August 2019

Managing IOT Data on Hyperledger Blockchain

Akhil David
akhildavid26@hotmail.com

MANAGING IOT DATA ON HYPERLEDGER BLOCKCHAIN

By

Akhil David

Bachelor of Technology (Information Technology)
Kurukshetra University
2016

A thesis submitted in partial fulfillment
of the requirements for the

Master of Science in Computer Science

Department of Computer Science
Howard R. Hughes College of Engineering
The Graduate College

University of Nevada, Las Vegas
August 2019

UNLV | GRADUATE COLLEGE

Thesis Approval

The Graduate College
The University of Nevada, Las Vegas

June 13, 2019

This thesis prepared by

Akhil David

entitled

Managing IOT Data on Hyperledger Blockchain

is approved in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science
Department of Computer Science

Yoohwan Kim, Ph.D.
Examination Committee Chair

Laxmi Gewali, Ph.D.
Examination Committee Member

Ju-Yeon Jo, Ph.D.
Examination Committee Member

Yingtao Jiang, Ph.D.
Graduate College Faculty Representative

Kathryn Hausbeck Korgan, Ph.D.
Graduate College Dean

Abstract

Blockchain is a rapidly evolving technology known for its security, immutability and decentralized nature. At its heart, it's used for storing various kinds of data like transactions. But it is not limited to just the transactions or the cryptocurrency. It can also be used to store many other things like assets, IoT data or even multimedia data like songs, pictures, and videos.

The number of IoT devices being connected to the internet is increasing day by day. In fact, Garter (Analyst Firm) predicts there will be 20.4 Billion IoT devices by the end of 2020 [IOTb]. With the increase in the number of IoT devices, there will be an increase in the amount of data they generate. Managing this huge data efficiently so that its available to every authorized user without any integrity loss will be very pivotal in the near future.

HyperLedger is an open source project hosted by Linux Foundation. There are a lot of sub-projects that come under the umbrella of HyperLedger consortia like HyperLedger Fabric, Indy, Composer and many more. HyperLedger Fabric is one of the projects initially developed by IBM and later contributed to HyperLedger. It allows us to develop private permissioned Blockchain following the best in industry standards and algorithms.

In this thesis, we are managing IoT data on the HyperLedger Fabric Blockchain. We will be collecting data from the IoT sensors and securely transmitting it to our node running the HyperLedger Blockchain using the MQTT protocol. After receiving data from the sensor we will process the data and add it to our ledger. We also evaluate the performance of our network taking various parameters like batch timeout, batch size, and message count into consideration.

Acknowledgements

The success of any project is never limited to the individual who is undertaking the work. Most often, it is the collective effort of the people that reflect in the success. First and foremost, I want to express my sincere thanks to my advisor Dr. Yoohwan Kim. I highly appreciate his able guidance, inspiration and encouragement throughout this research and Masters program. It was enriching experience to work with Dr. Yoohwan Kim. I highly appreciate his ideas and expertise in carrying and helping out in my research.

I would also like to extend my thanks to Dr. Ju-Yeon Jo, Dr. Laxmi Gewali, and Dr. Yingtao Jiang for theirsupport and for being a part of my thesis committee. My special thanks to Dr. Datta for his direction in the Masters program and help to achieve my goals. Thank you for your support and indelible help. I feel great joy in expressing thanks to all the professors and staff of Computer Science department,for their support, knowledge, encouragement and attention during my time at UNLV. I also gratefully acknowledge the funding support I got from the Computer Science department, UNLV to pursue my Masters degree.

Words alone cannot describe the immense gratitude that I owe to my parents for their love, deep affection, encouragement and blessings. Cheerful company and constant support of my sister Aditi David has gone a long way to help me during my studies. My Special thanks are due for my friends Callie, Shradha, Sanika, Vinay, Suresh, Aniket and many more. I cherish with appreciation their joyful company for making me comfortable during the study period and cheering me in the moments of despair. Thank you very much, I am glad to have you all in my life. I would also like to specially thank my all my colleagues for their help and support during the research work.

Akhil David

University of Nevada, Las Vegas

August 2019

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 What is Blockchain

Blockchain, as the name suggests, can be visualized as a chain of blocks linked together. What is there in blocks could be any type of data ranging from text, pictures, to audio and video as well. But what makes Blockchain different and unique is its immutable and distributed nature.

There are many nodes in a Blockchain network with each node having the same copy of the data known as a ledger. A ledger has a unique property of being immutable that is once a data is written to the ledger it would be there forever.

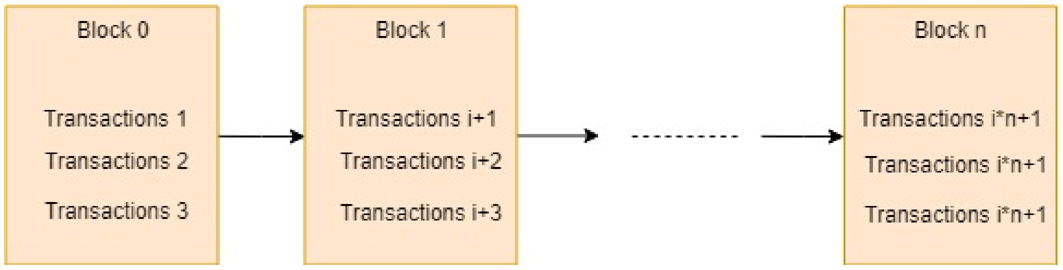


Figure 1.1: Blockchain: Blocks linked in the form of a chain

1.2 History

When Bitcoin: A Peer to Peer Electronic Cash System, which defined a "purely peer-to-peer version of electronic money", was published by Satoshi Nakamoto in 2008, Blockchain technology made its debut.

Over the past decade, Blockchain, the technology running Bitcoin[SMC$^+$17], has evolved into one of the world's biggest technological breakthrough with the potential to impact all industries from financial to supply chains to fabricating industries to educational establishments.

Blockchain has gone through the following stages:

- Beginnings with Bitcoin
- Blockchain Separates from Bitcoin
- Ethereum Rises: Smart Contracts
- Progress towards Proof of stake as alternative
- Scaling of the Blockchain on the horizon

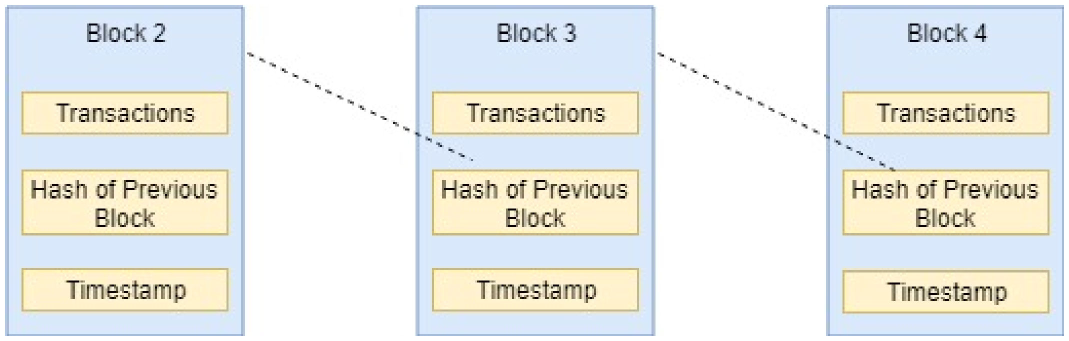


Figure 1.2: Blockchain: Previous Block Hash is part of New Block

1.3 Types of Blockchain

Blockchain can be categorized into two types:

- Public Blockchains

  They allow anyone to be involved as users, miners, developers, or members of the community.[HCD$^+$19] Public Blockchains are designed to be completely decentralized, with no one controlling which transactions is recorded in the Blockchain or the order in which they are processed.

- Private Blockchains

  Private Blockchains, also known as permissioned Blockchains [Vuk17]. Participants need consent to join the networks. Transactions are private and only accessible to participants in the ecosystem who have been authorized to join the network

1.4 Comparison with other Blockchains

1.4.1 HyperLedger Comparison with Ethereum

HyperLedger and Ethereum, are both non-proprietary. They can be used for developing industrial Blockchain networks. Both are a platform for attracting developers to develop Blockchain based tools and frameworks in a collaborative manner

But there are some differences as well. Ethereum was initially developed by Vitalik Buterin whereas HyperLedger since its beginning was associated with Linux Foundation and has received support from many companies like Intel, IBM, and many others [VS17]. Since Ethereum did not have any major financial backing, it introduced the concept of Initial coin offering (ICO) to raise the necessary funds required for its global and collaborative development.

HyperLedger didn't have any financial problem since its inception. Since HyperLedger was introduced half a year after Ethereum, it estimated the issues that crypto-currencies may cause and decided to stay from it. As was stated by the Executive Director of HyperLedger project "You will never see a HyperLedger Coin" - Brian Behlendorf

Not only HyperLedger avoids the use of any coin for fundraising but also its framework is highly inefficient for it. Though you can create a coin using HyperLedger Framework, it is going to be a lot more complex than doing it with Ethereum.

Ethereum focus is on public domain that is B2C applications while HyperLedger Focus is on private domain B2B applications. The difference between Ethereum and HyperLedger Fabric is summarized in the table 1.1

1.4.2 HyperLedger Comparison with R3 Corda

R3 Corda was one of the first players to catch the Distributed Ledger Technology's potential. While the rest of the world was focused on Bitcoin, R3 focused on the private aspect of Blockchain R3 Corda was one of the first players to catch the Distributed Ledger Technology's potential[BCGH16]. While the rest of the world was focused on Bitcoin, R3 focused on the private aspect of Blockchain. It is used for storing and managing financial agreements between organizations.

Just like HyperLedger Corda also has a consortium of organizations. Also, like HyperLedger R3 Corda is a private permissioned Blockchain but the difference is that R3 Corda allows sharing of data transactional data only between the parties involved in the transaction. Also, there are no blocks of data like bitcoin, Ethereum and HyperLedger instead the transaction data is sent only to

Table 1.1: Ethereum vs HyperLedger

Property	**Ethereum**	**HyperLedger Fabric**
Managing Entity	Ethereum Developers	Linux Foundation
Platform	Generic Platform for Blockchain	Modular Blockchain Platform
Consensus	POW, POS	Pluggaable Consensus like Kafka, Raft
Consensus Level	Ledger Level	Transction Level
Currency	Ether	None, possible by smart contract coding

parties involved in the transaction. It uses notary services to provide transaction ordering. It uses a vault for storing ledger data which contains information relevant to the node's owner[KLR$^+$17]. Vault also stores private key required to sign transactions

1.5 Objective

IoT and Blockchain are two rapidly evolving technologies which have the potential of affecting our lives on a daily basis in coming times. In this thesis, we want to implement a Blockchain storage solution for IoT data. We are using HyperLedger fabric as our Blockchain technology which is a private permissioned Blockchain. Our objective is to show that HyperLedger fabric can be used for managing the IoT data and we also evaluate the performance of our deployed network taking various parameters into consideration.

1.6 Outline

In chapter 2 we have discussed the motivation for carrying out this thesis, various challenges, and benefits for using IoT with Blockchain. In chapter 3, we introduce HyperLedger and HyperLedger

Table 1.2: HyperLedger vs R3 Corda

Property	**HyperLedger Fabric**	**R3 Corda**
Use Case	used for B2B applications	Application in Financial Industry
Management	Linux Foundation	R3 Company
Consensus	Only Orderers decide the order of blocks	Only the entities involved in transaction take part in decision making
Language	Write in Go	Written in Kotlin
Mode	Private Permissioned	Private Permissioned
Currency	None, Possible via chaincode	None

Fabric. In chapter 4, we introduce the IoT system that we will be using. In chapter 5, we propose our solution with the architectures we will be using. In chapter 6, we discuss the implementation details of the Network. In chapter 7, we show our performance analysis. In chapter 8 we talk about our future work

Chapter 2

Motivation

2.1 IoT data management challenge

IOT is a fairly new and rapidly evolving concept. Each organization may have a unique approach to it but the basic idea is pretty simple and can be categorized as

- Collect Data
- Analyze the data
- Generate results from analyzed data
- Use the generated results to modify the system
- Finally realize the benefits of improved system
- Repeat the above steps

This seems easy in theory, but IOT deployments are challenging. Some of the challenges are as follows

- Hardware Compatibility Issues: Most of the IOT data is captured using various sensors, which is transmitted to cloud or servers using the IOT gateways. If the organization's current physical devices and machines are not identified and understood then there could be compatibility issues with the sensors employed to collect IOT data
- Data Connectivity Issues It involves how the IoT devices talk to gateway or server/cloud. Determining the right set of protocols and data format for sending data to the IoT platform is necessary.

2.2 Benefits and challenges of storing IoT data on Blockchain

2.2.1 IoT challenges solved by Blockchain

- Security

  Security of IoT devices and data in one of the key issues in the mind of IoT adopters. Hackers have managed to gain access to IoT devices like gaining access to cars remotely or Personal refrigerator. This may not sound much but consider the IoT devices used in the medical industry. If a hacker gain access to implanted IoT device the danger could be life-threatening. These issues make the use of Blockchain for IoT data management a suitable option. Blockchain uses the best is industry encryption standard which may protect IoT data from any malicious user.

- Privacy

  Private Blockchain only allows authorized entities to update, add or access to the ledger data. Only those with the authorized keys can access the data. Only authorized IoT devices can add data to the Blockchain. Once the data is recorded on the ledger it can't be changed this makes sure that no one can corrupt the data once it has been recorded.

2.2.2 Challenges for storing IoT data on Blockchain

- Latency

  Once the IoT device records the data on a node, new update transactions are sent to all the nodes in the ledger. If there are a large number of nodes in the network then it can take some time for the recorded data to propagate to all nodes in the network. Since most of the IoT devices generate data around the clock. This can lead to some serious propagation delays

- Scalability

  The number of IoT has been increasing rapidly. We deal with IoT devices on a daily basis without even realizing it. Smartphones, smartphones, fitness trackers are some of the IoT devices we use on a daily basis. As the number of devices increases the number of transactions on the ledger will increase [ZXD+18]. Blockchain at present does not have the potential to handle millions of transactions per second. This leads to a scalability issue for storing IoT data on Blockchain.

– Storage

  As more and more IoT devices come online they generate more and more data. Also, most of the IoT devices generate data every second. Since all the nodes in the Blockchain network store the whole copy of the ledger, it leads to huge storage concern. Storing IoT data on all nodes in the network leads to highly inefficient use of storage and may not be economically feasible for a network having many IoT devices.

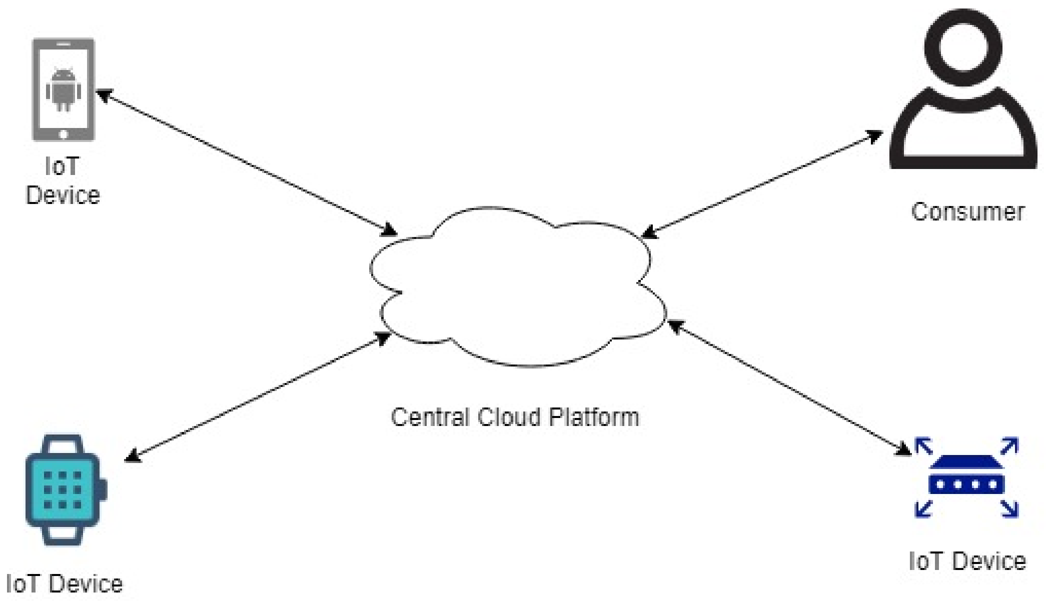


Figure 2.1: Centralized IoT System

2.3 Previous research

Over the past few years, Blockchain technology has observed an outburst of research and development pursuit, mainly within the FinTech industry. The ability to provide immutable records, along with their potential to enable confidence and trust among doubting or untrusted peers, are too attractive features to prevent this technology from its application to various other sectors.

There is a lot of interest from industry in combining IoT and Blockchain. Some companies have already started to explore their potential. Helium, [IOTa] San Francisco, California is one such company who is working on connecting low power IoT devices to the internet using Blockchain.

In recent years, the implementation of certain IoT devices and techniques in the supply chain management industry has also drawn a great deal of study interest. The authors of [SSA17] presented a use-case for inventory transparency specifically for the Agri-Food domain, also adopting some IoT devices. The aim was to investigate the use of RFID and NFC-based devices to accomplish direct on - the-field transparency and real-time information production, allowing for persistence through a cloud-based database which is not decentralized. This is indeed the classical paradigm most of the current IoT-based solutions have adopted by far. The use of IoT and Blockchain technologies, however,is still a research field that is under-explored but worth exploring.

2.4 Proposed idea

In this thesis, we are using HyperLedger Fabric Blockchain for storing IoT data. As HyperLedger is a private permission Blockchain we can control who will have access to our IoT data. HyperLedger allows access only to authorized parties as defined in this Policy. HyperLedger also uses state of the art encryption standards for its various transactions. This makes the HyperLedger Fabric Blockchain Highly secured from any malicious activity.

HyperLedger utilizes X.509 certificates for identity management. Therefore we can be sure only authorized entity is able to add data to our ledger. HyperLedger can allow read-write access to various entities depending on the confidence we have in that entity. This allows us to control the privacy of our IoT data.

We will be using DHT 11 sensor as our IoT devices. DHT 11 are the sensors which give us the Temperature and Humidity data of the surrounding environment. We would connect our sensors to the raspberry pi, which would allow us to record the data digitally. Our proposed idea is to record this IoT data securely on the ledger. We will use the MQTT protocol for securely sending our data to our HyperLedger Node.

We will configure a HyperLedger Network which will consist of two organizations and four peers. Each organization will have two peers each. Our network will also contain one Orderer which will be used for ordering our IoT transactions.

We will also do the performance evaluation of our network using various parameters like batch timeout, batch size and number of messages into consideration.

Chapter 3

Introduction to HyperLedger and HyperLedger Fabric

3.1 Background in HyperLedger

HyperLedger cultivates and promotes many Blockchain technologies related to the business aspect, which include smart contract engines, distributed ledger frameworks, client libraries, graphical interfaces, utility libraries, and sample applications. The HyperLedger umbrella design inspires the re-use of common building blocks and enables rapid development and innovation of Distributed ledger technology components.

As of 2019 there are six HyperLedger Frameworks. Out of which only two has active status and rest are in

1. Burrow
2. Fabric
3. Grid
4. Iroha
5. Indy
6. Sawtooth

HyperLedger project has six tools as follows.

- Caliper

- Cello
- Composer
- Explorer
- Quilt
- Ursa

3.1.1 HyperLedger Burrow

Status - Incubation (2019)
HyperLedger Burrow is one of the projects that come under the HyperLedger umbrella and is hosted by Linux Foundation. Monax contributed the burrow to the HyperLedger and was cosponsored by Intel. It provides a client module for running smart contract in machines based on EVM specification.[SS18] It uses Byzantine fault-tolerant Tendermint as its consensus algorithm. It has got the following components namely.

- consensus engine
  Using the Byzantine fault-tolerant Tendermint protocol for consensus over a set of known validators provides high throughput for transactions and prevents any forking
- Application Blockchain Interface (ABCI)
  ABCI is the interface between the conensus engine and the smart contract application.
- Smart Contract Application
  The smart contract code is included in the Burrow accounts. When the smart contract code is called to execute inside a burrow account it will execute inside the permissioned EVM.
- Permissioned EVM
  It provides a virtual environment to execute a smart contract according to ethereum specification. A random amount of gas is provided for the smart contact execution and prevent infinite loops
- Application Binary Interface (ABI)
  It is used for compiling, deploying and linking of smart contracts. It is also used to formulate transactions to call other smart contracts.

3.1.2 Fabric

Status - Active (As of 2019)

Fabric is also one if the open source venture under the HyperLedger consortium. It is used to provide enterprise-level Blockchain solutions.[YNZ$^+$19] It supports a modular architecture so can be configured according to the requirement.

HyperLedger Fabric provides the following functionalities [Hyp19].

1. Identity Management Every operation in HyperLedger is to be signed by certificates of various entities. HyperLedger provides a Certificate authority and Membership service Provides (MSP) for managing the Identities.

2. Privacy and Confidentiality An exclusive HyperLedger feature is the channel. It allows private path communication among certain members in a Blockchain network.

3. Efficient Processing In HyperLedger not every node is equal. Some execute the smart contract code while some order the transaction and while some just keep the copy of the ledger. Allowing the distribution of work among nodes provides a huge performance boost.

4. Chaincode functionality Chaincode contains the logic the transactions need to follow. All transactions submitted to a specific chaincode in a channel follow the same set of rules as specified in the chaincode.

5. Modular Design HyperLedger provides a modular design which can be configured as required. Desired Ordering, encryption and Identity algorithm can be plugged in fabric network. This allows for more adoption as each organization can configure it according to its needs.

3.1.3 Grid

Status - Incubation HyperLedger grid is a set of tools for providing supply chain management solutions. It is neither a Blockchain nor an application. It is a framework which provides best industry standards, libraries and SDKs for the supply chain in modular design. HyperLedger Grid includes the following components.

- SDKs that make it easier to write smart contract for supply chain solutions.

- Using best industry practices to provide smart contract business logic

- Smart contact for managing identity using Sawtooth.
- Implementations for supply chain specific data types.

3.1.4 Iroha

Status - Active
HyperLedger Iroha is another project hosted under Linux Foundation. Initially, Iroha was an open source project of a company in Japan named Soramitsu[SS18]. It was later contributed to HyperLedger by organizations like Soramitsu, Hitachi, NTT Data and Colu [Iro]. It is written in C++ and uses the Byzantine Fault tolerant (BFT) algorithm for consensus. It allows applications to be written in C++, Java, JavaScript, Python, Android, and IOS. The distinction between Iroha and other Blockchain is that only one out of every odd member has all the record information or ledger data. It is basically a permissioned Blockchain used for managing Identity, Serialized data, and assets.

3.1.5 Indy

Status - Active HyperLedger indy is used for managing identities using decentralized ledger technology. This allows organizations from a different domain to use and verify each other identities. It provides various tools and identities for managing identities. It uses the sovrin protocol. It allows users to manage their privacy. It uses the RBFT as its consensus protocol[AMQ13]. It is inspired by plentum BFT which makes the system fault tolerant and allows us to detect malicious node

3.1.6 Sawtooth

Status - Active It provides a modular approach for building a decentralized solution for various businesses. It provides different SDKs in different language like GO, Java, JavaScript, C++, Python, and rust. It allows us to write a smart contract in any language of our choice. It is one of the only Blockchains which allow [parallel execution of transactions. Seth Sawtooth-Ethereum integration project allows running of Ethereum smart contracts on Sawtooth

The various consensus supported in sawtooth are as follows

- RAFT It is a leader-follower based protocol. A leader is chosen from voting for a certain amount of time. if the leader goes down a new leader is elected and the process continues. It provides high throughput and fault tolerance and low latency

- Dev mode - It chooses a random leader and is used for testing
- Proof of elapsed time simulator: It gives agreement or consensus on any type of machine regardless of the equipment or underlying hardware we are utilizing.
- Proof of elapsed time protocol: It was designed by Intel and is exclusive to Intel CPU's. It solves the Byzantine general problem. It provides a quick consensus for very large networks.

3.2 Identity management

3.2.1 Introduction

There are various entities in Blockchain network like peer, orderers, applications, administrators. Each of these entities needs to have a valid identity in the network. In HyperLedger each user has an X.509 certificate which serves as a valid identity for each entity. These certificates play a key role in determining the permission access to various resources in the network. A certificate is as trustworthy as the genuineness of the authority it is coming from. A Membership Service Provider (MSP) is used for specifying the rules which identify the valid identities for the various entities.

3.2.2 Public Key Infrastructure PKI

PKI is used for providing secure communications in a network. It makes sure each entity in a network is valid. By default, fabric implements MSP which uses PKI (Public key infrastructure) and X.509 certificates for the entities. A PKI consists of the following elements:

1. Certificate Authorities
2. Public and Private Keys
3. Certificate Revocation Lists
4. Digital Certificates

3.2.3 Digital Certificates

A digital certificate could be seen as analogous to a person's ID/Driving License card. It contains a set of attributes which helps us identify the owner of the digital certificate. A digital certificate also includes the public key of the entity as well along with other relevant information whereas the private key is kept secret and never included in the digital certificate. All this information is encoded using the cryptography. Modifying or tampering of the details in the certificate will make the certificate invalid. In HyperLedger we use the X.509 standard for a digital certificate. As long as Certificate Authority CA 's identity is not compromised (i.e its private key is kept secret) we can be sure that the certificate of the entity has not been tampered.

3.2.4 Public Keys and Private keys

The public key is the key which entity distributes along with the digital certificate. The private key is kept secret. The relationship between public and private key is such that a message signed by the entity's private key can be verified by the entity's public key. The message signing using private key also ensures the message integrity as well as message authentication.

3.2.5 Certificate Authority

Certificate Authority widely distributes certificates to various entities. All the nodes in the Blockchain need to have a valid digital identity which may be distributed by one or more CA. The CA keep its private key as well as the entity's private key secret. CA makes its certificate publicly available which includes its public key. This makes entity to verify to each others certificate as a certificate signed by CA's private key can be verified by CA's certificate which includes its public key

3.2.6 PKI Vs MSP

A PKI provides a list of valid identities in the network. An MSP, in contrast, tells us what identities out of the valid identities provided by PKI forms a part of the Blockchain network. In other words, MSP is the one controlling the valid identities in the Blockchain network.

3.3 Hyperledger Blockchain configuration

The HyperLedeger configuration is the heart of the Blockchian network. We need to configure various entities peers, organizations, policies, channels and a lot more. Most of the HyperLedger configurations goes into the yml or yaml files.
HyperLedger provides the following binaries for generarting various types of configurations:

- configtxgen This toll is used for generating the first block of the ledger also known as the genesis block. This block is necessary bootstrapping the orderer as well as the channel configutrations.

- cryptogen It is used for generating the crypto material for various entities of the organization.

3.4 Use of dockers and network architecture

We use docker containers[KV17] for configuring various entities in the Blockchain Network. For our Blockchain network we have containers for the following entities:

- Peer containers

- Fabric CA (Certificate Authority) containers

- cli container to access all the peers in the network.

- Orderer container

- couchdb containers

All these containers are a part of a single docker network. Even the chaincode is executed in a separate container from the endorsing peer process.

3.5 Accessing CouchDB and query results

HyperLedger provides a docker image of CouchDB that should run on the same server as that of the peer. We need a CouchDB container per each running peer. CouchDB is an optional alternative to levelDB as state database which supports rich queries as it allows chaincode values to be modeled as JSON. It allows us to query actual data content rather than just keys. We can't switch a peer using levelDB to use CouchDB due to data compatibility issues.

3.6 Block structure and Blockchain structure

A HyperLedger Blockchain record or ledger comprises of two particular yet related components:

- A world state

- A Blockchain

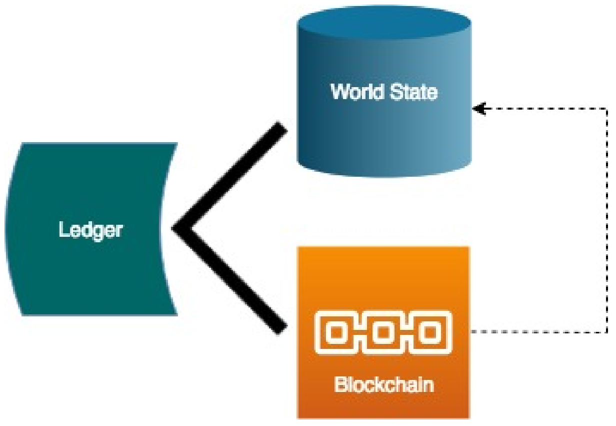


Figure 3.1: Ledger comprises of Blockchain and world state
World state is derived from Blockchain

3.6.1 World State

World state is a database that stores the current values of various states of the Ledger. Current values of these states can be easily read from the World state which avoids traversing the entire transaction log and thus saves time. Record or ledger states are represented as key-value pairs which are updated, created and erased frequently.

3.6.2 Blockchain

It is a log of all the transactions that lead to the world state. These transactions are organized into blocks which are linked together to form the Blockchain enabling us to have a record of all the history that led to the current world state. Unlike world state, it contains immutable blocks which contain a set of ordered transactions.

3.6.3 Blocks

Hyperledger Block composed of three parts:

The Block header H2 of block B2 contains the current blocks data D2's hash, the current block number which is 2, and a copy of previous blocks hash PH1 whose block number is1.

- Block Header:

  Block Header consists of three fields written when creating a block.

  - Block number:

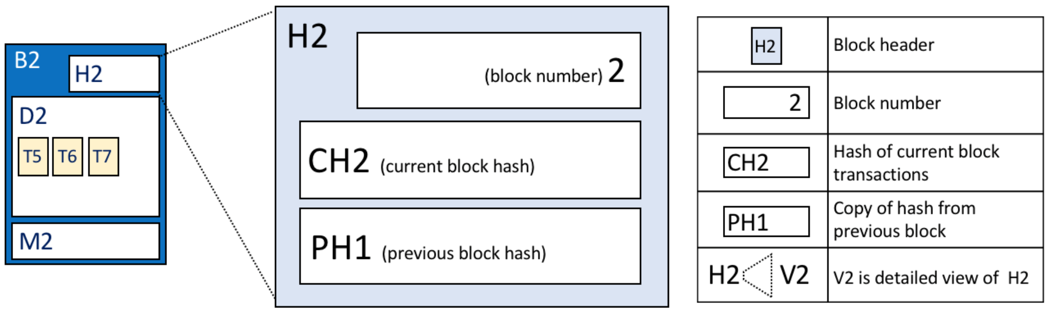

B2
H2
D2
T5
T6
T7
M2
H2
(block number) 2
CH2 (current block hash)
PH1 (previous block hash)
H2
Block header
2
Block number
CH2
Hash of current block transactions
PH1
Copy of hash from previous block
H2
V2
V2 is detailed view of H2

Figure 3.2: HyperLedger Block Structure

Block number is an integer which starts at 0 (For Genesis Block) and is increased by 1 for each new block that is attached to the Blockchain.

- Previous Block Hash:

  This contains the hash value of the previous block in the Blockchain which is essential for maintaining the links between blocks.

- Current Block Hash:

  This contains the hash value of all the transactions that are present in the current block
  .

- Block Metadata:

  Block Metadata contains the time the block was written, public key, signature and the certificate of block writer. Thereafter, the block committer also adds a flag which tells whether a transaction is valid/ invalid. This information about the valid/invalid transaction is not used in the hash as the hash gets created when creating the block and this flag is added when writing to the ledger.

- Block Data:

  This section contains a list of ordered transactions. These transactions are added to block data when the block is created. These transactions have a detailed but simple and understandable structure

3.6.4 Transactions

Changes to the World State are captured by Transactions. The block data structure of the Blocks contains the transactions.

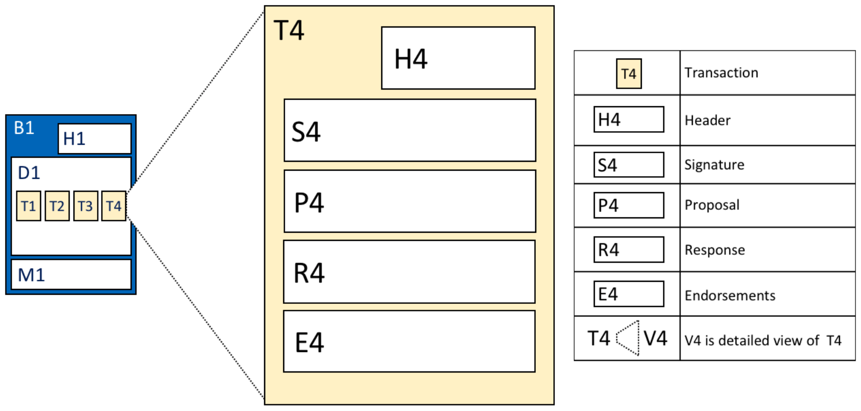


Figure 3.3: Transaction Block

The Transaction T4 inside block data D1 of block B1 contains a transaction signature S4, transaction header H4, a transaction response R4, a transaction proposal P4, and a list of endorsements, E4. Various fields of Blockdata structure containing transactions is as follows:

- Header:

  The header which is represented by H4 which contains important metadata about the transaction for example, the name and version of the appropriate chaincode.

- Proposal:

  The proposal which is represented by P4 which includes the input parameters provided to the chaincode by an application creating the proposed update of the ledger. This proposal provides a set of input parameters when the chaincode runs which, together with the current world state, determines the new world state.

- Endorsements:

  The endorsement is represented in E4, this is a list of each organization's transaction responses which are signed and that are sufficient to meet the endorsement policy. Only one transaction response is included in the transaction although there may be multiple endorsements. This is because each endorsement effectively encodes the particular transaction response from its organization which means that including any transaction response that does not match sufficient endorsement would be a waste of resource as it will get rejected and marked invalid and would not update the world state of the ledger

- Signature:

  This contains a cryptographic signature which is created by the application of the client. It is used to check that the details of the transaction were not modified as it requires the private key of the application to generate it.

- Response:

  This section captures the world state's before and after values as a Read-Write (RW-set) set. It is a chaincode output, and if the transaction is validated successfully, updating the world state will be applied to the ledger.

3.7 Exploring Blockchain

HyperLedger supports two types of state database which consist of CouchDB and levelDB. CouchDB is an optional external state database alternative. LevelDB is the default database of key-value state embedded in the peer process. Both CouchDB and levelDB can store any binary data but CouchDB also supports JSON. This allows running rich queries against the modeled JSON data.

They both also support getting and setting based on keys and running queries based on keys. Querying of keys by range is possible, and composite keys can be modeled against multiple parameters to allow equivalence queries. Using CouchDB and modeling assets as JSON allows us to run complex rich queries against the data values. These types of queries allow us to view and explore the data the Ledger. The response of these queries do not go to the orderer as transactions but are directly sent to the client application. The rich queries are not suitable for update transactions as the result set returned is not stable unless the application can handle the stability between the commit time and Chaincode execution time, or the potential changes that may happen in the subsequent transactions can be handled. CouchDB runs alongside the peer as a separate database process, so there are additional considerations regarding setup, management, and operations.

3.8 Ordering Algorithms

3.8.1 Ordering comparison to other Blockchains

Ordering is defined structuring the transactions that are received and creating a block out of those received transactions. Many of the existing Blockchains like Ethereum and bitcoin use a probabilistic approach for this. In these Blockchain networks, any peer can add data to the network with equal probability. Each peer competes to find a hash collision by trying various values of the nonce to get the desired hash. The node that gets the matched hash, broadcasts its proposed block with the nonce to all the other peers. Though finding the hash utilizes lots of computing power but verifying is almost instant. Once the peer verifies the genuineness of the block, they add it to there local ledger. But there is one issue with this approach it is possible that more than one peer find the hash solution at the same time if that happens than we can have multiple branches in the Blockchain ledger leading to forking.

In contrast, HyperLedger uses a deterministic approach. In HyperLedger we have separation of nodes which run the chaincode (smart contract) known as endorsers and the one which creates the block from the transactions known as orderer. We can have one node as orderer or multiple nodes which form the ordering service. In HyperLedger the consensus is that any block generated by the orderer is considered to be final and valid thus leading to deterministic nature. This deterministic nature helps us prevent ledger forking or branching. This gives HyperLedger a huge performance benefit. Also, separation of the node performing the execution of chain code and the node doing the ordering of the transactions provide a much better scalability and performance advantage over other Blockchains.

3.8.2 Types of ordering available

In HyperLedger we have 3 types of implementations available.

- Solo

  In Solo configuration, we have just one ordering node. Having just one orderer node makes the network lose its advantage of being fault tolerant. But Solo is useful when we have to check proof of concepts and testing our logic for applications. The solo network is not suitable for production and would provide very poor performance if used in the production environment.

- Kafka

  Kafka is a multi-node ordering service implementation provided for HyperLedger. Kafka follows a leader-follower configuration. It is a good option for the production environment. It uses Apache Zookeeper ensemble for managing various nodes. It is therefore scalable and fault-tolerant. Though managing various nodes using Apache Kafka is a tedious task.

- Raft

  Raft is the latest implementation provided in HyperLedger fabric version 1.4.1. Raft is similar to Kafka and follows a leader and follower configuration. In Raft a leader node is chosen and all the other nodes follow it. When the leader goes down a new leader is randomly chosen and all the nodes then follow the newly elected leader. This makes Raft crash and Fault Tolerant. Raft is also easier to configure than Kafka based ordering implementation.

Chapter 4

Introduction to IoT

The basic idea behind the Internet of Things (IoT) is pretty simple, it means all the things/devices in the world and connects them to the internet or network.

4.1 IoT system

The Internet of Things (IoT) system is a system of linked computing devices which can be digital or mechanical machines, or objects, or living beings or people that have unique attribute or identifiers and can transfer data across a network without requiring any human intervention or computer-to-computer interaction [KA13].

Four distinct components are integrated into a complete IoT system: sensors / devices, connectivity, data processing and user interface.

- Sensors/Devices
  First, they collect data from their environment from sensors or devices. This could be as simple as reading the temperature or as complex as a complete video feed. A device may have multiple sensors which measure various kinds of data. Be that as it may, regardless of whether it's an independent sensor or a full gadget, in this initial step information is being gathered from nature by something.

- Connectivity
  Next, the information is sent to the cloud, but some means is needed to achieve that. The sensors/gadgets can be associated with the cloud through an assortment of strategies like the WiFi, cellular, Bluetooth and satellite.

- Data Processing:

  Once the data reaches the cloud, some processing is performed on it by the software. This could be very simple, such as checking that the reading of temperature is within an acceptable range. Or it might also be very complex to identify objects (such as intruders in your home) by using computer vision on video.

- User Interface

  Next, in some way, the information becomes useful to the end-user. This may be through a user alert (email, text, notification, etc.). For instance, a text warning when the temperature in the cold storage of the company is too high. A user may also have an interface to proactively check in the system. For instance, a customer might want the video feeds in their home via an app or an internet browser.

4.2 Hardware

Our Hardware system consist of the following:

- Raspberry pi
- DHT11 Sensor
- System running Linux (Ubuntu)
- Jump wires
- breadboard

DHT11 Working

DHT11 sensor[DHT] consist of the following:

- Thermistor or NTC temperature Sensor
- Humidity Sensing Component
- IC on the sensor back

[DRDJ17] The component that senses humidity has two electrodes and between them is a substratum that has moisture. The conductivity of the substratum or the resistance between these

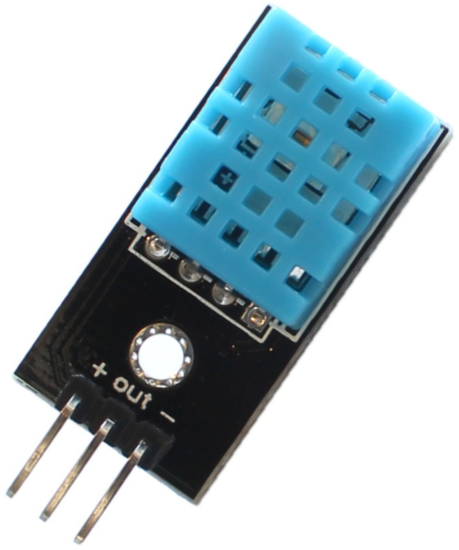


Figure 4.1: DHT11 Sensor Pins

electrodes changes as the humidity changes. The IC measures and processes this change in resistance, making it ready to be read by a microcontroller.

An NTC temperature sensor or a thermistor is used for measuring the temperature. In fact, a thermistor is a variable resistor that changes its resistance with temperature change. semiconductive materials such as polymers or ceramics are used in these sensors which for small changes of temperature provides a large temperature change. NTC means Negative Temperature Coefficient which means with an increase in temperature the resistance decreases.

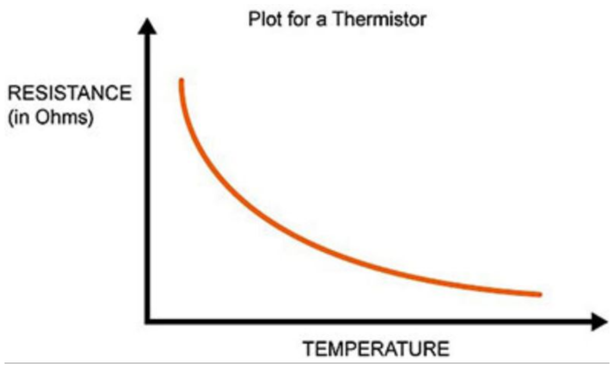


Figure 4.2: Resistance Vs Temperature of Thermistor

Table 4.1: DHT11 Sensor Pins Connections

DHT11 Pin	Signal	Raspberry Pi pin
1	3.3V	1
2	Data/Out	11(GPIO17)
3	not used	-
4	Ground	6 or 9

4.3 Data collection and communication

The data is collected from the DHT 11 sensor and is read with the help of a Python Library. The data should be read from the sensor with a gap of at least 2 seconds.

MQTT is an abbreviation for Message Queuing Telemetry Transport[HTSC08]. It is a lightweight subscribe and publishes system where you can publish and receive messages as a client. It is designed for constrained devices which have low-bandwidth. So it is suitable for IOT devices. It is a very simple protocol designed for messaging. It allows to publish and read data from the IoT devices or nodes.

MQTT consist of the following basic concepts:

- Publish/Subscribe:

  A message can be published to a topics and the message is send to the client who has subscribed to that topic.

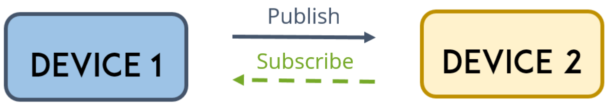


Figure 4.3: MQTT publish-subscribe

- Messages:

  Messages can be either command or data. It is the information that we want to transfer.

- Topics:

  Topics are used to register interest for incoming messages and specify where you want to assign the incoming message. Topics are depicted with strings which are separated by a forward slash. Each forward slash shows a topic level. Topics are case-sensitive. The role of the broker is receiving all the messages, filtering those messages, deciding who is interested in them and then publishing the message to all subscribed clients.

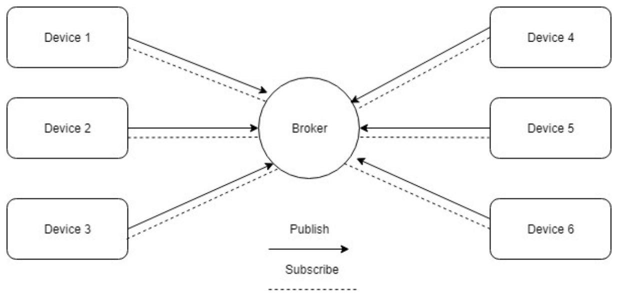


Figure 4.4: MQTT Broker

- Broker:

  The broker is the entity that receives all messages, and the filters those messages, deciding which topic it should go to and who is interested in those topics and posting the message to all subscribed customers or clients.

Chapter 5

Proposed solution

5.1 Overall system architecture

Our overall system architecture can be subdivided into two parts.

- IoT System
- Blockchain System

5.1.1 Blockchain System Setup

Our Hyperledger Blockchain setup consists of two organizations namely org1 and org2 and each organizations having two peers each. We also have a Certificate Authority for each organization named CA1 and CA2. Our peers are named as peer 0 and peer 1. The domain org1 and org2 are used to differentiate between the peers of different organizations.

5.1.2 IOT System Setup

Our IoT system consists of the DHT 11 sensor connected to the raspberry pi via a breadboard. The raspberry pi has the MQTT broker running on it. We also have a Linux machine running the MQTT Client and our HyperLedger Blockchain Network.

5.2 Communication Sequence

The communication sequence is as follows:

- DHT11 sensor measures the temperature and humidity data from the surrounding

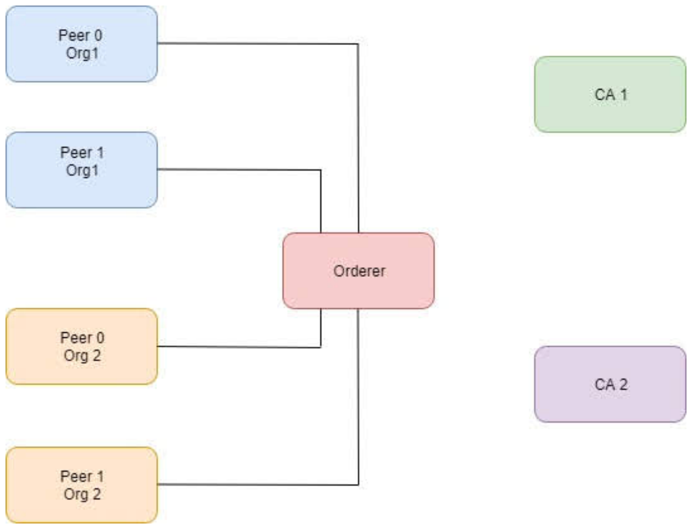


Figure 5.1: HyperLedger Network Setup

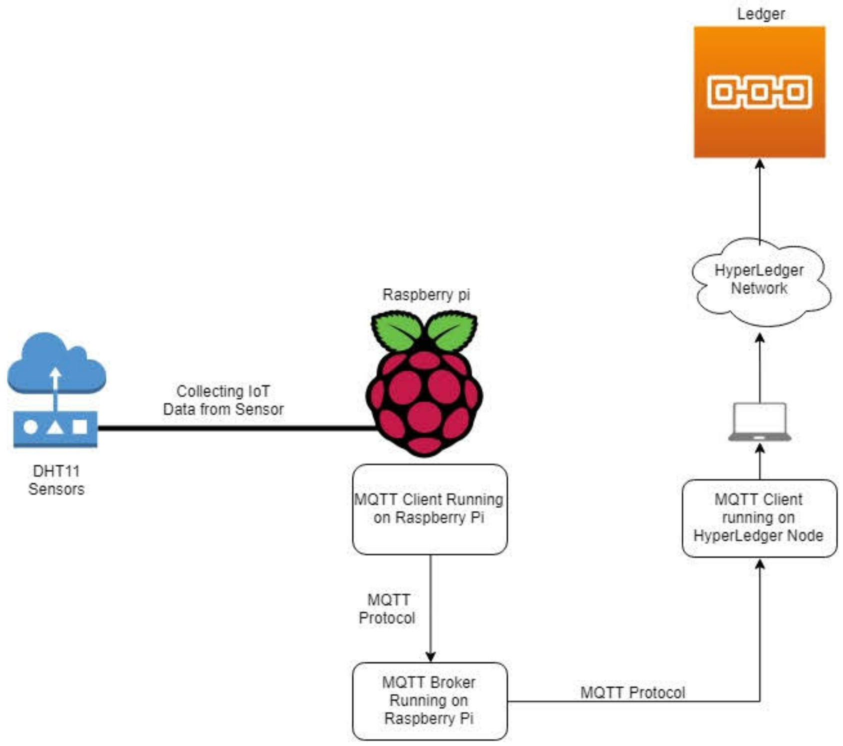

Ledger
HyperLedger Network
Raspberry pi
Collecting IoT Data from Sensor
DHT11 Sensors
MQTT Client Running on Raspberry Pi
MQTT Client running on HyperLedger Node
MQTT Protocol
MQTT Broker Running on Raspberry Pi
MQTT Protocol

Figure 5.2: IoT System Setup

- The data measured is read in raspberry pi using the python adafruit library.
- This data is then time stamped and pushed to a topic on the MQTT broker
- On the Linux machine the client subscribes the same topic and receives the payload.
- Then this payload is processed by a node.js application and a transaction proposal is generated.
- The proposed transaction is then signed by at least one peer from each organization.
- Then this signed transaction proposal is submitted to the Blockchain network.
- Inside the Blockchain network the chaincode is called and appropriate functions are called.
- The chaincode then makes updates to the Ledger and returns a response.
- Then a response is generated by chaincode and returned to the application.
- Finally the application notifies the used that the transaction was successful.

The communication sequence from sensor to Blockchain network is shown below:

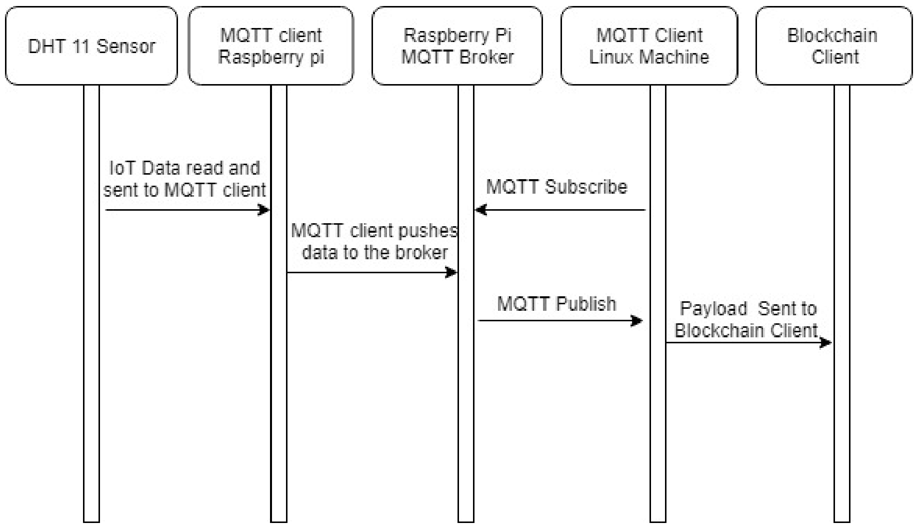
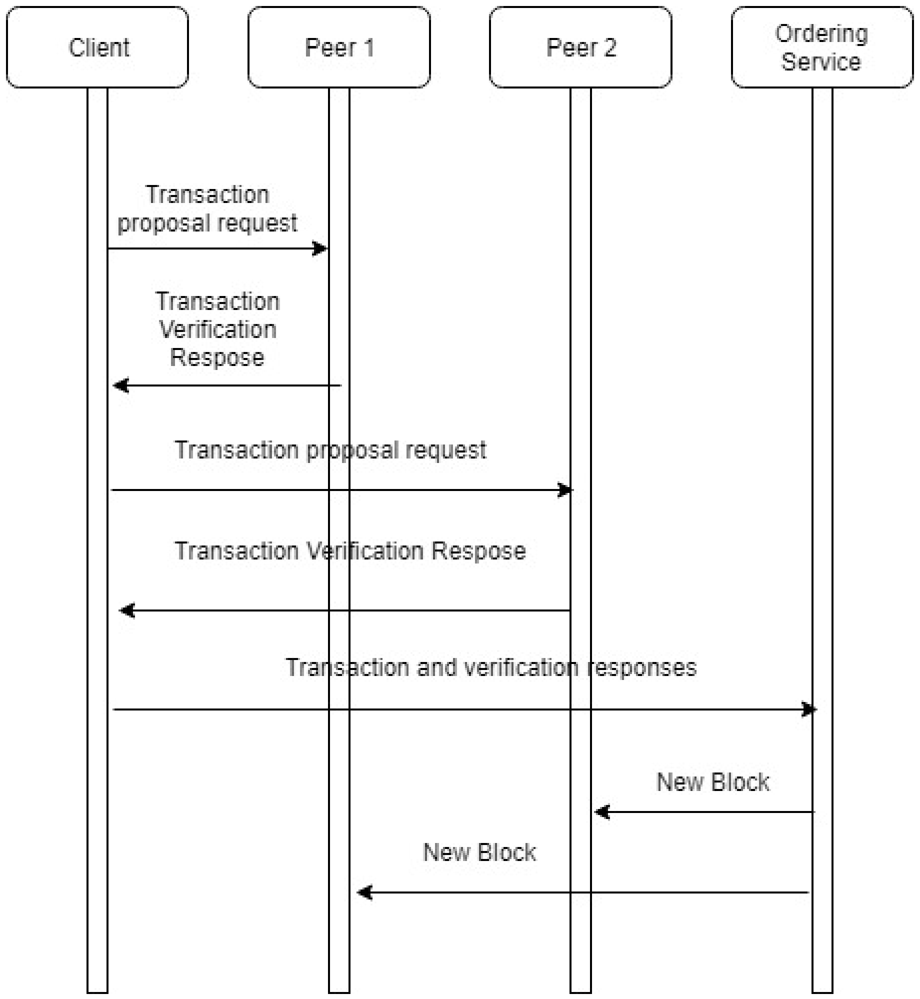

DHT 11 Sensor
MQTT client Raspberry pi
Raspberry Pi MQTT Broker
MQTT Client Linux Machine
Blockchain Client
IoT Data read and sent to MQTT client
MQTT Subscribe
MQTT client pushes data to the broker
MQTT Publish
Payload Sent to Blockchain Client

Figure 5.3: Communication Sequence Sensor to Blockchain

The transaction flow sequence for the Blockchain network is shown below:



Client
Peer 1
Peer 2
Ordering Service
Transaction proposal request
Transaction Verification Respose
Transaction proposal request
Transaction Verification Respose
Transaction and verification responses
New Block
New Block

Figure 5.4: Blockchain System Transaction Flow

5.3 Operation types

Our Blockchain system supports two main operation types.

- Insert data into the Ledger
- Read data from the Ledger

The following two operations are not supported as fundamental to the Blockchain principal.

- Updating any value in the Ledger
- Deleting any value from the Ledger.

For insertion we use the time stamp of the payload as the key and the temperature and humidity as the values. For reading we use the latest time stamp to retrieve the Last inserted value in the Ledger.

- Measuring Temperature and Humidity
  The DHT 11 Sensor is used for measuring the Temperature and Humidity. It can measure temperature in the range of 0-50 degree Celsius. The specification of DHT 11 sensor is as follows

Table 5.1: DHT11 Sensor Specification

Temperature Range	$50° \pm 2°$ Celcius
Humidity Range	20 - 80 % $\pm$ 5 %
Sampling Rate	1 Hz (One Reading Per Second)
Body Size	15.5 mm x 12 mm x 5.5 mm
Operating Voltage	3 - 5 Volts
Max Current	2.5 mA

- Reading Temperature and Humidity

  We use Python adafruit library for reading the temperature form the DHT11 sensor.

- Processing Temperature and Humidity

  Once the temperature and humidity values are read a payload is created with the data values and time stamp. This payload is then sent using MQTT to the broker for further processing.

5.4 Off-chain storage on local DB

Although Blockchain is basically a storage solution, it differs primarily from a database. A database is an organized data collection representing the current state of the system. A database's main functionality is to enable user queries to efficiently retrieve, fuse, and aggregate data. By contrast, most Blockchain implementations represent a ledger in which a history of transactions is recorded. While Ethereum keeps track of a contract state, it provides only limited means of retrieving and processing state data as defined explicitly by the contract. Flexible query language abstractions, data view, schema, join, etc. are not supported in ethereum. In addition, Ethereum records a history of all state changes, resulting in more expensive Blockchain space and less efficient storage compared to a database. Most applications based on Blockchain combine Blockchain with storage offchain.

HyperLedger has inbuilt support for the database to run complex queries on it. HyperLedger supports levelDB by default but provides CouchDB as an alternative too as well. All the queries are run on the World state database rather than on the Ledger which gives faster access to the data.

Chapter 6

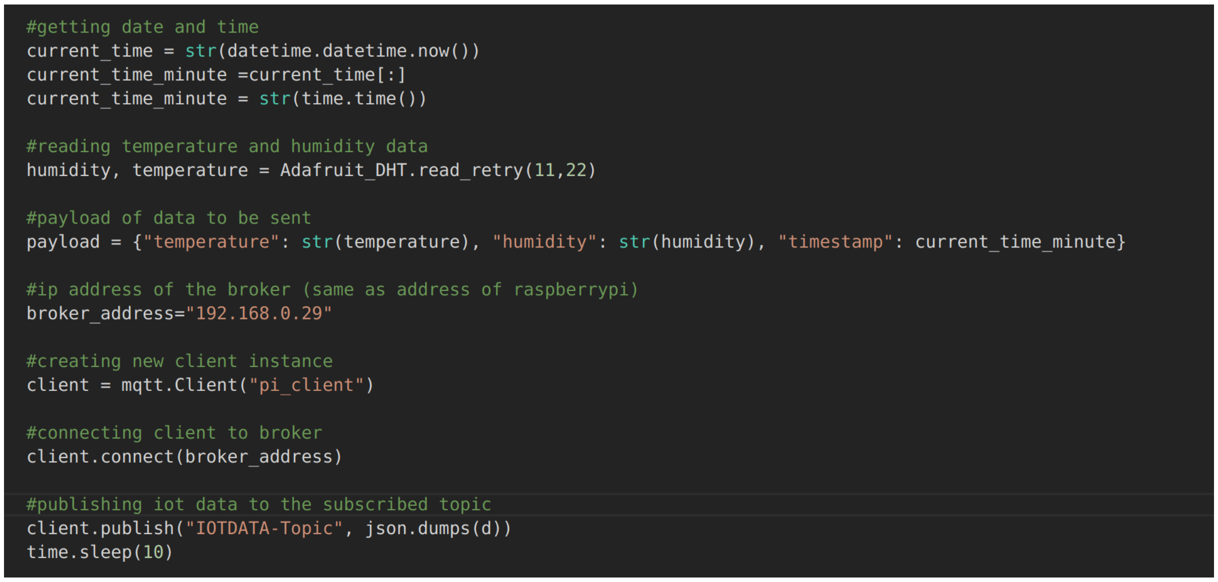
Hyperledger Blockchain Implementation

6.1 How to bring External data (IoT) data into Blockchain

We are using DHT11 sensors as our IoT device. It gives the temperature and humidity data from the surrounding. The DHT11 sensor is also connected to the raspberry pi. The data is read from the sensor using the adafruit python library. Then we use a python program to send this data to the Blockchain node. We use MQTT Protocol to send this payload which consists of the time stamp, temperature and humidity values. This payload is pushed to an MQTT broker running on the raspberry pi to a topic named "IOTDATA-TOPIC". The MQTT client running on HyperLedger fabric subscribes to the topic and receives this data. At the HyperLedger Node, a node.js application receives this data and creates a transaction proposal from the received payload. This transaction is accepted only if it has been signed by at least one peer from each organization. Once the transaction is signed and submitted, it is validated for the signatures. If the transaction is valid then the payload data is added to the ledger.

6.2 Chaincode software design (in Go language)

HyperLedger was one of the first platforms to support go, node.js and java as smart contract languages. Chaincode is simply a smart contract written in any of the supported languages which implement the prescribed interface. The HyperLedger chaincode runs in a docker container separate from the peer who is endorsing it. Chaincode acts as a middleware which uses the transactions



```
#getting date and time
current_time = str(datetime.datetime.now())
current_time_minute =current_time[:]
current_time_minute = str(time.time())

#reading temperature and humidity data
humidity, temperature = Adafruit_DHT.read_retry(11,22)

#payload of data to be sent
payload = {"temperature": str(temperature), "humidity": str(humidity), "timestamp": current_time_minute}

#ip address of the broker (same as address of raspberrypi)
broker_address="192.168.0.29"

#creating new client instance
client = mqtt.Client("pi_client")

#connecting client to broker
client.connect(broker_address)

#publishing iot data to the subscribed topic
client.publish("IOTDATA-Topic", json.dumps(d))
time.sleep(10)
```

Figure 6.1: Sending data from Raspberry pi

submitted by the application to manage the state of the Ledger. We are using go language for writing the chaincode.

6.3 Chaincode execution environment

The chaincode is executed in a docker container which is separate from the endorsing peer. Mostly, the chaincode encapsulated the business logic that is agreed by the various business members in the network so synonymously it can also be called as a "smart contract". The ledger state that created by one chaincode is not directly accessible to another chaincode within the same network. However, a chaincode can invoke another chaincode which is in the same network to make its state accessible if it has permission to do so.

chaincode lifecycle operations are as follows:

- The process of Packaging
- Installing
- Instantiating
- Upgrading

HyperLedger is also planning to add the add and stop feature as transactions which would allow to disable and re-enable a chaincode without uninstalling it.

6.3.1 Packaging

The chaincode package contains 3 parts:

- The chaincode: As defined by CDS or ChaincodeDeploymentSpec. In terms of code, the CDS defines the chaincode package and other properties such as name and version
- Instantiation policy: An optional installation policy that can be syntactically described in the same endorsement policy as described in the Endorsement Policy
- Signatures: A set of chaincode-owned entities signatures.

The signatures are used for the following purposes

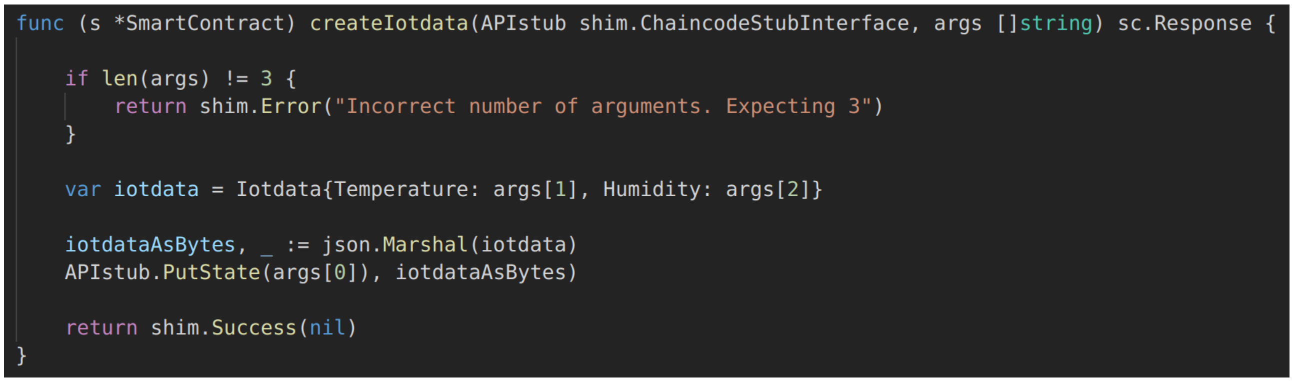
- To establish chain code ownership.
- To enable verification of the package contents.
- To detect packet manipulation or tampering.

There are two approaches to chain code packaging:

- If there are more than one chaincode owners, then the chaincode must be signed by multiple owners with multiple identities. In this case, we can create a signed CDS package which is passed serially to other owners for signatures.
- When you deploy a SignedCDS with only the identity signature of the node issuing the install transaction.

6.3.2 Installing Chaincode

The transaction install does the following two things. First it packages the smart contract or chaincode into a standard format called CDS or ChaincodeDeploymentSpec. Second, It also proceeds with installation of the packaged chaincode on the peer node The chaincode is to be installed on each endorsing peer node of the channel. It should not be installed on the peer nodes which are not endorsing peer nodes as it may affect the confidentiality of the chaincode logic.



```go
func (s *SmartContract) createIotdata(APIstub shim.ChaincodeStubInterface, args []string) sc.Response {

    if len(args) != 3 {
        return shim.Error("Incorrect number of arguments. Expecting 3")
    }

    var iotdata = Iotdata{Temperature: args[1], Humidity: args[2]}

    iotdataAsBytes, _ := json.Marshal(iotdata)
    APIstub.PutState(args[0]), iotdataAsBytes)

    return shim.Success(nil)
}
```

Figure 6.2: Chaincode: Function for Inserting data into the Ledger

6.3.3 Instantiating Chaincode

For initializing and creating a chaincode on the channel, the instantiate transaction is used. This type of transaction is used for invoking the LSCC or Lifecycle System Chaincode which binds the chaincode to a channel. The same chaincode operated independently and separately on each channel on which it is instantiated which means that the state is kept isolated to the channel irrespective of how many channels the same chaincode may be installed. The creator of an instantiate transaction must comply with the chaincode instantiation policy included in SignedCDS and must also be a writer on the channel configured as part of the channel creation process. This is necessary for channel security to prevent unauthorized entities from deploying chaincodes or making members on an unbound channel to execute chaincodes.

6.3.4 Upgrade

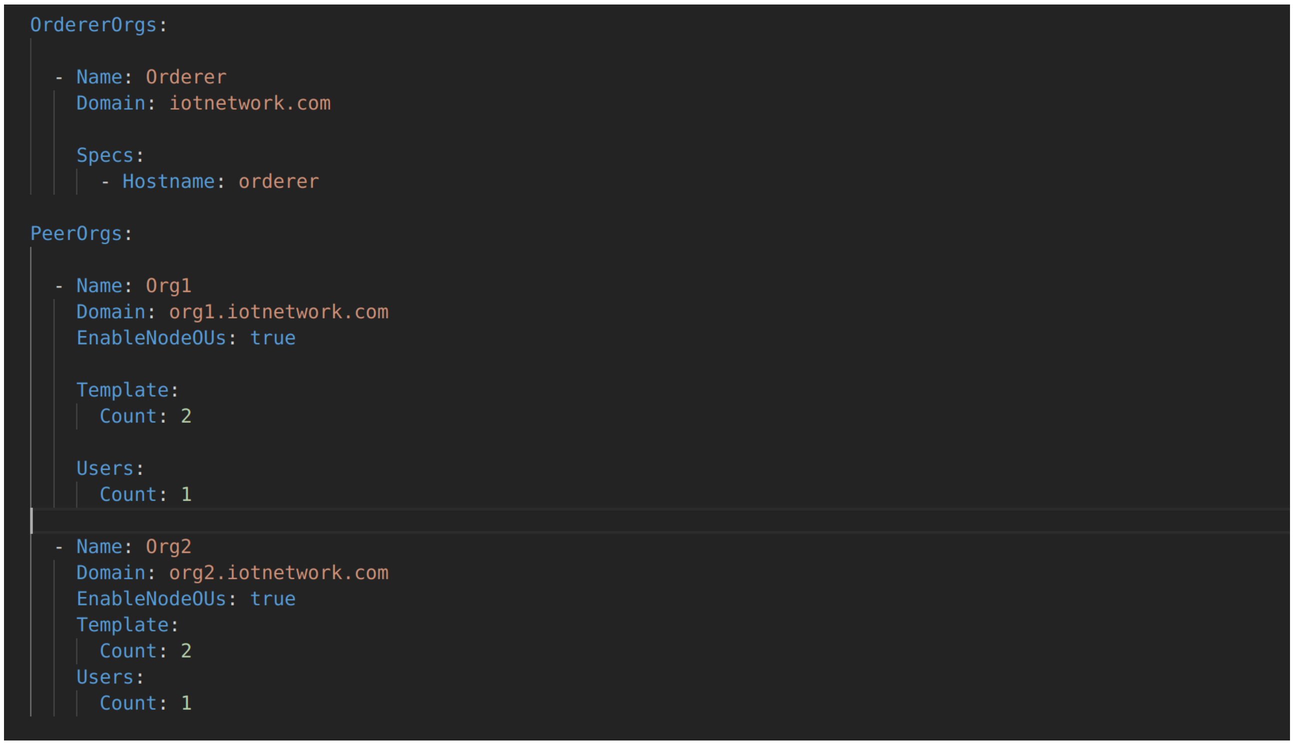
HyperLedger allows upgrading the chaincode as well. It can be upgraded by updating its version number, which is there in the SignedCDS. Other parts like owners and instantiation policy can also be updated while upgrading but these are optional. However, the name of the chaincode should stay the same while upgrading else it would be considered a different chaincode. Before upgrading the new smart contract or chaincode should be installed on all the endorsing peers. Upgrade transaction instantly binds the updated chaincode to the channel. The old channels which may has the same chaincode installed but the previous version will still continue to run the old chaincode version independently. In other words, the upgrade transaction only affects the channel where the upgrade transaction is submitted.

6.4 Building Blockchain Network

6.4.1 Building network artifacts

To generate the network artifacts which include cryptographic material (x509 certs and signing keys) for our different network entities, we used the cryptogen tool provided by HyperLedger.

It takes input from a crypto-config.yaml file which contains our network structure and allows us to generate keys and certificates for various entities described in the input file.



```
OrdererOrgs:

  - Name: Orderer
    Domain: iotnetwork.com

    Specs:
      - Hostname: orderer

PeerOrgs:

  - Name: Org1
    Domain: org1.iotnetwork.com
    EnableNodeOUs: true

    Template:
      Count: 2

    Users:
      Count: 1

  - Name: Org2
    Domain: org2.iotnetwork.com
    EnableNodeOUs: true
    Template:
      Count: 2
    Users:
      Count: 1
```

Figure 6.3: crypto-config.yaml: Input file for crypto generator

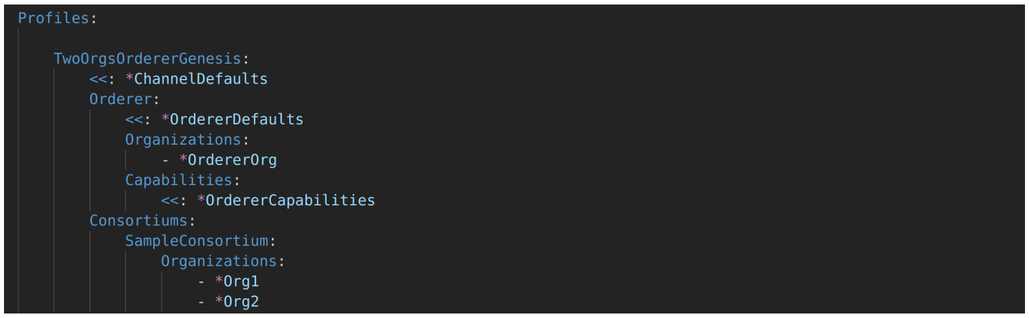
6.4.2 Building Configurations for the Network

In HyperLedger configurations are set and updated in the form of transactions. HyperLedger provides a binary configtxgen or configuration transaction generator for generating these configuration transactions. The configtx is used for generating the following:

- Channel configuration transactions
- Genesis Block

- Anchor peer update transactions

configtx contains all the configurations and policies for orderer, peers, organizations. It also contains the policies which define the rights for each organization.

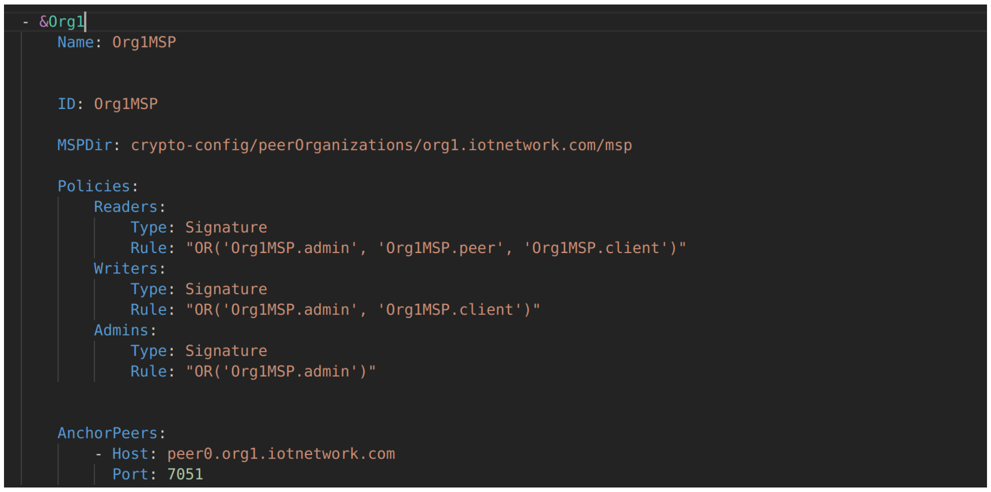


```
Profiles:

    TwoOrgsOrdererGenesis:
        <<: *ChannelDefaults
        Orderer:
            <<: *OrdererDefaults
            Organizations:
                - *OrdererOrg
            Capabilities:
                <<: *OrdererCapabilities
        Consortiums:
            SampleConsortium:
                Organizations:
                    - *Org1
                    - *Org2
```

Figure 6.4: Snippet configtx: profile for generating the genesis Block

The configtx also contains the policies as well the name and is of the various entities that will be used in the network. The policies are categorized as:

- Readers
- Writers
- Admins

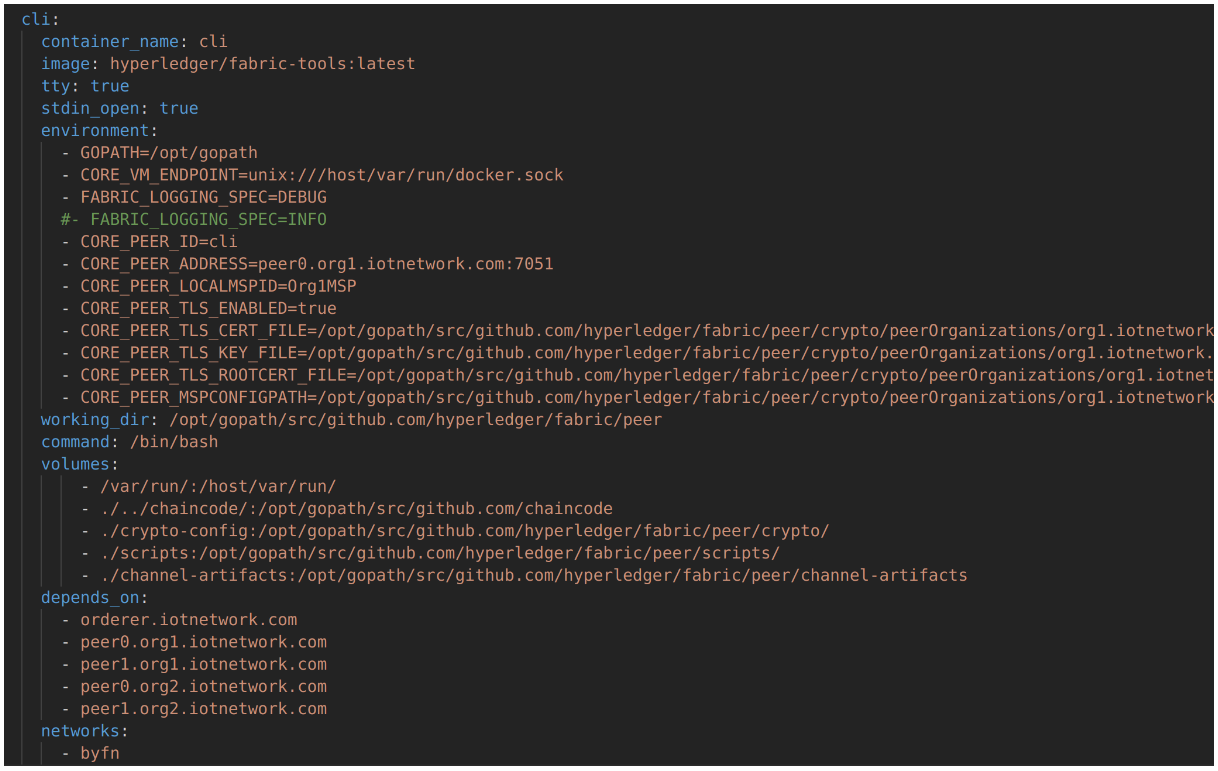


```
- &Org1
    Name: Org1MSP


    ID: Org1MSP

    MSPDir: crypto-config/peerOrganizations/org1.iotnetwork.com/msp

    Policies:
        Readers:
            Type: Signature
            Rule: "OR('Org1MSP.admin', 'Org1MSP.peer', 'Org1MSP.client')"
        Writers:
            Type: Signature
            Rule: "OR('Org1MSP.admin', 'Org1MSP.client')"
        Admins:
            Type: Signature
            Rule: "OR('Org1MSP.admin')"


    AnchorPeers:
        - Host: peer0.org1.iotnetwork.com
          Port: 7051
```

Figure 6.5: Snippet: policies for org1

6.5 Docker containers configurations

We need to define the configuration for various docker containers for various network entities like peers, orderers, CouchDB, cli, etc. All this configuration goes into docker compose file. The snippets of docker configuration for cli and orderer are shown below.



```
cli:
  container_name: cli
  image: hyperledger/fabric-tools:latest
  tty: true
  stdin_open: true
  environment:
    - GOPATH=/opt/gopath
    - CORE_VM_ENDPOINT=unix:///host/var/run/docker.sock
    - FABRIC_LOGGING_SPEC=DEBUG
    #- FABRIC_LOGGING_SPEC=INFO
    - CORE_PEER_ID=cli
    - CORE_PEER_ADDRESS=peer0.org1.iotnetwork.com:7051
    - CORE_PEER_LOCALMSPID=Org1MSP
    - CORE_PEER_TLS_ENABLED=true
    - CORE_PEER_TLS_CERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.iotnetwork
    - CORE_PEER_TLS_KEY_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.iotnetwork.
    - CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.iotnet
    - CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.iotnetwork
  working_dir: /opt/gopath/src/github.com/hyperledger/fabric/peer
  command: /bin/bash
  volumes:
      - /var/run/:/host/var/run/
      - ./../chaincode/:/opt/gopath/src/github.com/chaincode
      - ./crypto-config:/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
      - ./scripts:/opt/gopath/src/github.com/hyperledger/fabric/peer/scripts/
      - ./channel-artifacts:/opt/gopath/src/github.com/hyperledger/fabric/peer/channel-artifacts
  depends_on:
    - orderer.iotnetwork.com
    - peer0.org1.iotnetwork.com
    - peer1.org1.iotnetwork.com
    - peer0.org2.iotnetwork.com
    - peer1.org2.iotnetwork.com
  networks:
    - byfn
```

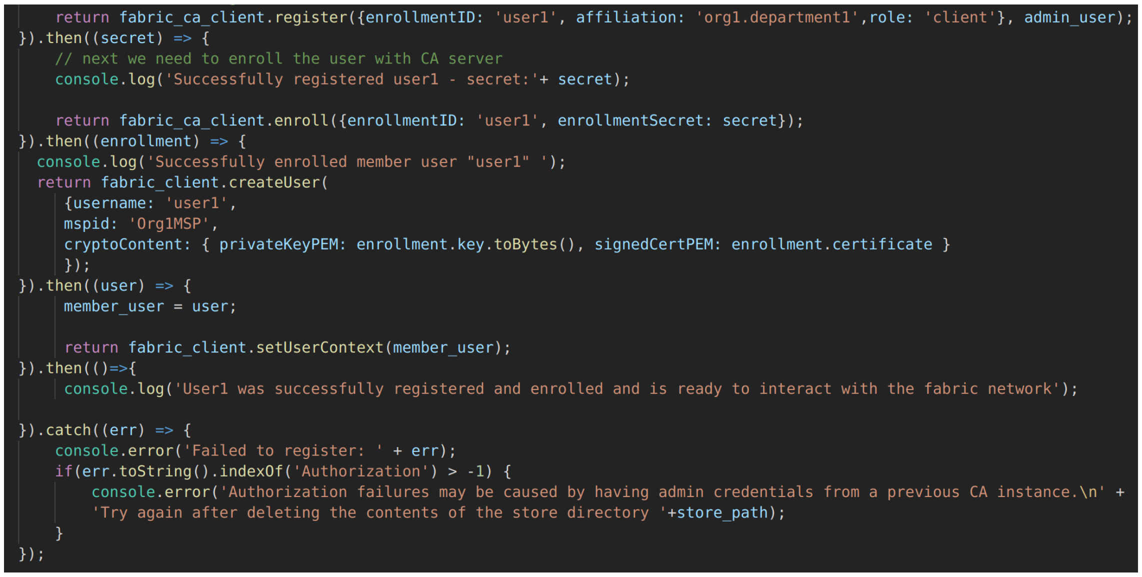
Figure 6.6: Snippet: cli docker configuration

The cli container is used for accessing the various peers and orderers in the network. we define all the attributes, the volumes to be mounted here. It also contains the path to the keys, certificates, MSP configuration as well. The various attributes defined are as follows:

- Core Peer Address including the port number
- Core Peer ID
- Core Peer TLS Enabled Flag
- Volumes that will get mounted into the peer directory structure and many more

6.6 Accessing Deployed Network

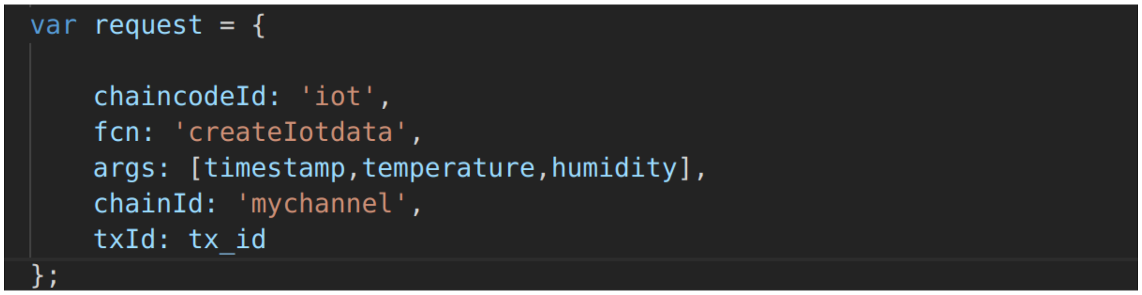
Before the client application can start submitting requests to the Blockchain network. We need to get the required certificates from the CA. Code snippet for registering a user with the CA and getting the user certificates is shown below.



```
    return fabric_ca_client.register({enrollmentID: 'user1', affiliation: 'org1.department1',role: 'client'}, admin_user);
}).then((secret) => {
    // next we need to enroll the user with CA server
    console.log('Successfully registered user1 - secret:'+ secret);

    return fabric_ca_client.enroll({enrollmentID: 'user1', enrollmentSecret: secret});
}).then((enrollment) => {
  console.log('Successfully enrolled member user "user1" ');
  return fabric_client.createUser(
     {username: 'user1',
     mspid: 'Org1MSP',
     cryptoContent: { privateKeyPEM: enrollment.key.toBytes(), signedCertPEM: enrollment.certificate }
     });
}).then((user) => {
     member_user = user;

     return fabric_client.setUserContext(member_user);
}).then(()=>{
     console.log('User1 was successfully registered and enrolled and is ready to interact with the fabric network');

}).catch((err) => {
    console.error('Failed to register: ' + err);
    if(err.toString().indexOf('Authorization') > -1) {
        console.error('Authorization failures may be caused by having admin credentials from a previous CA instance.\n' +
        'Try again after deleting the contents of the store directory '+store_path);
    }
});
```

Figure 6.7: Snippet: Transaction request A

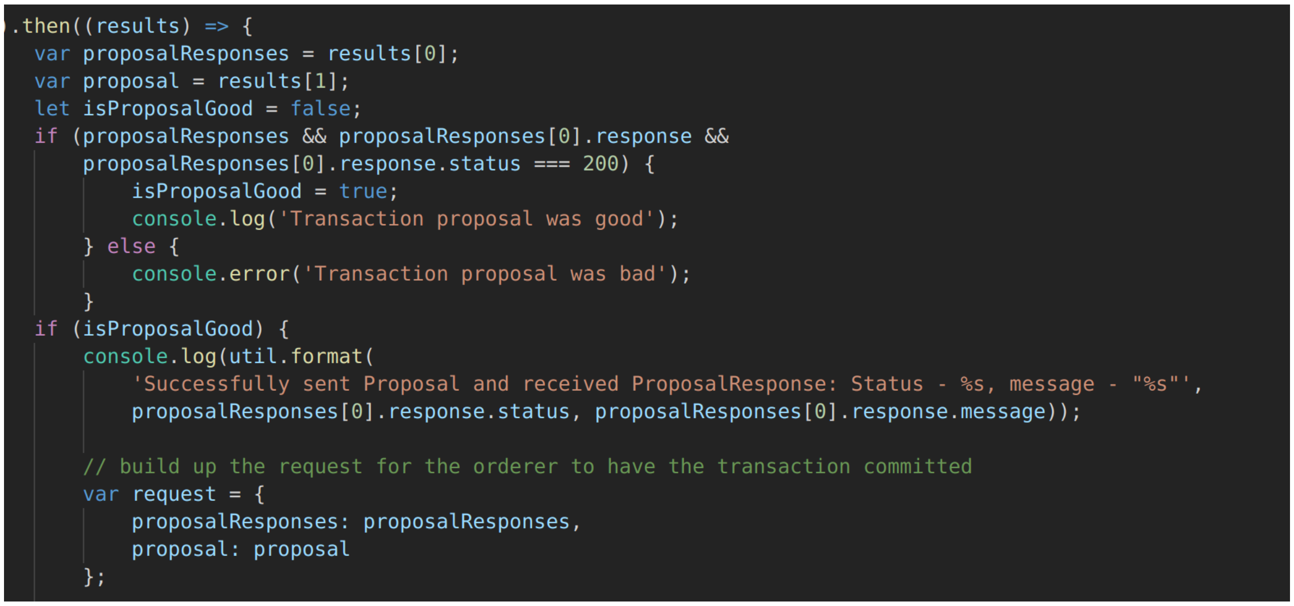
We build a node.js console application for interacting with Blockchain Network. Transaction request for inserting data into the ledger by the client is shown below.



```
var request = {

    chaincodeId: 'iot',
    fcn: 'createIotdata',
    args: [timestamp,temperature,humidity],
    chainId: 'mychannel',
    txId: tx_id
};
```

Figure 6.8: Snippet: Transaction request B

We create a request which contains the chaincode id, chaincode function to be called, arguments for the function, the channel id and a transaction id that we explicitly assign to the transaction.



```
.then((results) => {
    var proposalResponses = results[0];
    var proposal = results[1];
    let isProposalGood = false;
    if (proposalResponses && proposalResponses[0].response &&
        proposalResponses[0].response.status === 200) {
            isProposalGood = true;
            console.log('Transaction proposal was good');
        } else {
            console.error('Transaction proposal was bad');
        }
    if (isProposalGood) {
        console.log(util.format(
            'Successfully sent Proposal and received ProposalResponse: Status - %s, message - "%s"',
            proposalResponses[0].response.status, proposalResponses[0].response.message));

        // build up the request for the orderer to have the transaction committed
        var request = {
            proposalResponses: proposalResponses,
            proposal: proposal
        };
```

Figure 6.9: Snippet: Sending the transaction proposal with request to orderer

The transaction proposal is then sent with the above request to the network which checks is the transaction proposal was good or bad. If the proposal was good a request is generated for the orderer to have it committed by the peers.

Chapter 7

Performance Analysis

7.1 Transaction Latency

Transaction latency is the time difference between when the transaction gets submitted and the time the transaction gets committed across the network, confirming the transaction. Unlike lottery-based consensus protocols like Bitcoin or Ethereum where the finality of the transaction is determined using a probabilistic approach, Fabric's consensus process leads to a deterministic finality approach. The time to start the fabric network with all the docker containers ranges between 28-51 seconds.

7.2 Performance comparison under different transaction load

The parameters that we have taken into consideration for analysis are as follows:

- BatchTimeout:
  This is the amount of time the orderer waits before creating a batch.
- BatchSize:
  - Max Message Count:
    This is the maximum number of messages a batch can contain
  - Absolute Max Bytes:
    This is the maximum size in bytes of messages that can be allowed in a batch
  - Preferred Max Bytes:
    The preferred maximum number of bytes allowed for the serialized messages in a batch. A message larger than the preferred max bytes will result in a batch larger than preferred max bytes.

Table 7.1: Table 1 : Number of records vs Delay in writing

Number of records	**Delay in Writing (ms)**
50	155
100	246
250	612
500	1037
750	1469
1000	2394
1250	4127
1500	4681

The start up execution time for the below configuration was 31 seconds.

- BatchTimeout: 2 second
- BatchSize:
  - Max Message Count:10
  - Absolute Max Bytes: 99 Mega Bytes
  - Preferred Max Bytes: 512 Kilo Bytes

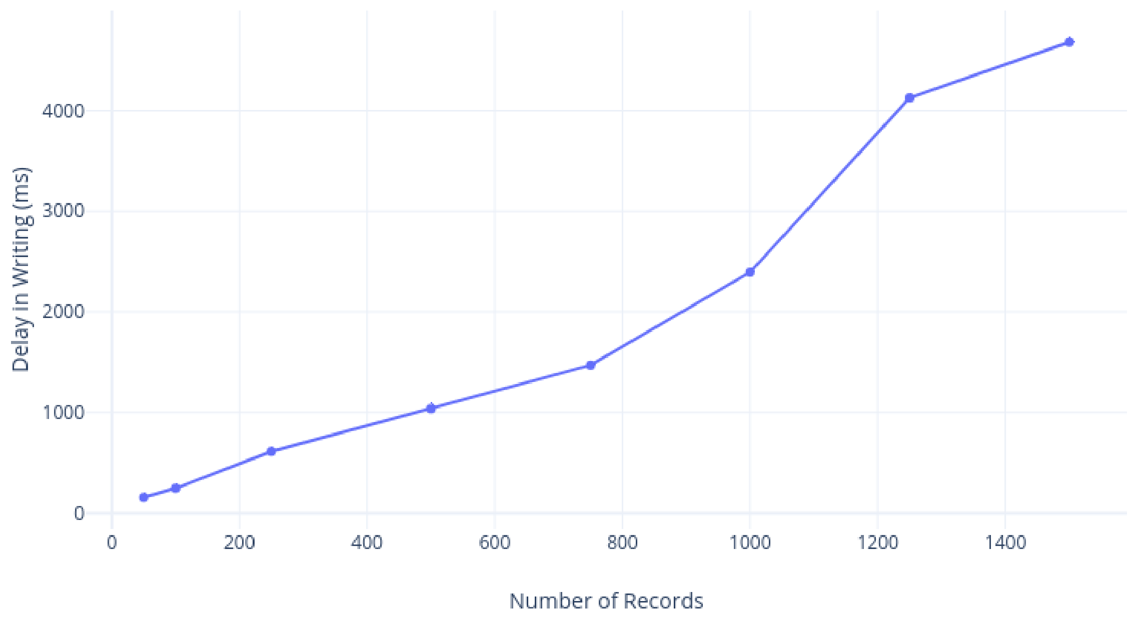


Figure 7.1: Plot 1 : Number of Records vs Delay in Writing

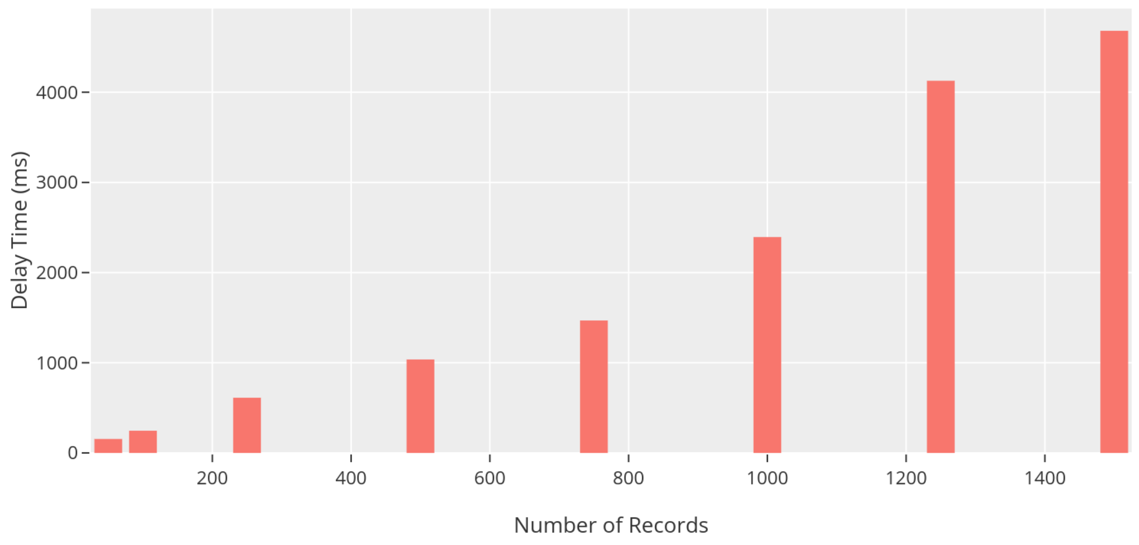


Figure 7.2: Bar Graph 1: Number of records vs Delay time

Table 7.2: Table 2 : Number of records vs Delay in writing

Number of records	**Delay in Writing (ms)**
50	84
100	169
250	369
500	785
750	1157
1000	1787

The start up execution time for the below configuration was 44 seconds.

- BatchTimeout: 4 second
- BatchSize:
  - Max Message Count:20
  - Absolute Max Bytes: 99 Mega Bytes
  - Preferred Max Bytes: 512 Kilo Bytes

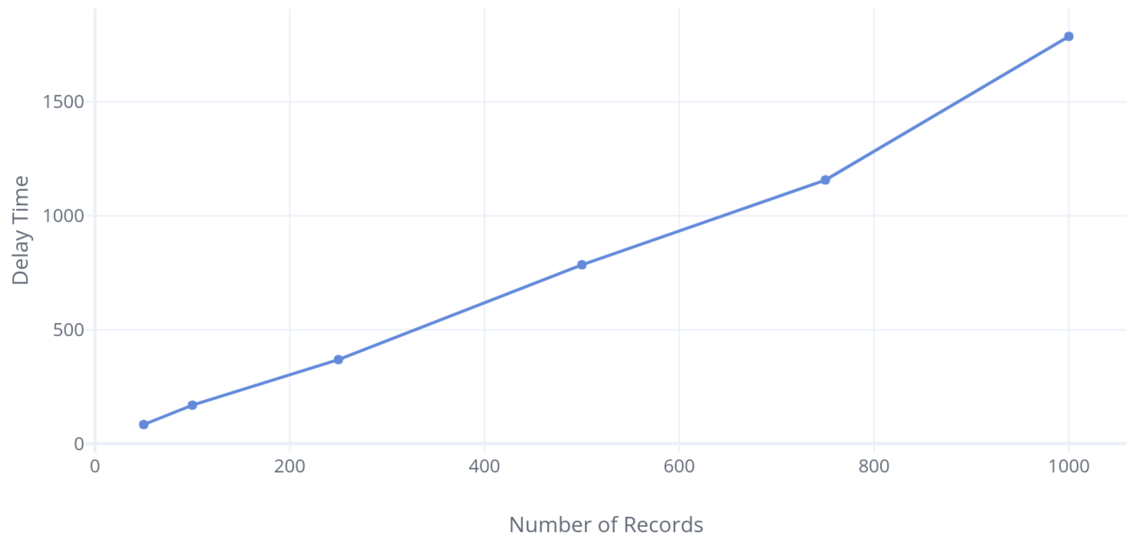


Figure 7.3: Plot 2 : Number of Records vs Delay in writing

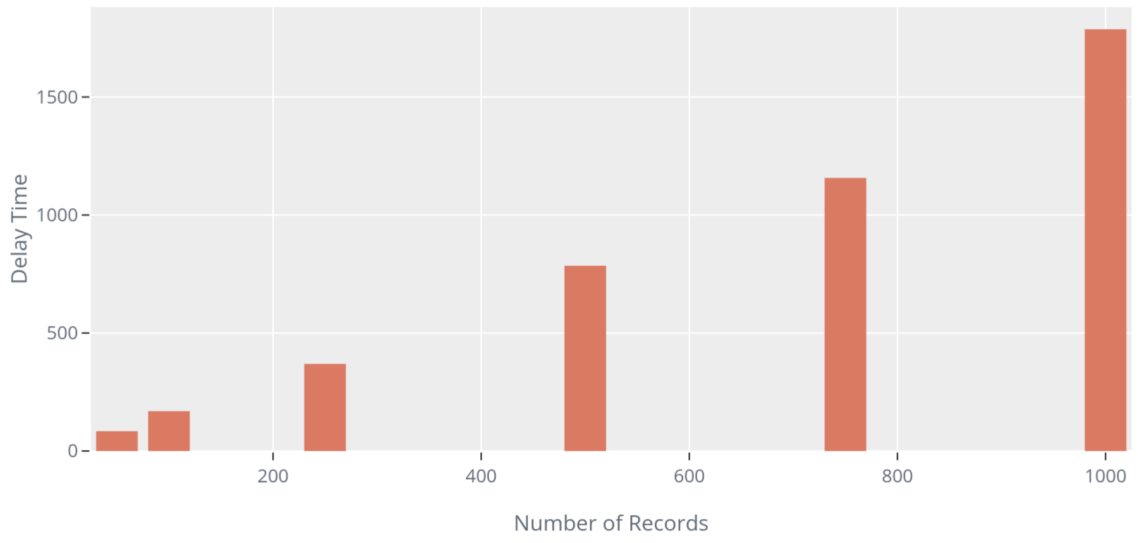


Figure 7.4: Bar Graph 2: Number of records vs Delay time

Table 7.3: Table 3 : Number of records vs Delay in writing

Number of records	**Delay in Writing (ms)**
50	111
100	172
250	432
500	772
750	1287

The start up execution time for the below configuration was 34 seconds.

- BatchTimeout: 1 second
- BatchSize:
  - Max Message Count:5
  - Absolute Max Bytes: 99 Mega Bytes
  - Preferred Max Bytes: 512 Kilo Bytes

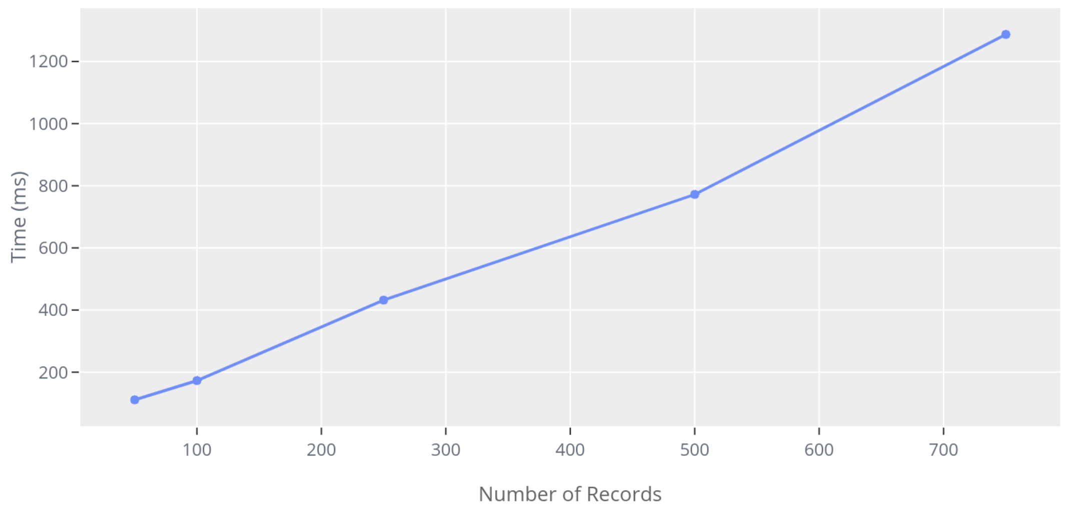


Figure 7.5: Plot 3 of Number of Records vs Delay in Writing

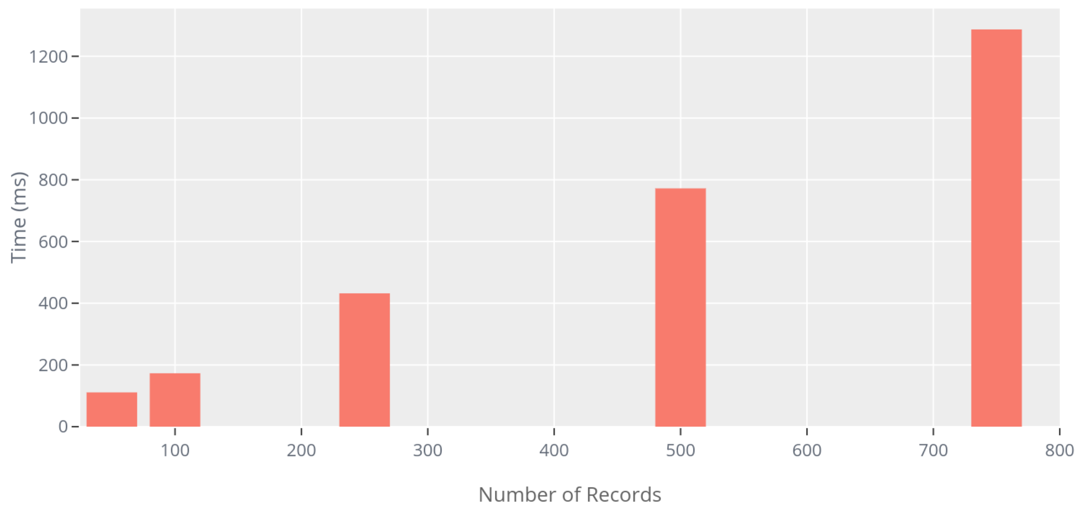


Figure 7.6: Bar Graph 3: Number of records vs Delay time

Comparing the performance of All 3 parameters sets:

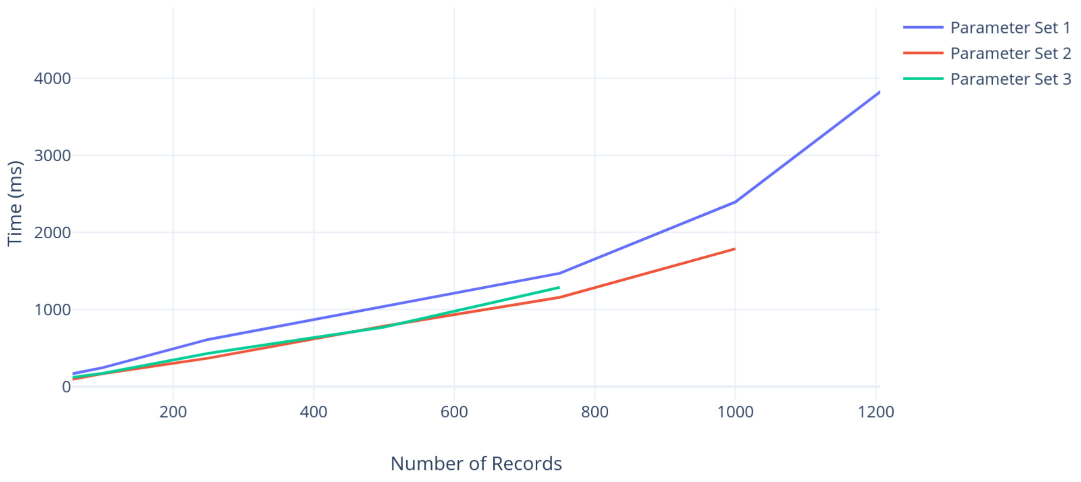


Figure 7.7: Plot 4 : Number of Records vs Delay in writing

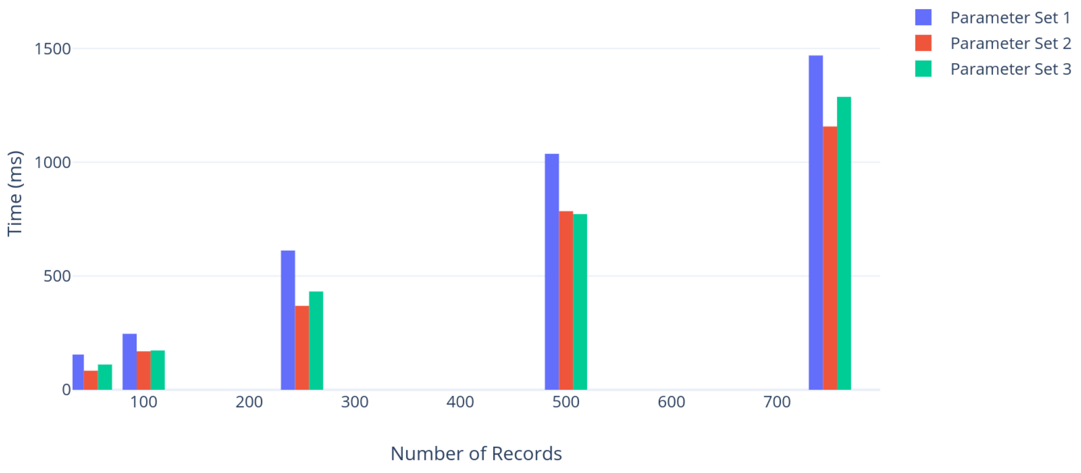


Figure 7.8: Bar Graph 4: Number of records vs Delay time

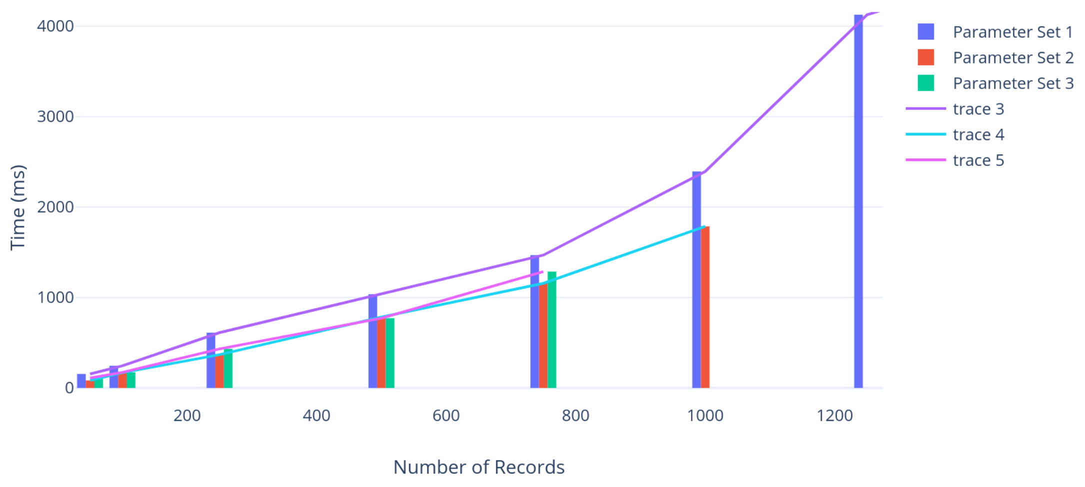


Figure 7.9: Graph : Number of records vs Delay time

7.3 Summary

As we can see from the above graphs that the time for writing increases as the number of records or the transactions increases. We also see that the performance varies as test our network with various parameters.

We can see the performance for parameter set 1 is better than the parameter set 2 and 3. For parameter set 1, we are using a batch timeout of 2 seconds and max message count of 10. This means that a block would be added to the ledger if either the number of messages has reached 10 or the time of 2 seconds is passed whichever is earlier.

For parameter set 2, we use a batch timeout of 4 sec and max message count of 20. So increasing the batch timeout and max message count does not necessarily increase the performance. In fact not only increasing but decreasing the batch timeout and max count value doesn't necessarily improve the performance as can we can see in the graph, the performance of set 2 and 3 is very similar where we have used batch time out of 1 sec and max message count of 5. We need to maintain a balance between the batch timeout and the max message count for the optimal performance.

Another observation is that our orderer is able to handle a maximum number of transactions for the parameter set 1 followed by the parameter set 2 and parameter set 3. This reason for this is that for set 3 we are using max message count of 5 which is the lowest for the 3 sets. This means that if a load of transactions is more then the batch timeout will become less relevant as the messages received by orderer is much before the batch timeout. But since the orderer takes some time to create a block and then send it to peers some transactions may get timeout as the orderer is not able to handle it within the required time frame.

We can conclude that batch timeout, max message count, absolute max bytes, preferred max bytes play a key role in the performance of the network. Increasing one parameter does not necessarily mean improvement in performance. A balance is to be maintained between these values depending on the load of transactions expected.

Chapter 8

Conclusion and Future Work

8.1 Conclusion

Our IoT Blockchain implementation works successfully which allows us to store IoT data which is trustworthy and verified. Our network works well for a decent number of transactions but having just one orderer limits its performance in terms of the number of transaction it can handle. On Average, our solo configuration orderer is able to handle about 750-1500 transactions depending on various parameters like the batch timeout, batch size and other.
We conclude that HyperLedger fabric platform can be used for managing IoT data and the inbuild endorsements, encryption and signing ensures the security, validity, and integrity of the data. All the transactions in HyperLedger are TLS encrypted which ensures that data cannot be compromised during the internal network communications.

8.2 Future Work

8.2.1 Using Kafka

Kafka is primarily a fault-tolerant, distributed, horizontally-scalable, commit log A commit log is basically an appended data structure. There is no modification or deletion [LNCB17] that leads to no read/write locks and the worst-case complexity O(1). With their corresponding Zookeeper ensemble, there may be multiple Kafka nodes in the Blockchain network. We plan to extend our configuration to support Kafka in which we have more than one orderers.

8.2.2 Using Raft

Our IOT Blockchain network implementation was done with solo configuration which uses one orderer. But we plan to extend our idea to Raft based ordering service too. Raft support became available in HyperLedger since version 1.3. Using Raft will make our network more scalable and fault tolerant [SK17]. It will also give a huge gain in terms of performance and load balancing. Also managing raft nodes is comparatively easier than Kafka and does not require the overhead of configuring and Managing Zookeeper ensemble along with Kafka nodes

Bibliography

[AMQ13] Pierre-Louis Aublin, Sonia Ben Mokhtar, and Vivien Quéma. Rbft: Redundant byzantine fault tolerance. In *2013 IEEE 33rd International Conference on Distributed Computing Systems*, pages 297–306. IEEE, 2013.

[BCGH16] Richard Gendal Brown, James Carlyle, Ian Grigg, and Mike Hearn. Corda: an introduction. *R3 CEV, August*, 2016.

[DHT] Dht11 & dht22 sensors temperature and humidity tutorial using arduino - https://howtomechatronics.com/tutorials/arduino/dht11-dht22-sensors-temperature-and-humidity-tutorial-using-arduino/.

[DRDJ17] PP Fathima Dheena, Greema S Raj, Gopika Dutt, and S Vinila Jinny. Iot based smart street light management system. In *2017 IEEE International Conference on Circuits and Systems (ICCS)*, pages 368–371. IEEE, 2017.

[HCD$^+$19] Dijiang Huang, Chun-Jen Chung, Qiuxiang Dong, Jim Luo, and Myong Kang. Building private blockchains over public blockchains (pop): an attribute-based access control approach. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, pages 355–363. ACM, 2019.

[HTSC08] Urs Hunkeler, Hong Linh Truong, and Andy Stanford-Clark. Mqtt-sa publish/subscribe protocol for wireless sensor networks. In *2008 3rd International Conference on Communication Systems Software and Middleware and Workshops (COMSWARE'08)*, pages 791–798. IEEE, 2008.

[Hyp19] Hyperledger fabric - https://hyperledger-fabric.readthedocs.io/en/release-1.4/, 2019.

[IOTa] Iot industry -https://builtin.com/blockchain/blockchain-iot-examples.

[IOTb] Iot statistics -https://www.vxchnge.com/blog/iot-statistics.

[Iro] Hyperledger iroha - https://www.hyperledger.org/projects/iroha.

[KA13] Kaivan Karimi and Gary Atkinson. What the internet of things (iot) needs to become a reality. *White Paper, FreeScale and ARM*, pages 1–16, 2013.

[KLR$^+$17] Chris Khan, Antony Lewis, Emily Rutland, Clemens Wan, Kevin Rutter, and Clark Thompson. A distributed-ledger consortium model for collaborative innovation. *Computer*, 50(9):29–37, 2017.

[KV17] Kitti Klinbua and Wiwat Vatanawood. Translating tosca into docker-compose yaml file using antlr. In *2017 8th IEEE International Conference on Software Engineering and Service Science (ICSESS)*, pages 145–148. IEEE, 2017.

[LNCB17] Paul Le Noac'H, Alexandru Costan, and Luc Bougé. A performance evaluation of apache kafka in support of big data streaming applications. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 4803–4806. IEEE, 2017.

[SK17] Ermin Sakic and Wolfgang Kellerer. Response time and availability study of raft consensus in distributed sdn control plane. *IEEE Transactions on Network and Service Management*, 15(1):304–318, 2017.

[SMC+17] Harish Sukhwani, José M Martínez, Xiaolin Chang, Kishor S Trivedi, and Andy Rindos. Performance modeling of pbft consensus process for permissioned blockchain network (hyperledger fabric). In *2017 IEEE 36th Symposium on Reliable Distributed Systems (SRDS)*, pages 253–255. IEEE, 2017.

[SS18] Chinmay Saraf and Siddharth Sabadra. Blockchain platforms: A compendium. In *2018 IEEE International Conference on Innovative Research and Development (ICIRD)*, pages 1–6. IEEE, 2018.

[SSA17] SP Srinivasan, D Sorna Shanthi, and Aashish V Anand. Inventory transparency for agricultural produce through iot. In *IOP Conference Series: Materials Science and Engineering*, volume 211, page 012009. IOP Publishing, 2017.

[VS17] Martin Valenta and Philipp Sandner. Comparison of ethereum, hyperledger fabric and corda. *[ebook] Frankfurt School, Blockchain Center*, 2017.

[Vuk17] Marko Vukolić. Rethinking permissioned blockchains. In *Proceedings of the ACM Workshop on Blockchain, Cryptocurrencies and Contracts*, pages 3–7. ACM, 2017.

[YNZ+19] Kazuhiro Yamashita, Yoshihide Nomura, Ence Zhou, Bingfeng Pi, and Sun Jun. Potential risks of hyperledger fabric smart contracts. In *2019 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, pages 1–10. IEEE, 2019.

[ZXD+18] Zibin Zheng, Shaoan Xie, Hong-Ning Dai, Xiangping Chen, and Huaimin Wang. Blockchain challenges and opportunities: a survey. *International Journal of Web and Grid Services*, 14(4):352–375, 2018.

Curriculum Vitae

Graduate College

University of Nevada, Las Vegas

Akhil David

akhildavid26@hotmail.com

Degrees:

Bachelor Degree in Information Technology 2016

Kurukshetra University, India

Thesis Title: Managing IoT Data on HyperLedger Blockchain

Thesis Examination Committee:

Chairperson, Dr. Yoohwan Kim , Ph.D.

Committee Member, Dr. Ju-Yeon Jo, Ph.D.

Committee Member, Dr. Laxmi Gewali, Ph.D.

Graduate Faculty Representative, Dr. Yingtao Jiang, Ph.D.