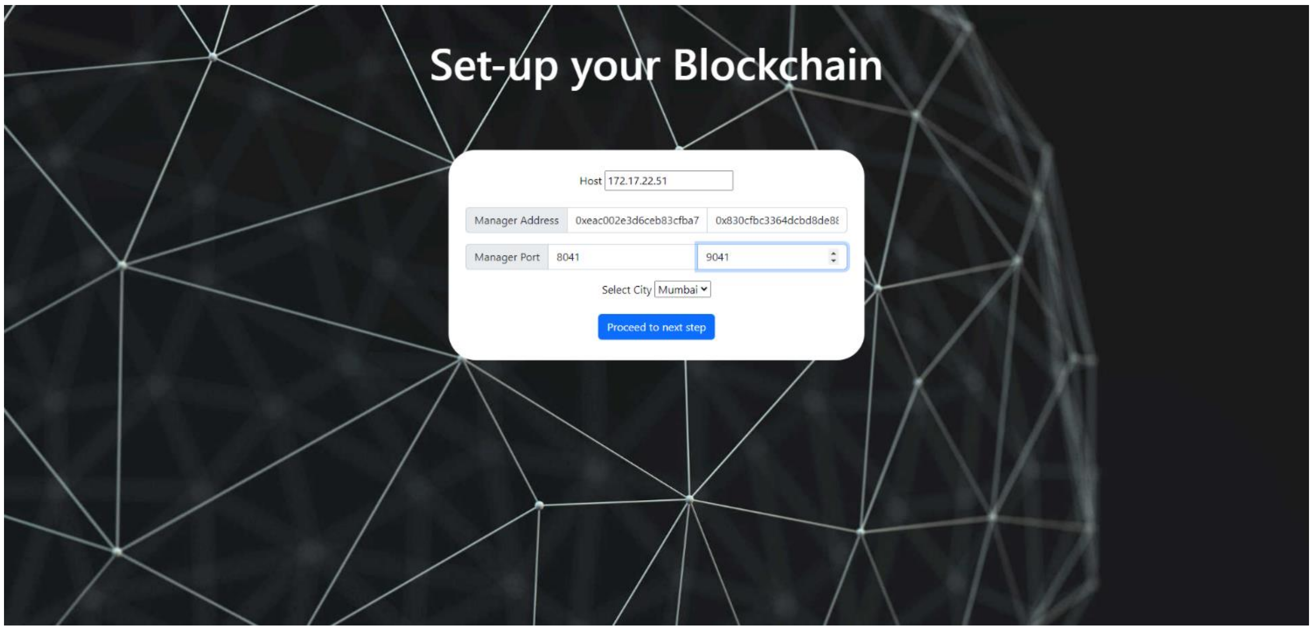


Fig. 17. Web App Flow



Set-up your Blockchain

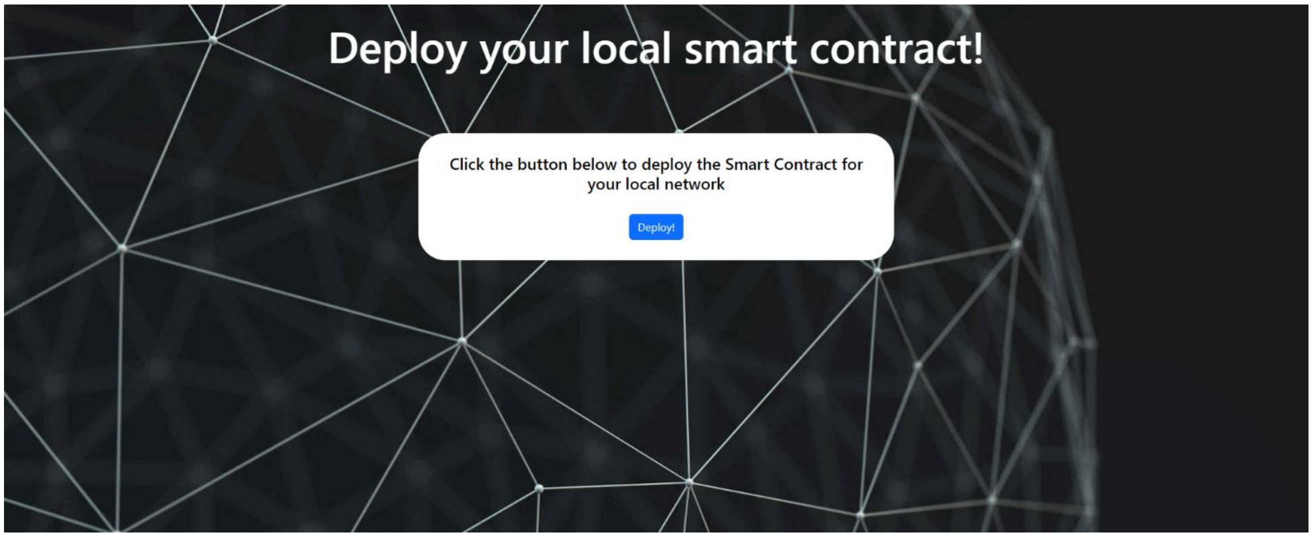
Host 172.17.22.51

Manager Address 0xeac002e3d6ceb83cfba7 0x830cfbc3364dcbd8de8

Manager Port 8041 9041

Select City Mumbai

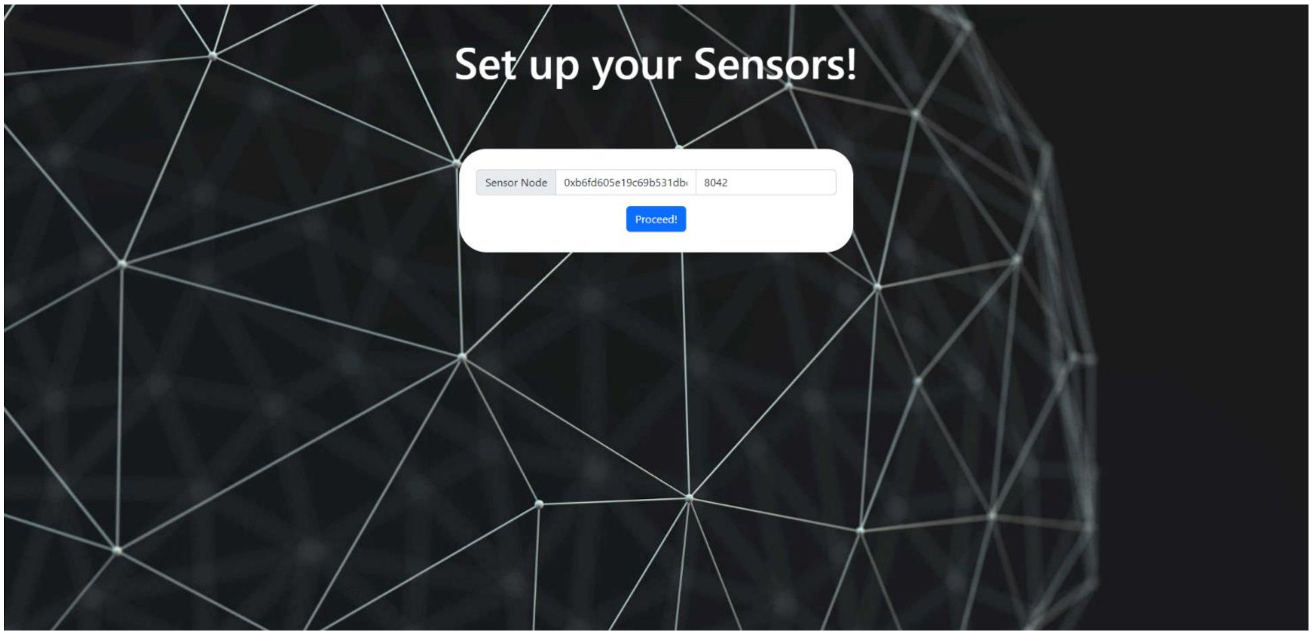
Fig. 18. Setup the blockchain



Deploy your local smart contract!

Click the button below to deploy the Smart Contract for your local network

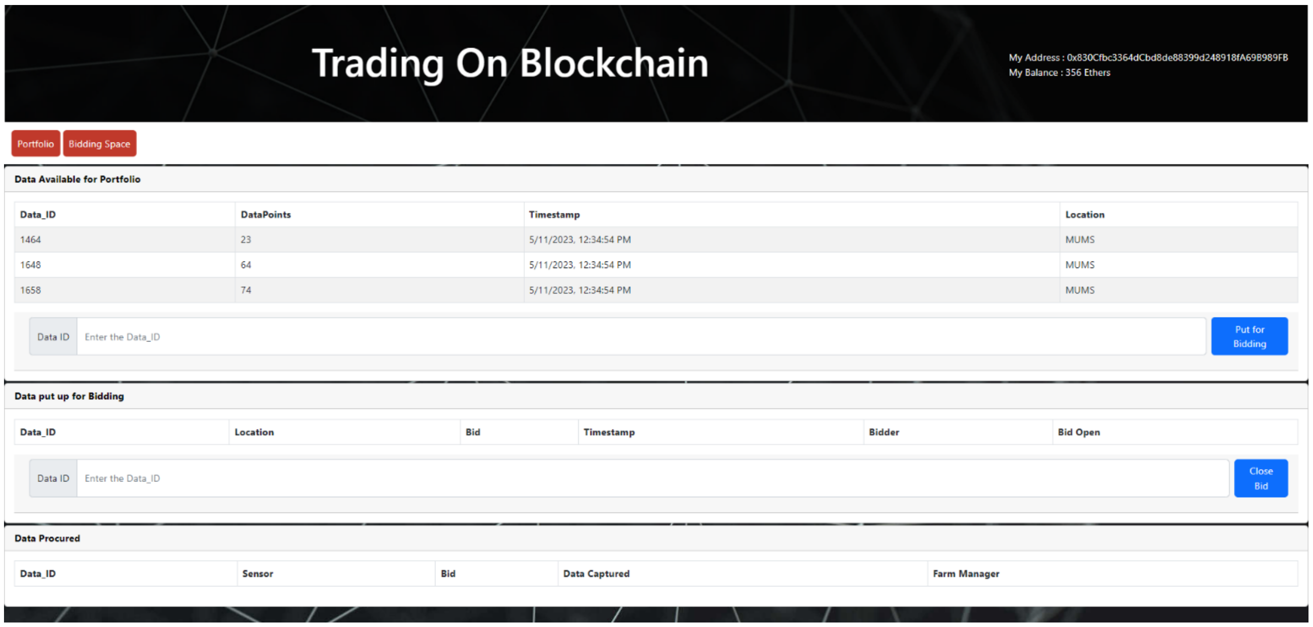
Fig. 19. Deploy the local smart contract



Set up your Sensors!

Sensor Node 0xb6fd605e19c69b531db 8042

Fig. 20. Enter details about the sensor


Trading On Blockchain

My Address : 0x830Cfbc3364dCbd8de88399d248918fA69B989FB
My Balance : 356 Ethers

Data Available for Portfolio

Data_ID	DataPoints	Timestamp	Location
1464	23	5/11/2023, 12:34:54 PM	MUMS
1648	64	5/11/2023, 12:34:54 PM	MUMS
1658	74	5/11/2023, 12:34:54 PM	MUMS

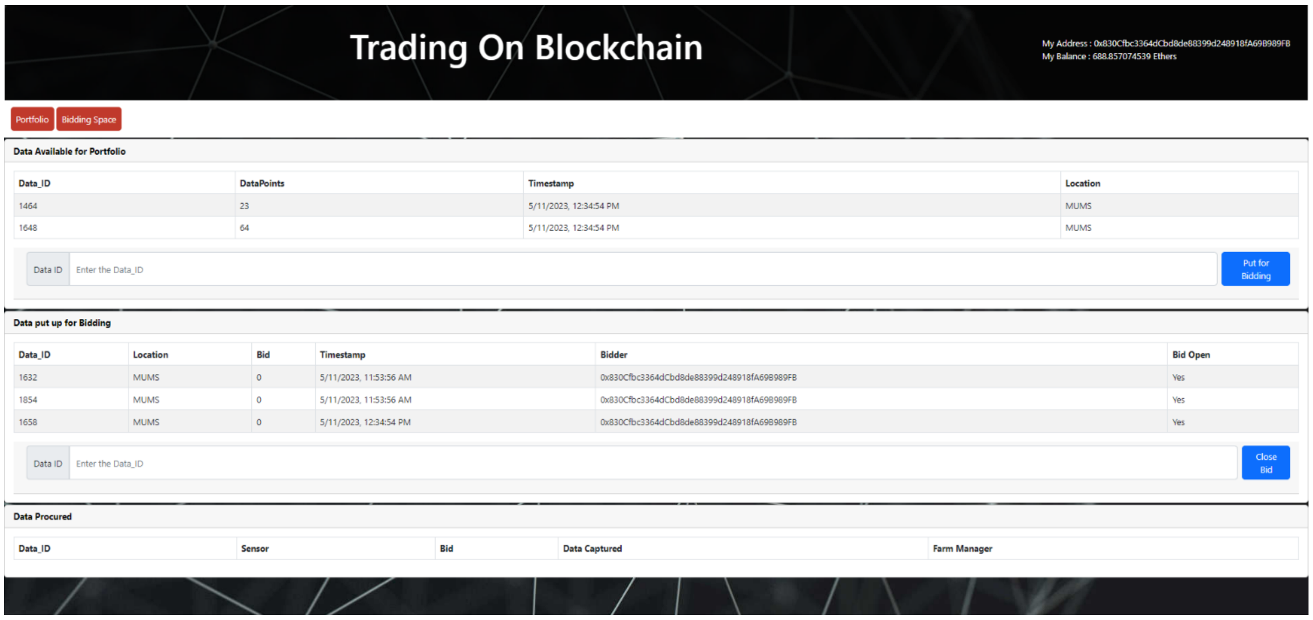
Data put up for Bidding

Data_ID	Location	Bid	Timestamp	Bidder	Bid Open

Data Procured

Data_ID	Sensor	Bid	Data Captured	Farm Manager

Fig. 21. Portfolio page


Trading On Blockchain

My Address : 0x830Cfbc3364dCbd8de88399d248918fA69B989FB
My Balance : 688.857074539 Ethers

Data Available for Portfolio

Data_ID	DataPoints	Timestamp	Location
1464	23	5/11/2023, 12:34:54 PM	MUMS
1648	64	5/11/2023, 12:34:54 PM	MUMS

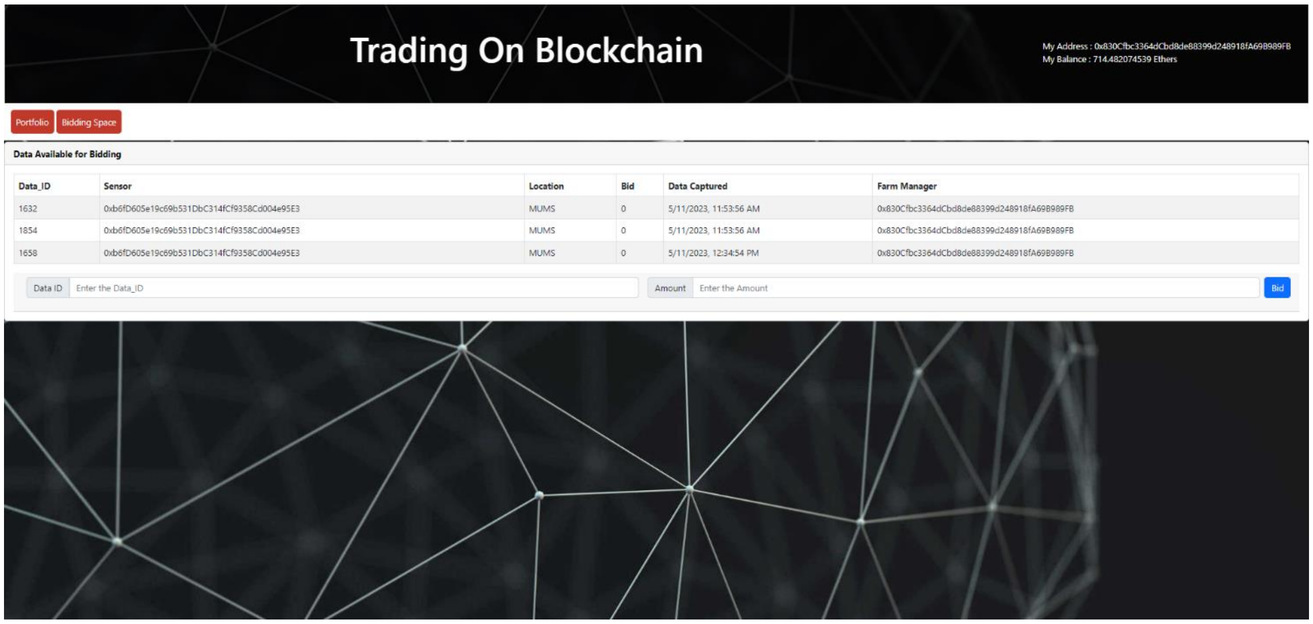
Data put up for Bidding

Data_ID	Location	Bid	Timestamp	Bidder	Bid Open
1632	MUMS	0	5/11/2023, 11:53:56 AM	0x830Cfbc3364dCbd8de88399d248918fA69B989FB	Yes
1854	MUMS	0	5/11/2023, 11:53:56 AM	0x830Cfbc3364dCbd8de88399d248918fA69B989FB	Yes
1658	MUMS	0	5/11/2023, 12:34:54 PM	0x830Cfbc3364dCbd8de88399d248918fA69B989FB	Yes

Data Procured

Data_ID	Sensor	Bid	Data Captured	Farm Manager

Fig. 22. Portfolio & data put up for bidding



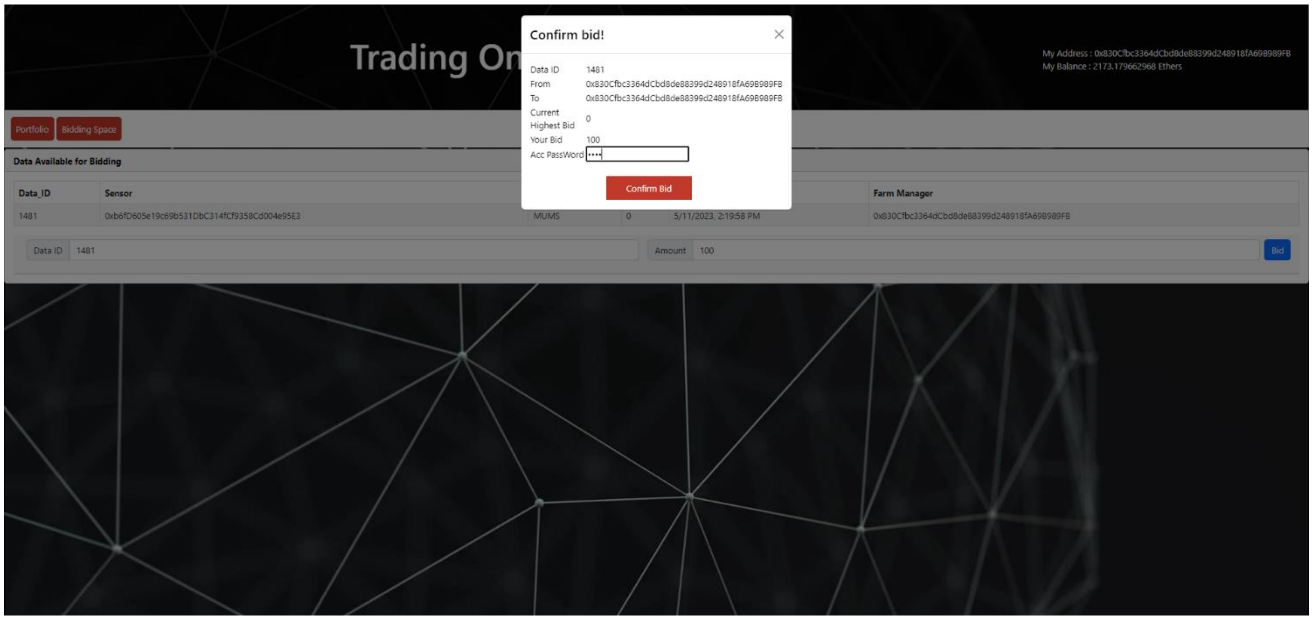
My Address : 0x830Cfbc3364dCbd8de88399d248918fA69B989FB
My Balance : 714.482074539 Ethers

Data Available for Bidding

Data_ID	Sensor	Location	Bid	Data Captured	Farm Manager
1632	0xb6fD605e19c69b531DbC314fCf9358Cd004e95E3	MUMS	0	5/11/2023, 11:53:56 AM	0x830Cfbc3364dCbd8de88399d248918fA69B989FB
1854	0xb6fD605e19c69b531DbC314fCf9358Cd004e95E3	MUMS	0	5/11/2023, 11:53:56 AM	0x830Cfbc3364dCbd8de88399d248918fA69B989FB
1658	0xb6fD605e19c69b531DbC314fCf9358Cd004e95E3	MUMS	0	5/11/2023, 12:34:54 PM	0x830Cfbc3364dCbd8de88399d248918fA69B989FB

Data ID: Enter the Data_ID
Amount: Enter the Amount

Fig. 23. Bidding Space



My Address : 0x830Cfbc3364dCbd8de88399d248918fA69B989FB
My Balance : 2173.179662968 Ethers

Confirm bid!

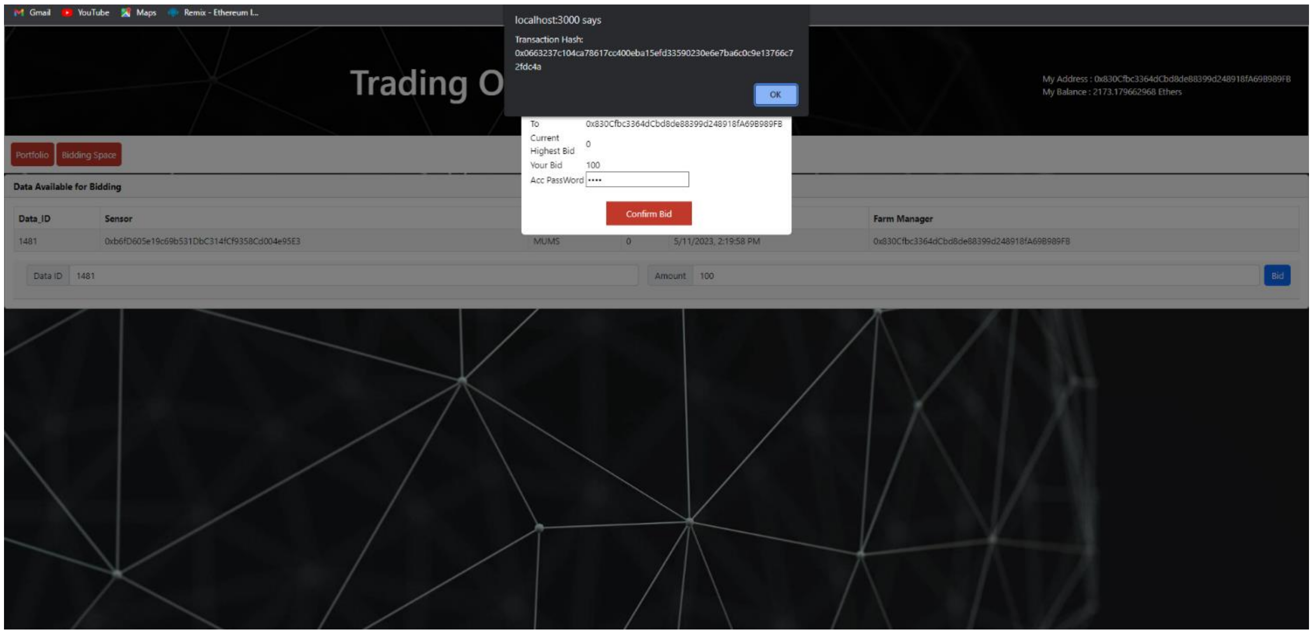
Data ID	1481
From	0x830Cfbc3364dCbd8de88399d248918fA69B989FB
To	0x830Cfbc3364dCbd8de88399d248918fA69B989FB
Current Highest Bid	0
Your Bid	100
Acc PassWord	••••

Data Available for Bidding

Data_ID	Sensor				Farm Manager
1481	0xb6fD605e19c69b531DbC314fCf9358Cd004e95E3	MUMS	0	5/11/2023, 2:19:58 PM	0x830Cfbc3364dCbd8de88399d248918fA69B989FB

Data ID: 1481
Amount: 100

Fig. 24. Confirm bid modal



localhost:3000 says

Transaction Hash:
0x0663237c104ca78617cc400eba15efd33590230e6e7ba6c0c9e13766c72fdc4a

My Address : 0x830Cfbc3364dCbd8de88399d248918fA69B989FB
My Balance : 2173.179662968 Ethers

To	0x830Cfbc3364dCbd8de88399d248918fA69B989FB
Current Highest Bid	0
Your Bid	100
Acc PassWord	••••

Data Available for Bidding

Data_ID	Sensor				Farm Manager
1481	0xb6fD605e19c69b531DbC314fCf9358Cd004e95E3	MUMS	0	5/11/2023, 2:19:58 PM	0x830Cfbc3364dCbd8de88399d248918fA69B989FB

Data ID 1481 Amount 100

Fig. 25. Successful bid transaction hash

Chapter 5

CONCLUSION AND FUTURE SCOPE

This chapter presents the conclusion and future scope of the implemented project.

5.1 Limitation

The implementation of a blockchain-based (IoT) data marketplace system also has several limitations that should be considered. One significant limitation is scalability, where the throughput and storage space of the blockchain used can become a significant problem in the long run since the data is stored on-chain. Another limitation is the initial deployment costs required to deploy various smart contracts for the blockchain-based (IoT) data marketplace, which can be a barrier to entry for smaller organizations or individuals. Moreover, the smart contracts programmed in Solidity do not offer support for floating-point numbers, which may affect the system's ability to use certain types of data. The implementation of a private Ethereum network and smart contracts requires a significant amount of technical expertise, which may limit the ability of non-technical users to use the system effectively. Additionally, the system may not be compatible with all types of sensor nodes, which could limit its utility in certain contexts. Therefore, it is essential to consider these limitations when implementing a blockchain-based (IoT) data marketplace system.

5.2 Conclusion

We successfully implemented a blockchain-based IoT data marketplace that achieved all of its intended objectives. The system provides a secure end-to-end solution for transactions between buyers and sellers of data, ensuring authenticity, integrity, and confidentiality. The intuitive interface allows users to view and manage their data according to their preferences, providing a user-friendly experience. By setting up a private blockchain network and maintaining a distributed ledger, we eliminated the need for third-party brokers, resulting in a cost-efficient solution for the participants.

5.3 Future scope of the project

As a future scope of the project, there are several areas that can be improved and expanded upon. Firstly, the data marketplace can be extended to include other types of data, such as financial or healthcare data, in addition to IoT data. This would increase the value and usability of the platform. Furthermore, incorporating decentralized storage protocols such as IPFS– which is a distributed and decentralized storage protocol or Swarm–a decentralized storage and content distribution platform would enable us to store large amounts of data off-chain, resulting in faster transactions and reduced costs. This would ensure that our network does not slow down as the platform scales up. In addition to technical improvements, it is also important to focus on improving the user experience. This could be achieved through automation of processes for different sensors, and collecting data from the website for analytics. By doing so, we can make the platform more user-friendly and efficient. These future developments will enhance the functionality and value of the platform, making it more versatile and accessible for users.

Bibliography

[1] Ahmed Suliman, Zainab Husain, Menatallah Abououf, Mansoor Alblooshi, Khaled Salah, “Monetization of IOT data using smart contracts”, 2018 The Institution of Engineering and Technology (IET) Journal.

[2] Muhammad Salek Ali, Koustabh Dolui, Fabio Antonelli, “IoT Data Privacy via Blockchains and IPFS”, 2017 Proceedings of the Seventh International Conference on the Internet of Things.

[3] Zhiqing Huang , Xiongye Su, Yanxin Zhang, Changxue Shi, Hanchen Zhang, Luyang Xie, “A Decentralized Solution for IoT Data Trusted Exchange Based-on Blockchain”, 2017 3rd IEEE International Conference on Computer and Communications.

[4] Kazım Rıfat Özyılmaz, Mehmet Dogan, Arda Yurdakul, “IDMoB: IoT Data Marketplace on Blockchain”, 2018 Crypto Valley Conference on Blockchain Technology.

[5] Paulo Valente Klaine, Lei Zhang, Muhammad Ali Imran, “An Implementation of a Blockchain-based Data Marketplace using Geth”, 2021 3rd Conference on Blockchain Research & Applications for Innovative Networks and Services.

[6] Georgios Papadodimas, Georgios Palaiokrasas, Antonios Litke, Theodora Varvarigou, “Implementation of smart contracts for blockchain based IoT applications”, 2018 9th International Conference on the Network of the Future (NOF).

[7] Seyoung Huh, Sangrae Cho, Soohyung Kim, “Managing IoT Devices using Blockchain Platform”, 2017 19th International Conference on Advanced Communication Technology (ICACT)

[8] Wiem Badreddine, Kaiwen Zhang, Chamseddine Talhi, “Monetization using Blockchains

for IoT Data Marketplace”, 2020 IEEE International Conference on Blockchain and Cryptocurrency (ICBC).

[9] R. Vishnu Prasad, Ram Dantu, Aditya Paul, Paula Mears, “A Decentralized Marketplace Application on The Ethereum Blockchain ”, 2018 IEEE 4th International Conference on Collaboration and Internet Computing.

[10] Shaimaa Bajoudah, Changyu Dong1 and Paolo Missier, “Toward a Decentralized, Trust-less Marketplace for Brokered IoT Data Trading using Blockchain”, 2019 IEEE International Conference on Blockchain.

[11] Matthias Knapp, Thomas Greiner, Xinyi Yang, “Pay-per-use Sensor Data Exchange between IoT Devices by Blockchain and Smart Contract based Data and Encryption Key Management”, 2020 International Conference on Omni-layer Intelligent Systems”.

[12] Haya R. Hasan, Khaled Salah, Ibrar Yaqoob, Raja Jayaraman, Sasa Pesic, Mohammed Omar, “Trustworthy IoT Data Streaming using Blockchain and Ipfs”, 2022 IEEE Access.

Acknowledgements

We sincerely thank Prof. Shilpa Vatkar, our project guide, for her valuable guidance, encouragement and support in carrying out this project. We take this opportunity to thank all our faculties who have directly or indirectly helped in our project. We are also very thankful to our HOD of the Electronics and Telecommunications Engineering department, Dr. Bharati Singh, for encouraging and allowing us to present the project on the topic "Hierarchical Distributed Ledger for Ensuring Reliable and Authentic data Collection in IoT with a Peer to Peer Ethereum Blockchain Network". We deeply express our gratitude to Dr. Shubha Pandit, Principal, KJSCE, Vidyavihar, to provide us with this opportunity to learn and explore our technical knowledge.