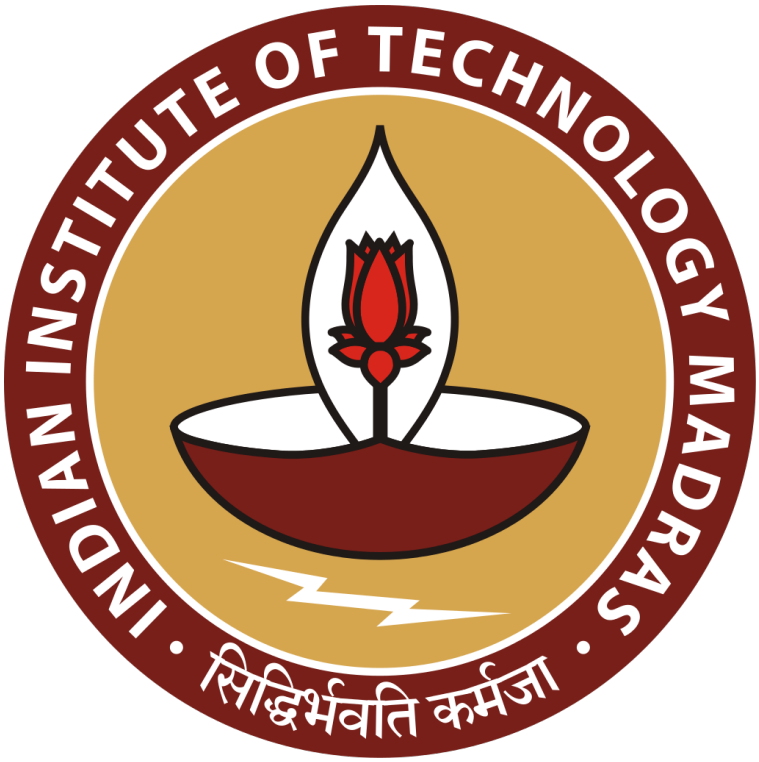
Hospital Management Information System

(HMIS)



Submitted by:

Hrishikesh Tiwari (CS22M047)

Under the guidance of:

Prof. John Augustine

Visit **https://github.com/hrishi-13/HMIS-dApp** for more details.

THESIS CERTIFICATE

This is to certify that the thesis entitled

"HOSPITAL MANAGEMENT INFORMATION SYSTEM (HMIS)"

submitted by

Hrishikesh Tiwari

to the Indian Institute of Technology Madras, for the award of **Master of Technology**, is a bona fide record of research work carried out under the supervision of **Dr. John Augustine**. No part of this thesis has been submitted for any other degree or diploma.

The research work presented in this thesis has been carried out at the

Computer Science Department

IIT MADRAS

Supervisor: Dr. John Augustine

Place: Chennai

Date: May 10, 2024

Acknowledgment

I would like to express my sincere gratitude to Prof. Prabhu Rajagopal and Prof. John Augustine who have contributed to the successful completion of phase 1 of this project. Without their support, guidance, and encouragement, this project would not have been possible. I would like to thank them for their mentorship throughout this project. Their expertise, patience, and unwavering support were instrumental in shaping the direction of this work.

I always wanted to learn about blockchain technologies and with my professors insightful feedback and constructive criticism, I have tried to learn as much as possible. I would like to extend my appreciation to Ankit, Anirudh and Vijay Sir who provided valuable insights and shared their knowledge, contributing to a collaborative learning environment. Last but not least, I acknowledge the resources and facilities provided by CNDE Lab and CCD Lab, which were crucial for the successful execution of phase 1 of this project. This project was made possible through the combined efforts of all these individuals and organizations.

Abstract

In today's dynamic healthcare landscape, efficient patient management and electronic medical record (EMR) automation are critical for enhancing healthcare services.

The proposed system utilizes blockchain technology to create a decentralized, transparent, and immutable ledger for patient data, ensuring the integrity, security, and interoperability of health records. This innovative approach not only facilitates seamless access to accurate patient information for healthcare providers but also empowers patients to have control over their data, enhancing patient privacy and consent management.

The project aims to design and develop a blockchain-based Hospital Management Information System (H-MIS) that integrates patient management and EMR automation, also ensure data security, privacy, and integrity using blockchain technology. The security part involves improving the resistance of the HMIS towards various attacks (Ransomware attack, Network Sniffing, DDOS, ,etc). By leveraging smart contracts, the system automates and streamlines various healthcare processes, from billing to insurance claims, improving efficiency and reducing administrative costs.

CONTENTS

Abbreviations

IITM IIT Madras

EHR Electronic Health Record

EMR Electronic Medical Record

HLF Hyprledger Fabric

POW Proof of Work

POS Proof of Stake

MSP Membership Service Provider

pBFT Practical Byzantine Fault Tolerance

CA Certificate Authority

SDK Software Development Kit

CHAPTER 1

INTRODUCTION

1.1 EXISTING SOLUTIONS

Currently, Electronic Health Record (EHR) Systems are widely used, which serve as electronic representations of patients' medical records. EHRs digitally store various important patient information like medical histories, diagnoses, medication details, and treatment protocols, essentially replacing traditional paper charts. Additionally, Health Information Exchange (HIE) Systems are employed by hospitals. These systems enable efficient exchange of patient information among various healthcare providers and entities, enhancing the coordination of patient care.

Despite their transformative impact on healthcare administration, these systems face certain challenges. For instance, EHR and HIE systems often encounter difficulties with interoperability, hindering the seamless exchange of information across diverse platforms and systems. Furthermore, securing patient data is a critical issue, as hospitals and healthcare organizations frequently become targets of cyberattacks.

1.2 MOTIVATION

Blockchain technology is a decentralized and distributed system utilized across various sectors, including supply chain management, logistics, and finance. It offers a secure and distributed ledger that can be queried without unauthorized access, thereby reducing the need for multiple resources and costs for secure database access. Hyperledger Fabric, a permissioned blockchain framework, ensures trust among network participants through the use of Certificate Authorities (CAs) and Membership Service Providers (MSPs), making it ideal for multiple entities using a shared database.

In healthcare, the storage and exchange of medical information are crucial. However, sharing personal data across various entities without adequate security measures risks critical data breaches. Moreover, improper handling of customer and personal data could lead to severe consequences, such as unauthorized access or modification of personal medical records. The decentralized, encrypted nature of blockchain provides robust security against unauthorized access and cyber threats. Each transaction on a blockchain is encrypted and connected to the previous one, forming a tamper-proof historical record.

Challenges in data security and privacy persist in current electronic health record systems, especially regarding the interoperability of data exchange. The healthcare sector faces significant risks related to data breaches and potential lapses in security protocols. Blockchain in healthcare can safeguard personal and medical patient information, ensuring access and modifications are controlled by authorized individuals through smart contracts. These contracts facilitate specific operations among various identities within the system and streamline administrative processes, including billing, claims processing, and compliance with regulations.

Consequently, there is a clear need for a decentralized method of data sharing and storage, where patients have greater confidence in the security and privacy of their data, and all involved parties gain a comprehensive view of overall events and interactions.

1.3 WHY BLOCKCHAIN

Here are some ways in which blockchain can be useful in the context of HMIS:

1. **Data Security and Integrity:** The unchangeable characteristic of blockchain technology guarantees that recorded data remains unaltered and undeletable unless there is agreement across the network. This aspect is vital for preserving the accuracy and confidentiality of medical records, patient details, and other critical information.

2. **Interoperability:** Blockchain can facilitate data exchange and interoperability between different departments, hospitals, and healthcare systems. This can lead to improved patient care by ensuring that accurate and up-to-date information is

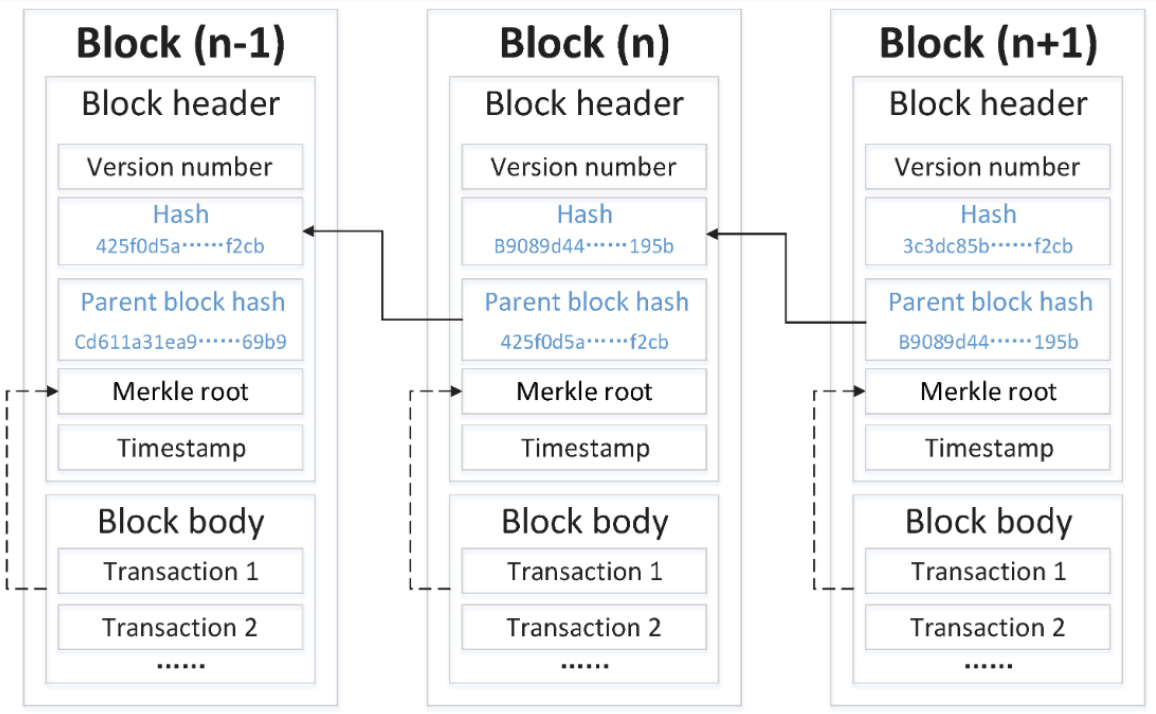
accessible to authorized parties.

3. **Medical Records Management:** Blockchain technology enables the secure storage, retrieval, and sharing of medical records among qualified healthcare providers, ensuring patient confidentiality. This approach streamlines patient care by offering a consolidated perspective of the patient's medical history.

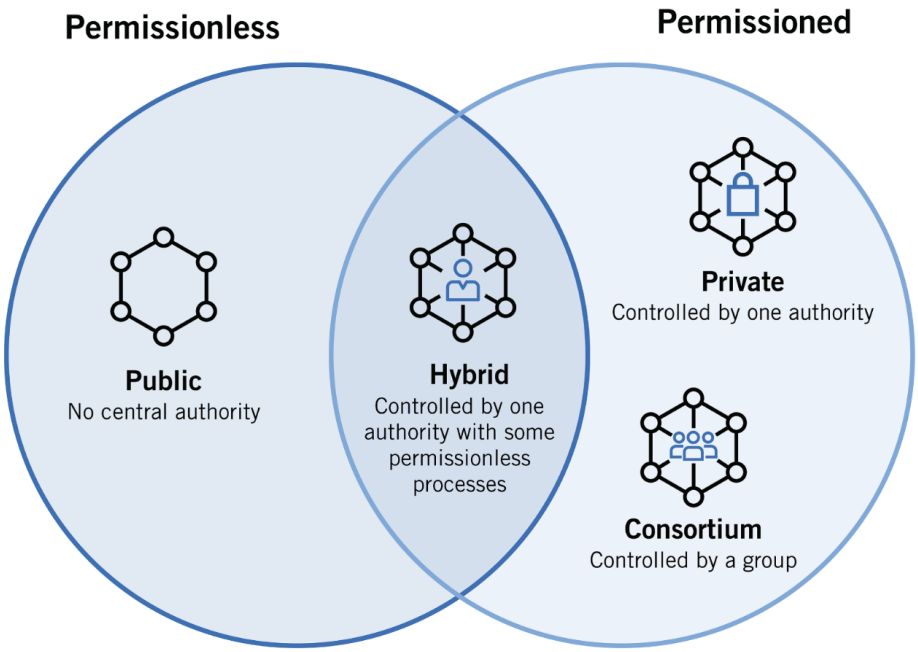
4. **Consent Management:** Blockchain can help manage patient consent for data sharing and treatment decisions. Patients can provide explicit consent for specific data access, enhancing transparency, and patient control.

5. **Decentralization of Data:** Traditional HMIS might involve centralized databases vulnerable to data breaches. Blockchain decentralized nature can distribute data across the network, making it more resistant to hacking attempts.

Figure 1. Blockchain structure.



1.3.1 Types of blockchain networks



- **Public Blockchains:** These blockchains operate as open and unrestricted networks where participation, ledger access, and transaction validation are available to all without specific permissions. They are highly decentralized, relying on a large number of nodes to secure the network. Examples: Bitcoin and Ethereum are well-known public blockchains.

- **Private Blockchains:** Private blockchains restrict participation to a predefined group of known entities. Participants are typically vetted and authorized. These networks offer greater control and privacy but sacrifice some degree of decentralization. They are often used for internal business processes, supply chain management, and consortiums. Examples: Hyperledger Fabric and Corda.

- **Consortium Blockchains:** These are governed by a group of organizations or consortium members, who collectively make decisions about the network so these are semi-permissioned. While more controlled than public blockchains, they offer a balance between decentralization and permissioned access. Ideal for industries or groups where multiple entities need to collaborate while maintaining some level of trust. Examples: Quorum and B3i.

- **Hybrid Blockchains:** Hybrid blockchains combine elements of both public and private blockchains. Some parts of the ledger may be public, while others are private. They are suitable for scenarios where transparency and privacy must coexist, such as in supply chain tracking. Examples: Dragonchain and VeChain.

CHAPTER 2

MTP PHASE 1

2.1 HYPERLEDGER FABRIC

2.1.1 About HLF

- HLF, an open-source initiative, was developed to facilitate the application of blockchain technology in business environments that demand stringent standards for privacy, confidentiality, and a structure driven by organizational needs.

- Hyperledger Fabric differs from other blockchain platforms, such as Ethereum, because it is a permissioned and private system. This permissioned nature of Hyperledger Fabric ensures the privacy and integrity of data and guarantees its confidentiality and separates it from public networks where the ledger is accessible to all and transactions are accepted by all participants. Unlike open and permissionless systems where participants with anonymous identities are allowed and transactions are validated through mechanisms like "proof of work," and "proof of stake". Hyperledger Fabric operates on a network where members join via a Membership Service Provider (MSP).

- Hyperledger Fabric provides the feature of establishing channels, enabling a set of members to maintain a distinct transaction ledger. This functionality is particularly crucial in scenarios where certain participants, potentially rivals in business, prefer to keep their transactions confidential from each other, such as offering exclusive prices to select members. In situations where a channel is formed between two members, only these specific members, and no one else, will possess the ledger records pertinent to that channel.

2.1.2 HLF Components

Peers

Peers represent the nodes within the network tasked with preserving a version of the ledger and running the chaincode. These peers are crucial for validating transactions, keeping up-to-date ledger records, and engaging in the consensus process, making them an essential element of the network's infrastructure.

Chaincode

In its documentation, Hyperledger Fabric (HLF) makes a distinction between smart contracts and chaincode, although the terms are often used interchangeably. Smart contracts refer to a set of functions that execute the agreed-upon terms among the parties involved in a transaction. Chaincode, in contrast, encapsulates a group of smart contracts. It represents the package that gets deployed to a channel. When multiple smart contracts are combined into one chaincode and this chaincode is executed, it makes all contained smart contracts accessible to applications.

Ledger

A ledger that chronologically documents every transaction, maintaining an unalterable and transparent record of all activities within the network. The blockchain serves as this ledger, recording every transaction. It offers an unchangeable historical account of all transactions, thereby guaranteeing the integrity of the data.

Channel

Channels are used to partition the network into private sections. Each channel has its own ledger and only includes specific network participants. This allows for data isolation and privacy.

Endorsement Policy

The endorsement policy specifies which channel members, belonging to which organizations, must validate or sanction transactions made by smart contracts. Should the transaction fail to receive the necessary endorsements, it will be declined.

Ordering node

An orderer represents an individual node in the ordering service. Several orderers collaborate to constitute the entire ordering service. These orderers are tasked with gathering transactions from various channels and organizing them into blocks. Subsequently, these blocks are disseminated to peer nodes, leading to updates in their

respective ledgers. Essentially, the primary function of orderers is to oversee the arrangement and verification of transactions prior to their incorporation into the blockchain ledger, thereby guaranteeing the orderly and consistent processing of transactions.

Ordering Service

The ordering service receives the endorsed transactions from clients, sequences them into blocks, and subsequently distributes these blocks to the peer nodes. This service consists of several orderer nodes collaborating to ensure a decentralized, resilient system. It is important to note that the ordering service does not engage in transaction processing or smart contract execution. Instead, its primary role is to establish the sequence of transactions and bundle them into blocks.

Identity

In Hyperledger Fabric, an identity represents a unique entity that participates in the network, such as users, peers, or any component that interacts with the blockchain. An identity is associated with a unique cryptographic certificate issued by a trusted Certificate Authority (CA). The certificate is used to sign transactions and messages, ensuring the authenticity and integrity of communications within the network. An illustration of this scenario is that certain card terminals might exclusively support Visa, while others exclusively support American Express. In contrast, there are terminals that accept both American Express and Visa cards.

Membership Service Provider (MSP)

MSP oversees the management of identities, certificates, and associated details. It is responsible for tasks such as certifying the validity of certificates, overseeing revocation lists, and allocating roles to various identities, thereby ensuring a robust and secure system for managing identities. The MSP sets the criteria for how identities are verified, authenticated, and granted permission to access the network.

Certificate Authority

In a blockchain network, a node can partake through a digital identity assigned by a trusted authority within the system. This authority, known as the Certification Authority (CA), provides X.509 certificates to those participating in the network, guaranteeing secure authentication and encrypted exchanges. The CA plays a crucial role in verifying the identity and reliability of the network's participants, thus facilitating secure and trustworthy interactions.

2.1.3 Transaction lifecycle

Transaction Proposal

The transaction begins with a proposal from a client application. This proposal specifies the desired changes to the ledger and includes endorsements from endorsers who verify that the transaction is valid.

Endorsement

The proposal is sent to endorsing peers (typically a subset of the network) who simulate the transaction, validate it, and attach their digital signatures if the transaction is valid.

Ordering

Endorsed transactions are collected by orderer nodes to form a block. The orderer nodes arrange transactions in the desired order and create blocks. Each block contains a set of ordered transactions.

Consensus

The orderer nodes broadcast the blocks to all peers. Peers validate the transactions within the block to ensure they comply with the rules defined by the smart contracts. Consensus is achieved among peers regarding the validity of transactions.

Commitment Once a consensus has been achieved, the peers add the block to their individual versions of the ledger. At this point, the transaction is deemed complete and

becomes a permanent record in the ledger's historical data.

State Update

The execution of valid transactions results in updates to the state database. These changes are recorded in the world state, reflecting the new state of assets or data.

Notification

Once a transaction is successfully committed, events or notifications can be generated to inform client applications about the transaction's status.

2.1.4 Chaincode lifecycle

Install

The chaincode is packaged into a format that can be deployed to peers on the network. Each organisation that wishes to use the chaincode installs it on their peers.

Approve for Deployment

In a permissioned network, a chaincode must be approved by a set of organisations before it can be deployed to the network.

Commit Chaincode

Following its approval, the chaincode is then deployed to the channel. At this stage, the smart contract is activated and becomes functional. After being deployed, it can be initiated and start running on the network.

Instantiate

The instantiation stage creates an instance of the chaincode on a specific channel, allowing it to be used. This process can involve providing parameters or configuration specific to the chaincode. After instantiation, the chaincode is available for clients to interact with.

Invoke and Query

Once the chaincode is instantiated, clients can send transactions to invoke the chaincode's functions and query its state. These transactions are processed, and changes to the ledger are recorded.

Upgrade

Over time, you may need to update the chaincode to introduce new features, fix bugs, or make other improvements. The upgrade process involves installing a new version of the chaincode, approving it for deployment, and then upgrading the existing chaincode on the channel.

2.1.5 Consensus Mechanism

- Hyperledger Fabric employs the Practical Byzantine Fault Tolerance (pBFT) as its consensus mechanism. PBFT is a widely recognized protocol developed to ensure enhanced security and resilience in decentralized networks.

- Hyperledger Fabric employs Practical Byzantine Fault Tolerance (pBFT) as its consensus mechanism. To maintain system stability, pBFT requires a total of 3N + 1 nodes, with 'N' representing the maximum number of nodes that can malfunction. Consequently, for any decision to be made within the network, it necessitates the endorsement of 2N + 1 nodes. This approach allows for decentralized consensus without relying on intricate mathematical computations, as seen in methods like proof-of-stake or proof-of-work.

- PBFT guarantees consensus achievement across network nodes, even when up to a third of them may be compromised or malfunctioning, ensuring that transactions are executed in a consistent and sequential way. Hyperledger Fabric (HLF) employs pBFT as its default mechanism. Alternative choices for the ordering service include Raft and Kafka-based systems.

A} **Leader Selection:** In a PBFT-based network, a leader (also known as the primary or primary replica) is selected through a round-robin or other rotation method. The leader proposes a block of transactions to be added to the blockchain.

B} **Pre-Prepare Phase:** The leader distributes the suggested block across the network to all other nodes. Upon receipt of this block, each node conducts a validation process and, if the block is deemed valid, transmits a "pre-prepare" message as a broadcast.

C} **Prepare Phase:** When a node gets pre-prepare messages from over two-thirds of the nodes in the network, it disseminates a "prepare" message. This message signifies that the node has confirmed the legitimacy of the suggested block and is prepared to add it to the blockchain.

D} **Commit Phase:** After receiving a sufficient number of prepare messages (again, typically more than two-thirds of the network), a node broadcasts a "commit" message. The commit message signifies that the node is ready to add the block to the blockchain.

E} **Transaction Finalization:** When a node obtains confirmation messages from over two-thirds of the nodes in the network, it deems the block as fully confirmed and appends it to the blockchain. At this point, the transactions contained in that block are regarded as conclusive.

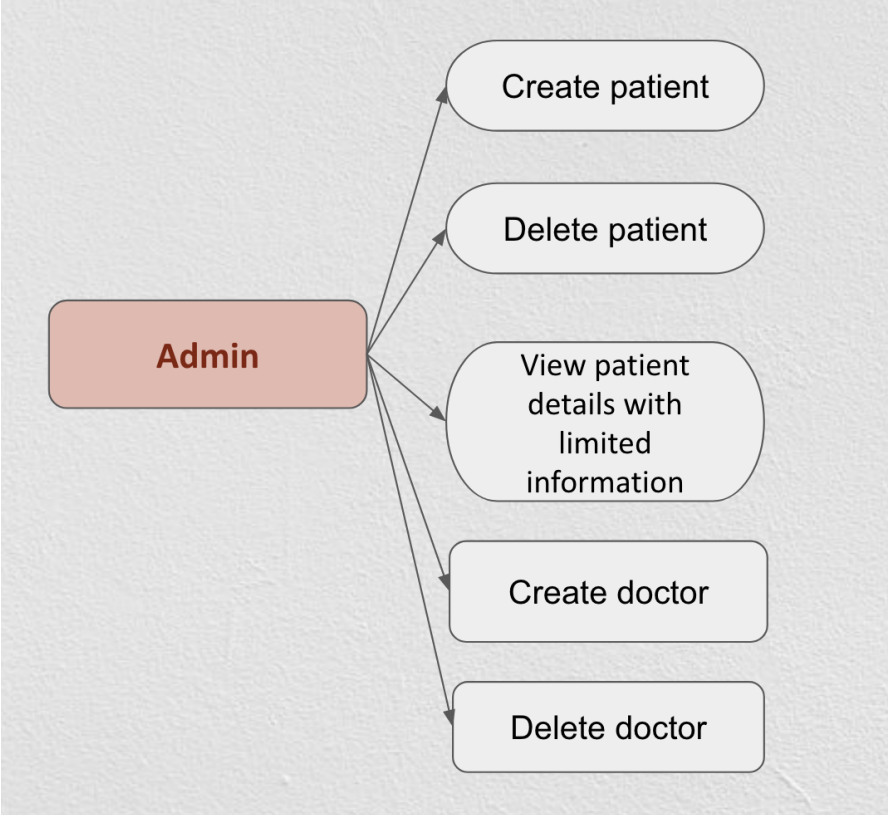
2.2 CHAINCODE

2.2.1 Smart Contracts

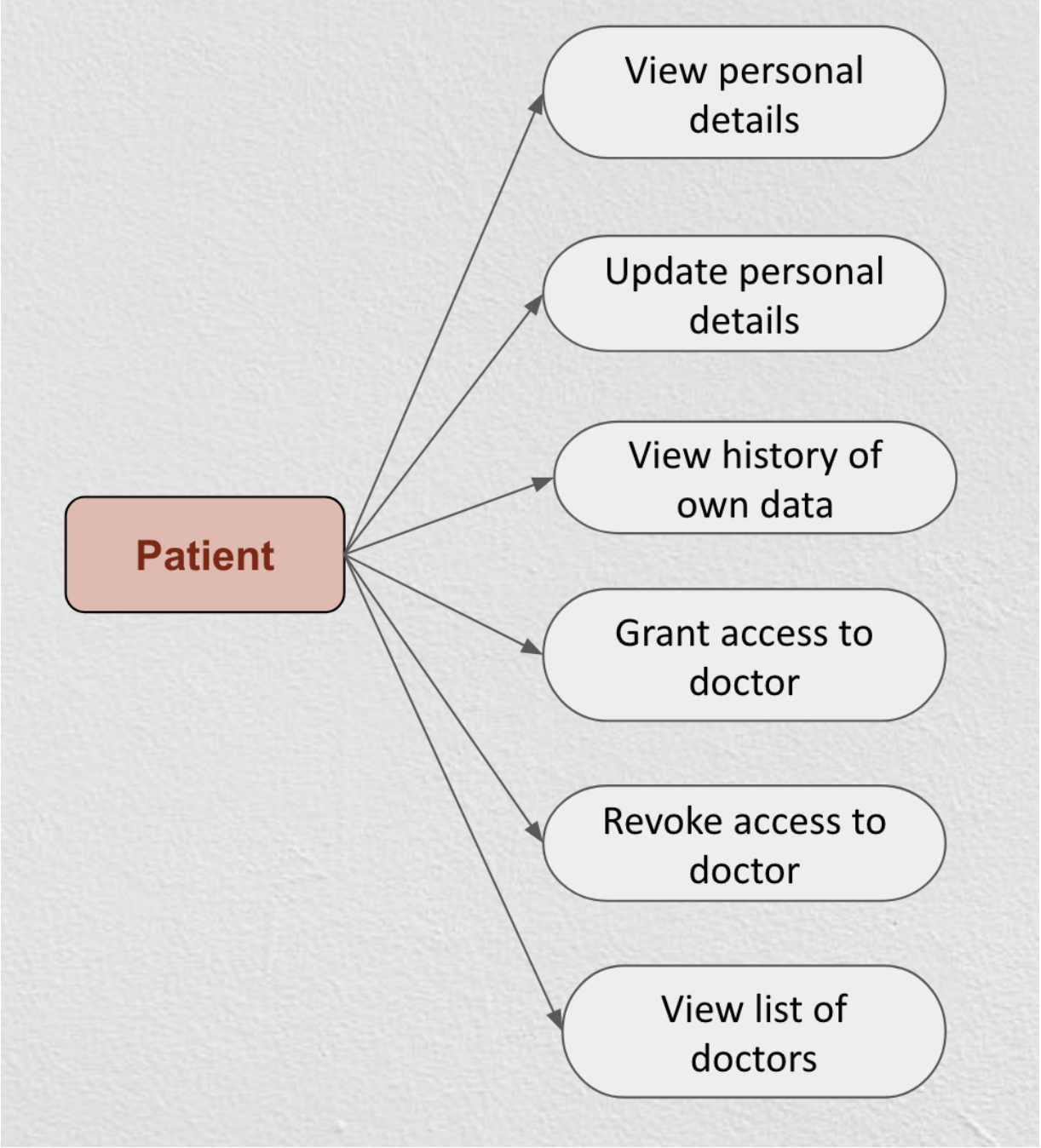
In the system, there are three distinct types of users, each assigned one of the following roles: admin, patient, and doctor. Every user category has its unique set of functionalities and use cases.

- **PrimaryContract:** This contract initiates with patient personal details, the base patient class.

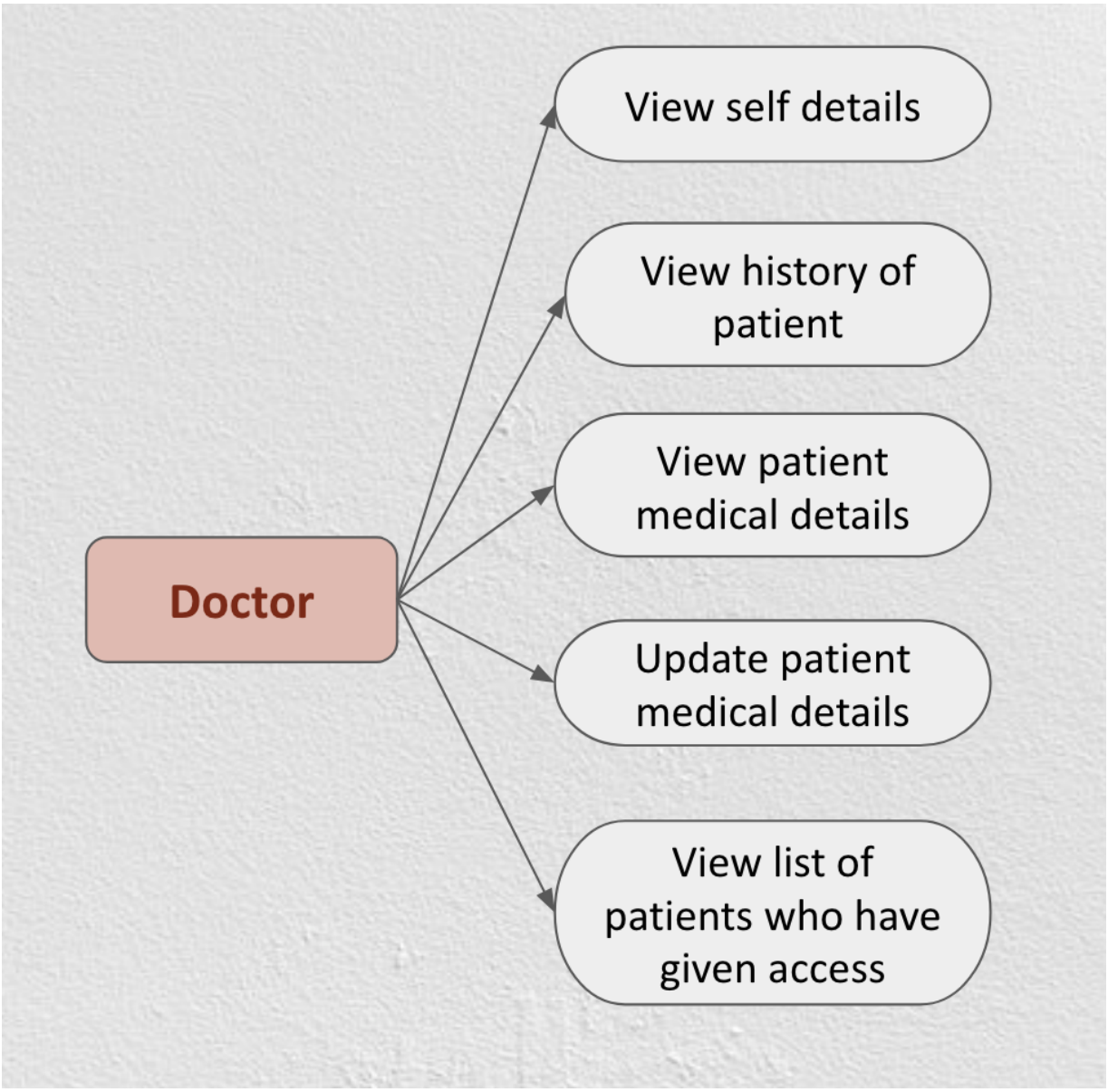
- **AdminContract:** This system possesses capabilities for generating new patient records, accessing current patient information using patient ID or full name, and removing patient entries from the ledger. Administrators have the ability to browse all patient records across the network, which improves the precision and effectiveness of data collection in the hospital's administrative processes.



- **PatientContract:** The PatientContract in the blockchain-based system focuses on patient-centered data management. It contains the logic that is required for the patient. Patients can update or view the personal details via the methods which we have defined in the contract. Patients can also manage access to their data, granting or revoking permissions to doctors, and it provides functionality for patients to retrieve their medical history, ensuring comprehensive and controlled access to personal health records.



- **DoctorContract:** The DoctorContract in the blockchain-based system is designed for healthcare professionals to interact with patient data. It allows doctors to read and update patient medical details, ensuring they have the necessary permissions for access.


Doctor
View self details
View history of patient
View patient medical details
Update patient medical details
View list of patients who have given access

2.2.2 Chaincode Implementation details

patientContract

Constructor method of the Patient class, is used to create a new patient object when an instance of the class is created. It takes several parameters representing various attributes of a patient, such as patientId, firstName lastName, password, and so on

```
class Patient{

  constructor(patientId, firstName, lastName, password, age, phoneNumber, emergPhoneNumber, address, bloodGroup,
    changedBy = '', symptoms = '', allergies = '', diagnosis = '', treatment = '', followUp = '')
    {
      this.patientId = patientId;
      this.firstName = firstName;
      this.lastName = lastName;
      this.password = crypto.createHash('sha256').update(password).digest('hex');
      this.age = age;
      this.phoneNumber = phoneNumber;
      this.emergPhoneNumber = emergPhoneNumber;
      this.address = address;
      this.bloodGroup = bloodGroup;
      this.symptoms = symptoms;
      this.allergies = allergies;
      this.diagnosis = diagnosis;
      this.treatment = treatment;
      this.followUp = followUp;
      this.permissionGranted = [];  // Initializes an empty array for storing permissions
      return this;  // It is used to return the newly created patient object
  }
}
```

- readPatient function first checks if a patient exists, and if so, it retrieves the patient's data from the blockchain ledger, parses it into a JavaScript object, and returns a modified object with selected patient attributes. If the patient does not exist, it throws an error. The provided code defines an asynchronous function named getAllPatientResults within a Hyperledger Fabric smart contract. This function is responsible for retrieving and formatting results from an iterator obtained from a query on the blockchain ledger.

- Patient Contract contains updatePatientPersonalDetails, updatePatientPassword, getPatientPassword, getPatientHistory, grantAccessToDoctor, revokeAccessToDoctor functions below are the examples of pdatePatientPersonalDetails and grantAccessToDoctor code.

```
async updatePatientPersonalDetails(ctx, args) {
    args = JSON.parse(args);
    let isDataChanged = false;
    let patientId = args.patientId;
    let newFirstname = args.firstName;
    let newLastName = args.lastName;
    let newAge = args.age;
    let updatedBy = args.changedBy;
    let newPhoneNumber = args.phoneNumber;
    let newEmergPhoneNumber = args.emergPhoneNumber;
    let newAddress = args.address;
    let newAllergies = args.allergies;

    const patient = await this.readPatient(ctx, patientId)
    if (newFirstname !== null && newFirstname !== '' && patient.firstName !== newFirstname) {
        patient.firstName = newFirstname;
        isDataChanged = true;
    }

    if (newLastName !== null && newLastName !== '' && patient.lastName !== newLastName) {
        patient.lastName = newLastName;
        isDataChanged = true;
    }

    if (newAge !== null && newAge !== '' && patient.age !== newAge) {
        patient.age = newAge;
        isDataChanged = true;
    }

    if (updatedBy !== null && updatedBy !== '') {
        patient.changedBy = updatedBy;
    }

    if (newPhoneNumber !== null && newPhoneNumber !== '' && patient.phoneNumber !== newPhoneNumber) {
        patient.phoneNumber = newPhoneNumber;
        isDataChanged = true;
    }

    if (newEmergPhoneNumber !== null && newEmergPhoneNumber !== '' && patient.emergPhoneNumber !== newEmergPhoneNumber) {
        patient.emergPhoneNumber = newEmergPhoneNumber;
        isDataChanged = true;
    }
```

```
    if (newAddress !== null && newAddress !== '' && patient.address !== newAddress) {
        patient.address = newAddress;
        isDataChanged = true;
    }

    if (newAllergies !== null && newAllergies !== '' && patient.allergies !== newAllergies) {
        patient.allergies = newAllergies;
        isDataChanged = true;
    }

    if (isDataChanged === false) return;

    const buffer = Buffer.from(JSON.stringify(patient));
    await ctx.stub.putState(patientId, buffer);
}
```

```
async grantAccessToDoctor(ctx, args) {
    args = JSON.parse(args);
    let patientId = args.patientId;
    let doctorId = args.doctorId;

    // Get the patient asset from world state
    const patient = await this.readPatient(ctx, patientId);
    // unique doctorIDs in permissionGranted
    if (!patient.permissionGranted.includes(doctorId)) {
        patient.permissionGranted.push(doctorId);
        patient.changedBy = patientId;
    }
    const buffer = Buffer.from(JSON.stringify(patient));
    // Update the ledger with updated permissionGranted
    await ctx.stub.putState(patientId, buffer);
};
```

```
async revokeAccessFromDoctor(ctx, args) {
    args = JSON.parse(args);
    let doctorId = args.doctorId;
    let patientId = args.patientId;

    // Get the patient asset from world state
    const patient = await this.readPatient(ctx, patientId);

    // Remove the doctor if existing
    if (patient.permissionGranted.includes(doctorId)) {
        patient.permissionGranted = patient.permissionGranted.filter(doctor => doctor !== doctorId);
        patient.changedBy = patientId;
    }
    const buffer = Buffer.from(JSON.stringify(patient));
    // Update the ledger with updated permissionGranted
    await ctx.stub.putState(patientId, buffer);
};
```

adminContract

It Contains createPatient, readPatient and deletePatient functions. Below is the readPatient code.

```
//Read patient details based on patientId
async readPatient(ctx, patientId) {
    let asset = await super.readPatient(ctx, patientId)

    asset = ({
        patientId: patientId,
        firstName: asset.firstName,
        lastName: asset.lastName,
        phoneNumber: asset.phoneNumber,
        emergPhoneNumber: asset.emergPhoneNumber
    });
    return asset;
}
```

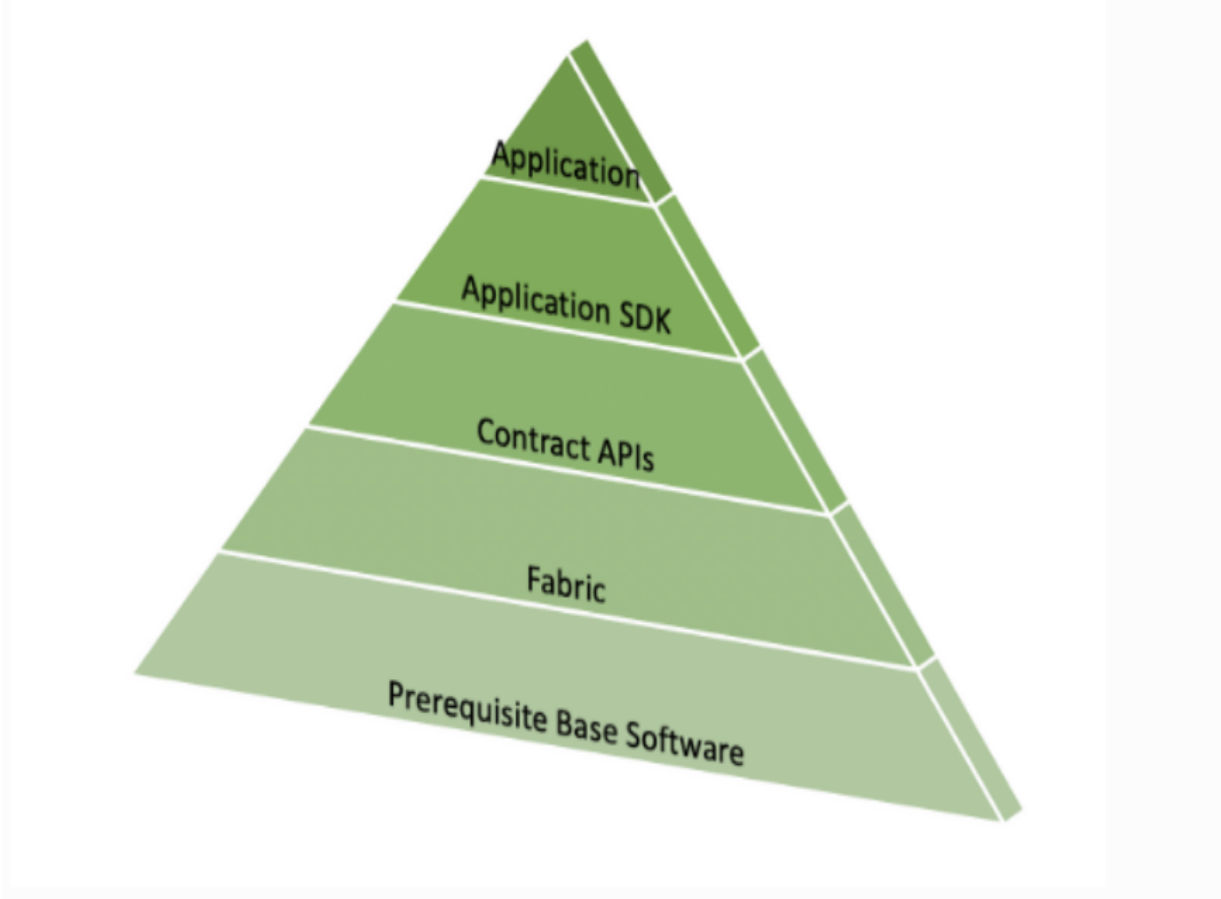
```
// Delete patient from the ledger based on patientId
async deletePatient(ctx, patientId) {
    const exists = await this.patientExists(ctx, patientId);
    if (!exists) {
        throw new Error(`The patient ${patientId} does not exist`);
    }
    await ctx.stub.deleteState(patientId);
}
```

doctorContract

It contains readPatient, updatePatientMedicalDetails, queryPatientsByLastName, queryPatientsByFirstName, getPatientHistory. Below is the code for updatePatientMedicalDetails function.

```
//This function is to update patient medical details. This function should be called by only docto
async updatePatientMedicalDetails(ctx, args) {
    args = JSON.parse(args);
    let isDataChanged = false;
    let patientId = args.patientId;
    let newSymptoms = args.symptoms;
    let newDiagnosis = args.diagnosis;
    let newTreatment = args.treatment;
    let newFollowUp = args.followUp;
    let updatedBy = args.changedBy;

    const patient = await PrimaryContract.prototype.readPatient(ctx, patientId);

    if (newSymptoms !== null && newSymptoms !== '' && patient.symptoms !== newSymptoms) {
        patient.symptoms = newSymptoms;
        isDataChanged = true;
    }

    if (newDiagnosis !== null && newDiagnosis !== '' && patient.diagnosis !== newDiagnosis) {
        patient.diagnosis = newDiagnosis;
        isDataChanged = true;
    }

    if (newTreatment !== null && newTreatment !== '' && patient.treatment !== newTreatment) {
        patient.treatment = newTreatment;
        isDataChanged = true;
    }

    if (newFollowUp !== null && newFollowUp !== '' && patient.followUp !== newFollowUp) {
        patient.followUp = newFollowUp;
        isDataChanged = true;
    }

    if (updatedBy !== null && updatedBy !== '') {
        patient.changedBy = updatedBy;
    }

    if (isDataChanged === false) return;

    const buffer = Buffer.from(JSON.stringify(patient));
    await ctx.stub.putState(patientId, buffer);
}
```

Steps to use the application



Duplicate the repository from GitHub, which is equipped with applications, resources, and configurations crucial for working with Hyperledger Fabric. In order to obtain the Docker containers and replicate the samples repository, execute the following instructions: Use ./install-fabric.sh docker samples binary to install the fabric samples, download Hyperledger Fabric Docker images, and acquire Hyperledger Fabric binaries. Following this, we can proceed to initiate the network.

- **down:** The script ./network.sh is designed to shut down the network and remove any existing artifacts or containersleft over from previous executions, essentially clearing Docker containers.

- **up:** The command ./network.sh up activates the network, establishing a fabric network comprising an Orderer node (org) and two peer nodes (org1, org2). Initially, no channel is formed, although docker containers are set up.

- **up createChannel:** ./network.sh up createChannel is used for simultaneously initiating the network and establishing a channel in one combined action.

- **deployCC:** Utilize ./network.sh deployCC -ccn basic -ccp ../asset-transfer-basic/chaincode-go -ccl go to initiate a chaincode on a channel, typically defaulting to asset-transfer-basic. Once a channel is established, you're then able to begin utilizing smart contracts for interactions with the channel's ledger.

Flowchart

Here's a basic flowchart to illustrate the process:

patientClass -> primaryContact -> (adminContract or patientContract) -> doctorContract

1. **adminContract:** It manages patient information in the ledger.
    - Create a new patient (createPatient).
    - Read patient details (readPatient).
    - Delete a patient (deletePatient).
    - Retrieve all patients' details (queryAllPatients).
2. **doctorContract:** Inherits from adminContract. It manages medical aspects of patient information.
    - Read patient details, with access control checks (readPatient).
    - Update patient medical details (updatePatientMedicalDetails).
    - Retrieve patients' medical history (getPatientHistory).
    - Query patients by name with specific doctor access rights.
3. **patientContract:** Inherits from PrimaryContract. It allows patients to manage their personal and medical information.
    - Read personal patient details (readPatient).
    - Update personal details (updatePatientPersonalDetails).
    - Retrieve personal medical history (getPatientHistory).
    - Grant or revoke access to doctors (grantAccessToDoctor, revokeAccessFromDoctor).

Smart Contracts Hierarchy

2.3 SECURITY

How using blockchain provides better security for this attack as compared to a traditional system:

1. **Decentralization:** Blockchain functions through a network of nodes that are decentralized. In contrast to centralized systems, which are vulnerable at a single failure point, compromising a blockchain system would require overpowering a substantial part of its network. This dispersed structure of blockchain increases the difficulty for attackers to focus on and incapacitate the entire system.

2. **Data Immutability:** When information is logged onto a blockchain, changing or erasing it becomes exceedingly difficult without consensus from the majority of the network's participants. Therefore, even in the event of an attack, the accuracy and consistency of the data are preserved.

3. **Consensus Mechanisms:** Blockchains utilize consensus protocols to authenticate and establish the sequence of transactions. These protocols play a crucial role in

safeguarding against the inclusion of illegitimate or harmful transactions within the blockchain, guarding against any attempts by attackers to overwhelm the network with a deluge of transactions.

4. **Cryptography:** Blockchain uses cryptographic techniques to secure data and transactions. These techniques make it extremely difficult for attackers to intercept, manipulate, or forge data during an attack.

5. **Smart Contracts:** Several blockchain platforms offer support for smart contracts, enabling the automation of specific security protocols. For instance, in the event of unusual network activity, smart contracts have the capability to momentarily restrict access or regulate inbound traffic, thereby lessening the severity of a DDoS attack.

6. **Transparent Transactions:** Blockchain's transparency means that all transactions are visible to network participants. This transparency can help in identifying and isolating malicious traffic patterns during an attack.

7. **Access Control:** Systems based on blockchain technology can establish authentication protocols and robust access control, ensuring network access is limited to duly authorized users. This approach effectively minimizes the potential attack surface by barring unapproved access.

8. **Data Distribution:** In blockchain networks, data is distributed across multiple nodes. Even if some nodes become inaccessible during an attack, other nodes can continue to operate and provide access to data and services.

9. **Redundancy:** Many blockchain networks have data redundancy across multiple nodes. Even if some nodes are compromised by ransomware, other nodes can continue to operate and provide access to data and services.

2.4 SECURITY AGAINST ATTACKS

2.4.1 Ransomware Attack

- Ransomware represents a form of a malware that locks and encrypts the files or systems of its victims, making them unusable. The perpetrators then ask for a ransom, usually in the form of cryptocurrency, to provide the decryption key needed to regain access to the locked files or systems.

- **Delivery Methods:** Ransomware may be distributed via harmful attachments, deceptive phishing emails, compromised websites, or through flaws in software systems.

Challenges:

- **Data Accessibility:** Ransomware can encrypt data, making it inaccessible to healthcare providers. This can have severe consequences, especially during emergencies.
- **Data Integrity:** While blockchain provides data integrity, ransomware can potentially exploit vulnerabilities in smart contracts or other blockchain components to manipulate data.
- **Decryption Key Security:** Securing the decryption keys is crucial. If attackers gain access to these keys, they can decrypt and potentially alter sensitive health data.
- **System Downtime:** Ransomware can cause system downtime, impacting hospital operations and patient care.
- **Data Backup and Recovery:** Ensuring that data backups are secure and up-to-date is essential to recover from a ransomware attack. However, managing backups in a decentralized blockchain system can be challenging.

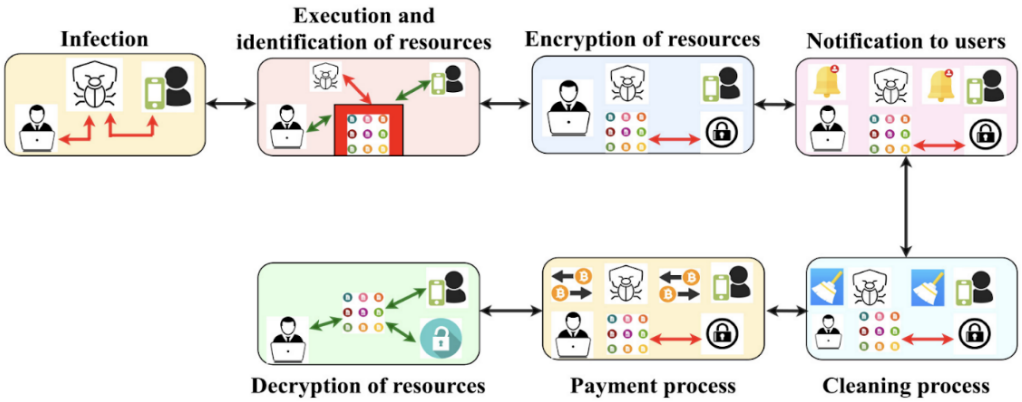


Fig. 1. Working mechanisms of ransomware (Source: [10]).

Fig. 2. Blockchain enabled generic ransomware defense system.

Existing Security Mechanism:

1. **Data Encryption:** Implement encryption for data whether it is stored (at rest) or being transferred (in transit) to safeguard it against unauthorized access.

2. **Immutable Ledger:** The immutable nature of the blockchain's ledger means that once information is logged, it is permanent and cannot be altered or erased. This provides a safeguard against ransomware attacks that aim to encrypt or alter data.

3. **Regular Backups:** Regular backups of data, stored in a secure and decentralized manner, help ensure data recovery in case of a ransomware attack.

4. **Smart Contracts:** Use smart contracts to automate responses to ransomware attacks, such as isolating infected systems or revoking access.

5. **Secure Key Management:** Secure management of cryptographic keys used for encryption and decryption is crucial to prevent against unauthorized access.

6. **Network Security:** Establish robust network security protocols, including intrusion detection systems, the deployment of firewalls, and encrypted communication channels, to block the entry of ransomware.

Better Solutions:

In order to thwart an external assault, wherein the Ransomware perpetrator endeavors to transmit a transaction containing the Ransomware attack link to encrypt the MHR outside the network, all phases of the external attack algorithm are detailed below.

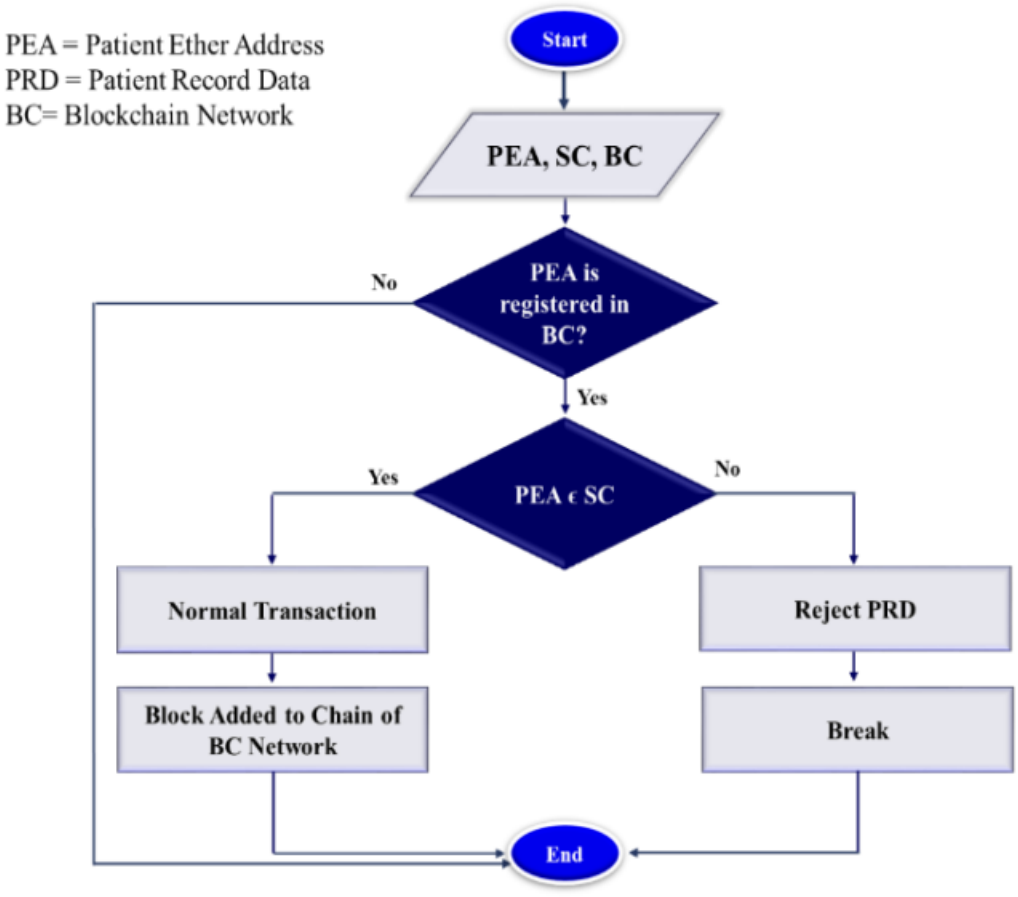


Fig. 3. Flowchart of External Ransomware Attack

- After the Medical Health Record (MHR) is uploaded to the application in the Hyperledger Fabric network, the smart contract initiates the verification process. It

carefully examines the sender's address to confirm its registration in the Blockchain network. It's worth noting that all addresses associated with patients in the private Blockchain network can be easily identified. If the smart contract successfully validates the sender's address as legitimate, it proceeds to verify the requested transaction, ensuring it contains data and is not empty. Once the validation is successful, the transaction is approved and executed. Conversely, if the smart contract cannot verify the sender's address as registered, it takes actions to block the transaction request, preventing any further processing.

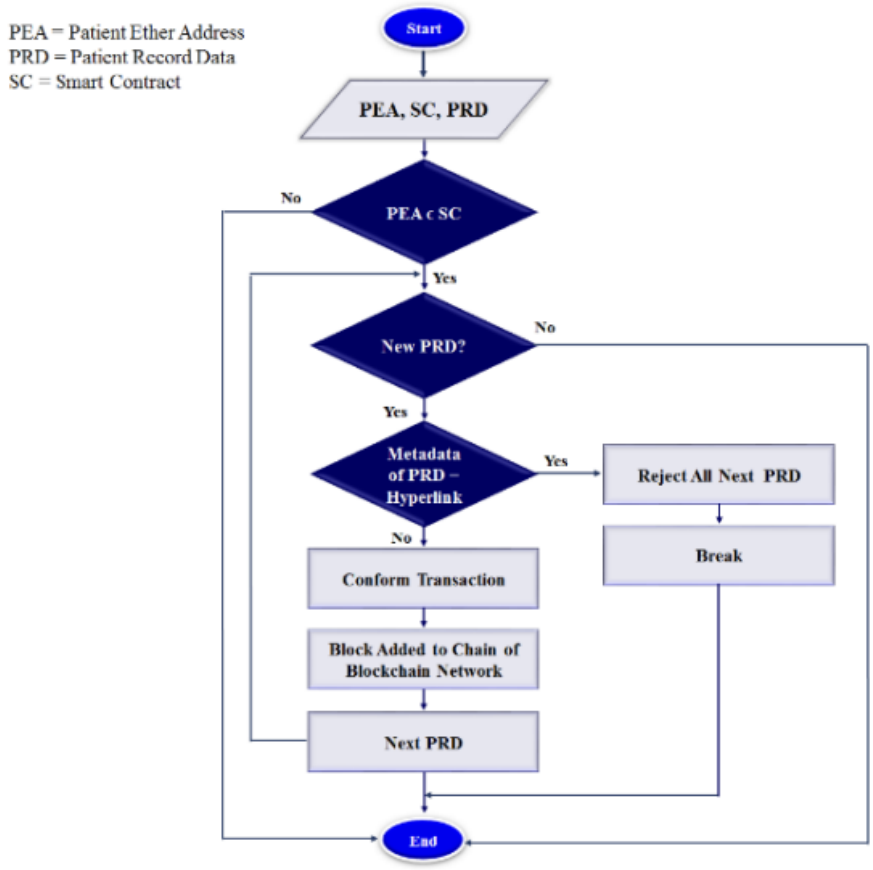


Fig. 4. Flowchart of Internal Attack

- Following the upload of the Medical Health Record (MHR) to the Hyperledger Fabric application, the smart contract initiates a verification process. Initially, it verifies whether the sender's identity and permissions are registered within the Hyperledger Fabric network, as all participant identities in the private network are already well-known. If the sender's identity is recognized and holds the necessary permissions, the smart contract proceeds with the validation of the requested transaction. This validation encompasses confirming that the transaction is not empty and complies with any predefined data format or content restrictions set by the network's policies. Transactions failing to meet these criteria are declined by the smart contract. In instances where the transaction is incorrect or unauthorized, the smart contract takes actions to prevent further transactions from that sender

and may activate specific measures outlined in the network's governance rules or policies.

2.4.2 Network Sniffing

Network sniffing, also known as packet sniffing or network packet analysis, is a type of cyber attack that involves intercepting and inspecting the data packets flowing through a computer network. In the context of a blockchain-based hospital management information system, network sniffing can pose several security risks and privacy concerns. Here's how network sniffing can impact such a system:

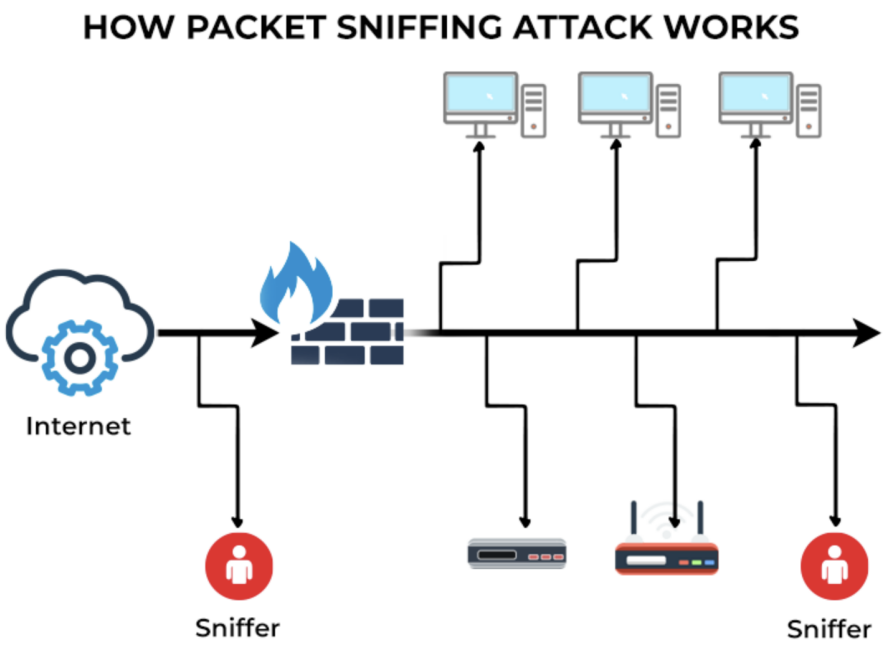
- **Data Interception:** Network sniffers monitor and scrutinize data packets moving through the network. If the hospital management system's communication channels lack adequate encryption, this could lead to a situation where an intruder might capture confidential patient information, personal data, encompassing medical histories, and financial specifics.

- **Privacy Violation:** Sniffing attacks pose a significant threat to privacy by potentially allowing unauthorized individuals to obtain patients' private medical records, treatment strategies, and other confidential data. Such breaches can lead to the illicit acquisition of personal health information (PHI), raising significant legal and ethical issues.

- **Credential Theft:** Attackers may capture login credentials (such as usernames and passwords) sent in plaintext if the system lacks encryption. Once obtained, these credentials can be used for unauthorized access to the hospital management system or other related systems.

- **Data Manipulation:** In addition to intercepting data, attackers with network access can modify or inject malicious data packets into the network stream. This could lead to false medical records, incorrect diagnoses, or altered treatment plans.

- **Distributed Denial of Service (DDoS):** Network sniffing can be used to identify network weaknesses, which could be exploited to launch DDoS attacks on the blockchain-based system, causing network disruption and downtime.



Challenges:

- **Data Sensitivity:** Hospital management information systems handle highly sensitive and personal health information, which, if intercepted, can have severe consequences for patient privacy and data security.
- **Blockchain Vulnerabilities:** While blockchain is considered secure, it is not completely immune to attacks. If an attacker can intercept network traffic, they may be able to exploit vulnerabilities in the blockchain protocol or smart contracts.
- **Network Security:** Hospitals and healthcare organizations may not have robust network security measures in place, making them vulnerable to network sniffing attacks.
- **Encryption Limitations:** While data on a blockchain is typically encrypted, data in transit may not always be fully secured, providing opportunities for sniffing attacks.

Existing Security Mechanism:

1. **Encryption:** Utilizing robust encryption protocols like HTTPS and SSL/TLS for data in transit guarantees that intercepted data remains challenging to decipher or manipulate.
2. **Secure Wireless Networks:** Employing secure wireless protocols like WPA3 to safeguard wireless networks against unauthorized entry and eavesdropping

attempts.

3. **VPN (Virtual Private Network):** Using VPNs to create secure communication channels that can protect data from being intercepted during transmission.

4. **Blockchain Security:** Leveraging the built-in security attributes of blockchain, such as cryptographic hashing, digital signatures, and agreement mechanisms, to guarantee the integrity and legitimacy of data.

5. **Secure Channel for Key Exchange:** Ensuring secure channels for exchanging cryptographic keys, which are used for encryption and decryption, to prevent key interception.

Better Solutions:

- Use fixed node identities and uphold a secure peer roster to deter attackers from introducing counterfeit nodes into the network. Introduce robust access control measures within the blockchain network to limit entry to authorized individuals exclusively. Evaluate the adoption of IPv6 in place of IPv4 to bolster network security and diminish vulnerability to address-centric attacks.

- Utilize secure communication protocols such as HTTPS and encrypted channels to safeguard blockchain-related information, encompassing usernames and passwords. Opt for a switched network design instead of hubs to guarantee that data is exclusively directed to its intended recipients within the blockchain network.

- To enhance security within the blockchain environment, introduce password-based authentication for shared directories and services to thwart unauthorized entry.

2.4.3 DDOS Attack

A DDoS (distributed denial of service) attack is a type of attack that attempts to overwhelm a network by flooding it with a large amount of traffic. The goal of this type of attack is to prevent the HMIS from being accessible to its target users, which is usually healthcare providers or patients. A DDoS attack can happen in a number of ways, but one of the most common is a DDoS attack on a Blockchain-based Hospital. Here's how it can happen in a Blockchain-Based HMIS:

- Overloading Blockchain Nodes: Attackers may target the blockchain nodes that are part of the HMIS, attempting to overwhelm them with traffic in order to prevent them from processing and validating transactions.

- Targeting Network Infrastructure: Attackers may also target the underlying network infrastructure of the HMIS, such as servers and data centers, with the goal of overloading them and causing a system outage.

- Exploiting Smart Contracts: If the HMIS utilizes smart contracts, attackers could exploit vulnerabilities in the smart contracts to create conditions that lead to a DDoS attack.

A DDoS (Distributed Denial of Service) attack on a blockchain-based Hospital Management Information System (HMIS) could have serious consequences, potentially hindering healthcare providers' access to crucial patient data, thereby jeopardizing patient treatment. Such an attack might also lead to financial repercussions and harm the reputation of the medical facility. To counteract the threat of DDoS attacks, it is essential to deploy strong cybersecurity strategies, including the use of firewalls, systems for detecting and preventing intrusions, and services specifically designed for DDoS attack mitigation.

Figure 1. Distributed denial of service (DDoS) attack [6].

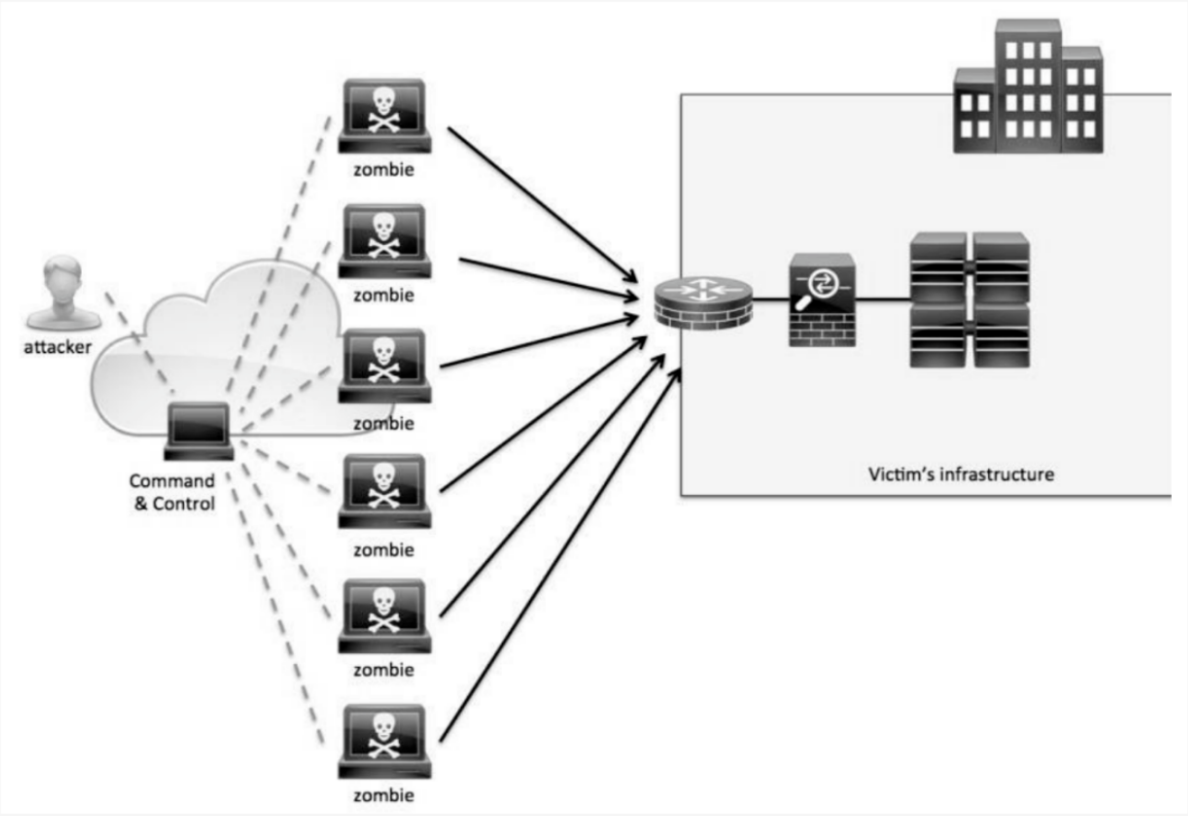
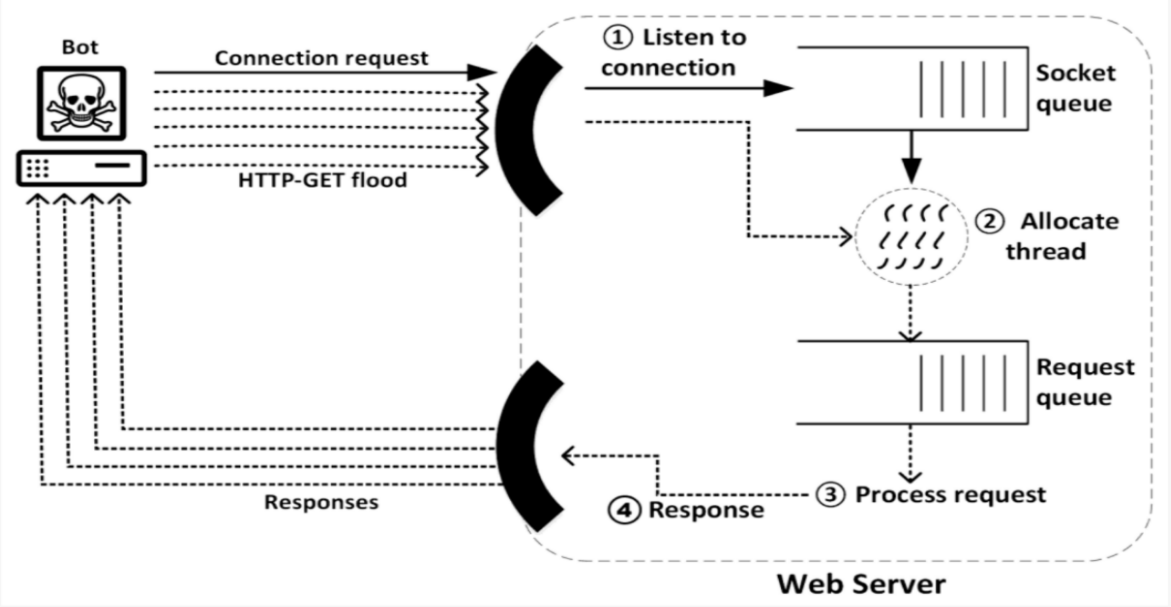


Figure 2. Application layer DDoS attack example [42].



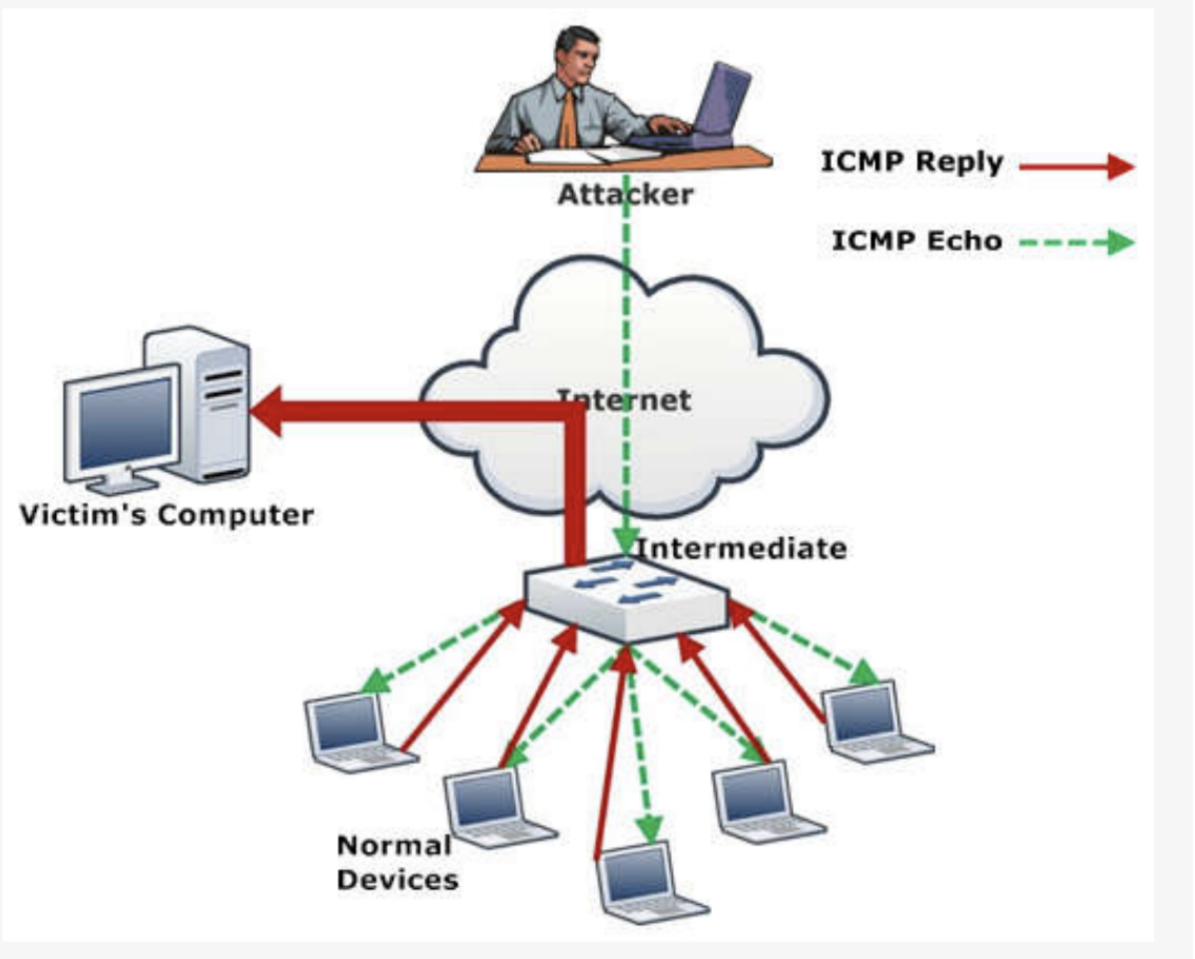
Challenges:

- **Network Overload:** A DDoS attack involves overwhelming the network with a flood of internet traffic, which can lead to the network being overloaded and becoming slow or completely unavailable.

- **Blockchain Node Overload:** The blockchain nodes may be targeted to disrupt the validation and processing of transactions, potentially causing delays or halting the blockchain operation.

- **Smart Contract Vulnerabilities:** If smart contracts are employed in the HMIS, DDoS attacks can exploit vulnerabilities in the smart contracts to create conditions that overload the system.

- **Scalability Issues:** Blockchains typically have limitations on the number of transactions they can process per second. A DDoS attack can exploit this limitation to cause a backlog of transactions, resulting in delays in data recording and retrieval.



Existing Security Mechanism:

1. **DDoS Mitigation Services:** Utilize specialized DDoS mitigation services such as Cloudflare, Akamai, or Arbor Networks, which can detect and mitigate large-scale DDoS attacks.

2. **Web Application Firewalls (WAF):** Employ Web Application Firewalls to filter out malicious traffic and protect against application-layer DDoS attacks.

3. **Use of IDPS:** Deploy Intrusion Detection and Prevention Systems to recognize and halt DDoS attacks by detecting established patterns of such attacks.

4. **Traffic Control:** Employ methods such as rate limiting and traffic shaping to regulate network traffic flow and avert system overloads.

5. **Content Delivery Networks (CDN):** Employ Content Delivery Networks to spread inbound traffic over several servers, thereby diminishing the severity of DDoS attacks.

Better Solutions:

- Uncle blocks: The defense mechanism against Uncle blocks BDoS has been specifically tailored to counter Nakamoto consensus-based cryptocurrencies like Bitcoin, but its effectiveness varies when applied to different cryptocurrencies, including Hyperledger Fabric. This variance in effectiveness can be attributed to the inherent uncle block mechanism in Hyperledger Fabric, which provides incentives to miners who successfully mine blocks directly linked to the primary blockchain.

- This characteristic disrupts BDoS attacks because, even if a rational miner does not emerge as the winner of the mining competition, their block still receives a reward. As a result, when an attacker publishes a block header, it no longer significantly diminishes the expected profits of rational miners within the Hyperledger Fabric network.

- It's crucial to emphasize that this approach doesn't offer incentives to blocks that aren't directly linked to the primary blockchain. This leaves room for the possibility of alternative attack strategies that could potentially reduce the expected reward, such as publishing two-block headers to create a fork from the most recent block in the chain. However, conducting a detailed design and investigation of such potential attacks falls beyond the scope of this research within the context of Hyperledger Fabric.

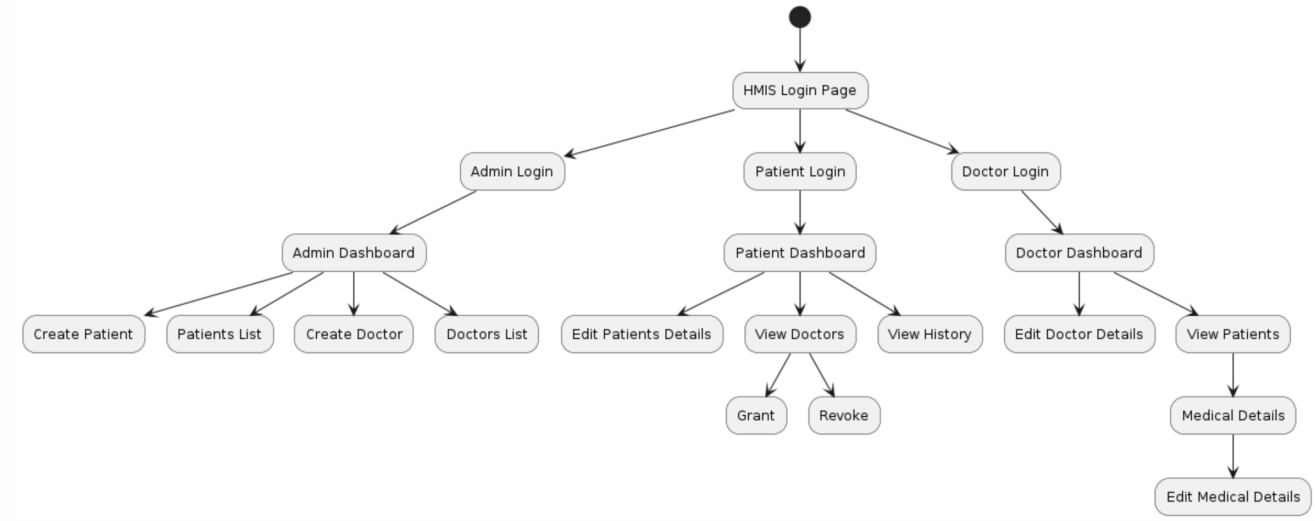
- Disregarding Attacker's Blocks in the Mining Competition: Another potential approach to mitigate the impact of the attack involves altering the behavior of miners so that, in the event of a blockchain fork, miners prioritize blocks that were

not generated by the attacker. However, the key challenge lies in distinguishing attack blocks from legitimate ones. Employing a third-party service for this purpose is not feasible as it contradicts the decentralized principles of the system and opens the door to false accusations. Instead, we propose a classification method based on the time gap between receiving the block header and receiving the complete block. In cases of non-attack blocks, we can reasonably assume that this time interval remains within a certain bound, such as one minute. Blocks with longer intervals are deemed suspicious in this context.

CHAPTER 3

UI DESIGN AND FLOW

3.1 SCREENS FLOW



First there will be a login page, We can basically login using ADMIN, DOCTOR and PATIENT roles.

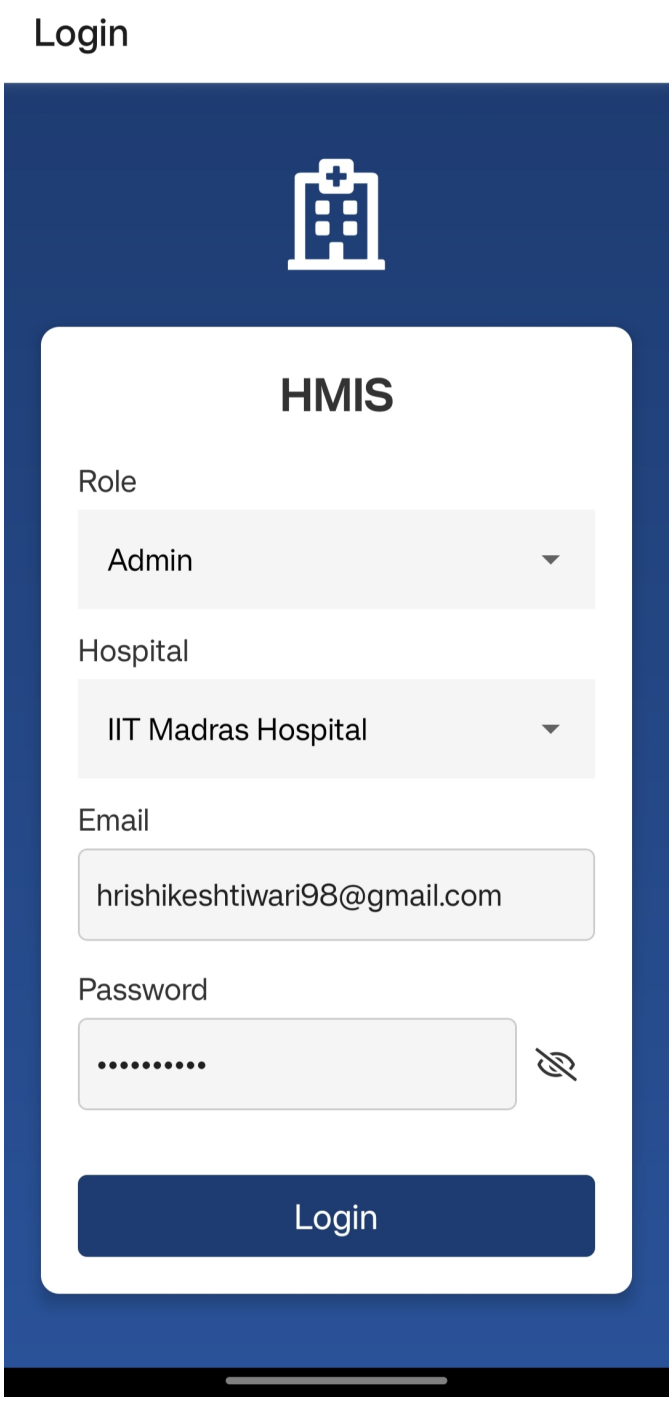
- By login using admin credentials, we have a list of all the patients and all the doctors created by the admin. We possess the capability to add a new patient and a new doctor through the "create patient" and "create doctor" functions, respectively.

- By login using patient credentials, We have “patient dashboard”, which contains “edit patient personal details”, “View patient history” and “the list of all the doctors with the function of grant or revoke access to the doctors”.

- By login using doctor credentials, We have Doctor Dashboard which contains, “Edit Doctor Details”, “View the list of patients who have given access to the doctor”, and “View medical details” Functionalities.

3.2 SCREENS

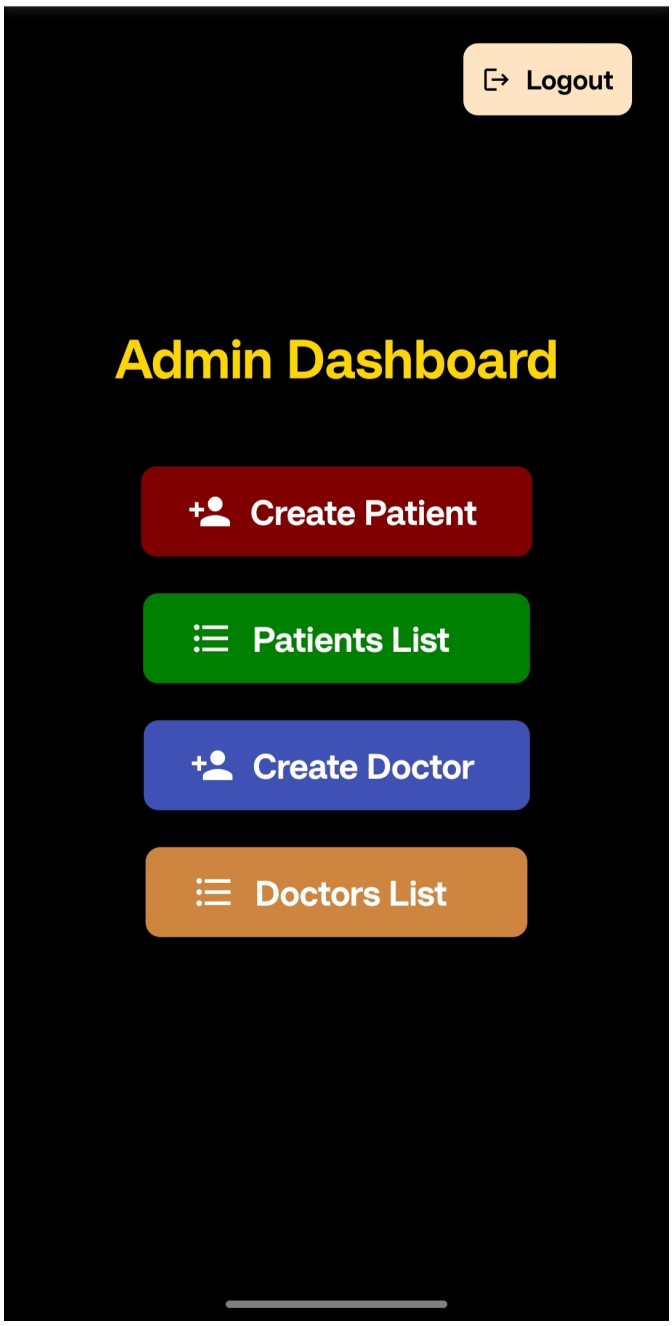
3.2.1 Admin Login



When a hospital joins the network, an admin account is created for it. During the hospital's addition to the network, the admin's details must be provided. These details,

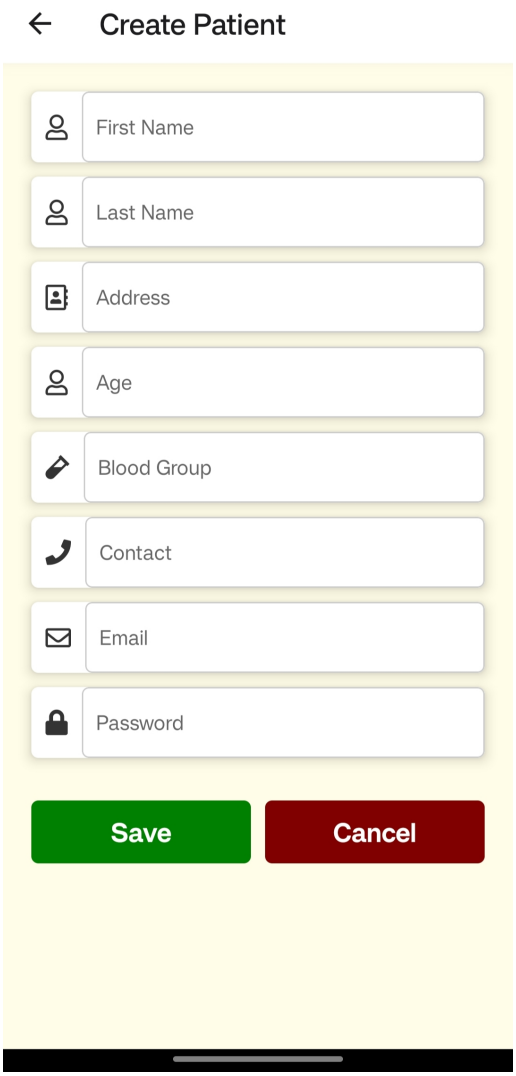
such as **email and password**, are located in the fabric-ca-server-config.yaml file under the hospital CA configuration. Admin login page is illustrated in the above figure. Users have the option to log in as an admin, doctor, or patient, but they must select the **admin** role. As the Android application is utilized by all participants, the admin is required to choose the hospital and enter their credentials. These credentials are subsequently verified against those recorded in the blockchain network, as set up in the Hospital CA config YAML file.

3.2.2 Admin Dashboard



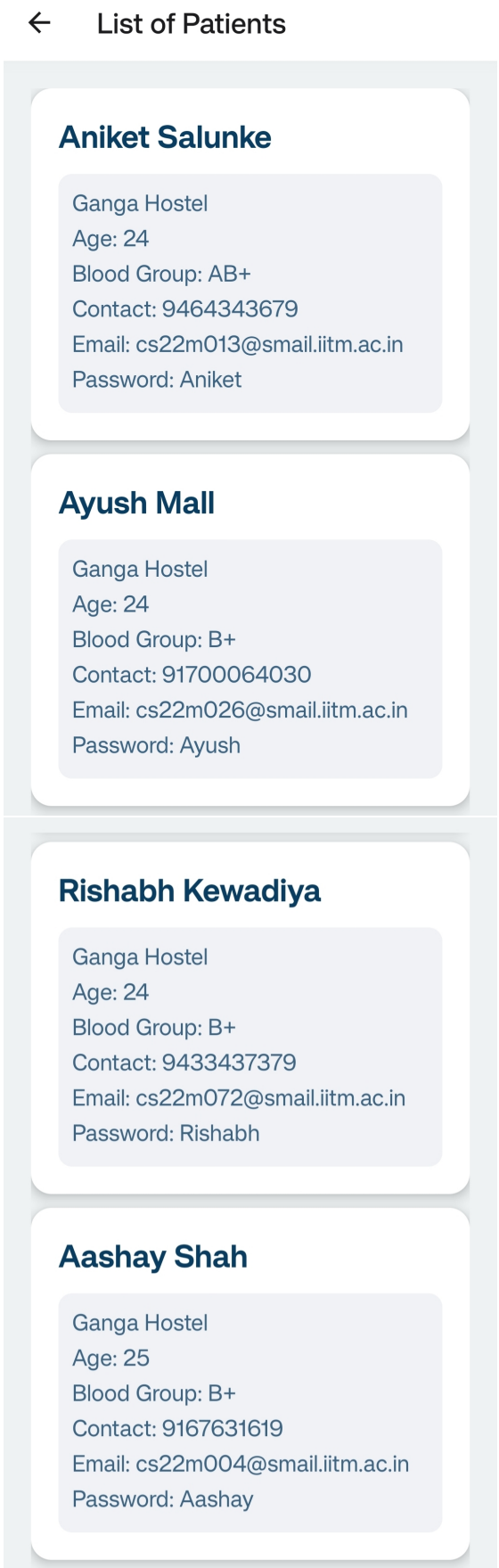
Upon successful login by the Admin, an Admin Dashboard appears, featuring options like **Create Patient**, **Patients List**, **Create Doctor**, and **Doctors List**, as illustrated in the figure above. Clicking the Logout button allows the user to exit from the page.

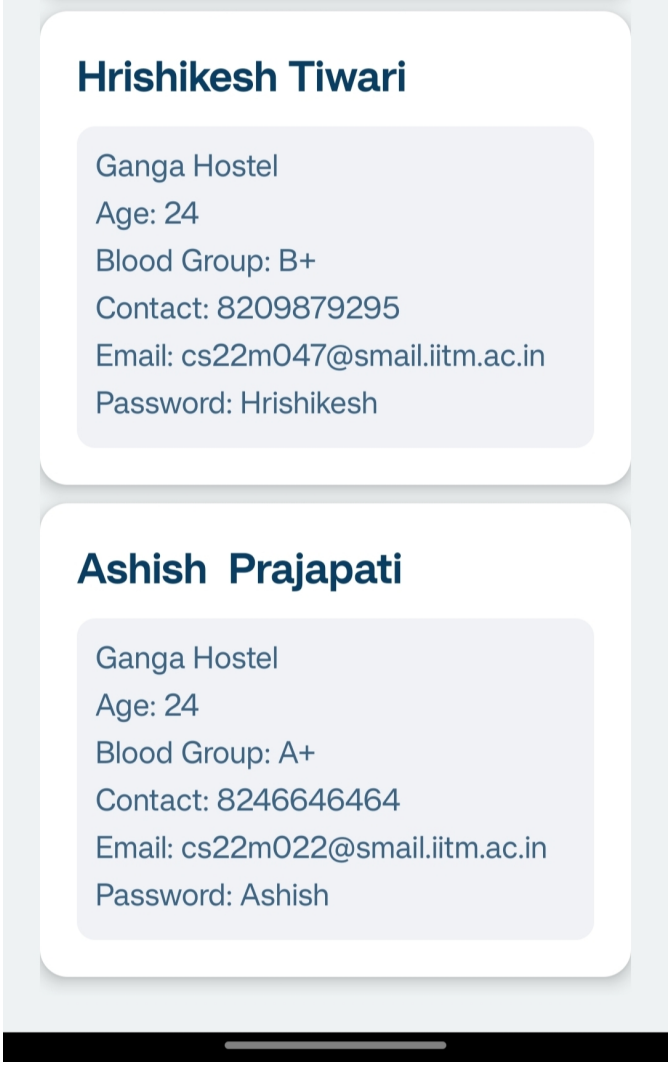
3.2.3 Create Patient



The admin can create a patient profile, which involves initiating a transaction in the ledger to add an object in the world state. The admin must input the patient's basic information as depicted in the figure above. Upon clicking **Save**, the transaction is executed, disseminated to all network peers, and the patient data undergoes validation. If the data is validated and endorsed, it is recorded in the ledger. Additionally, the patient is registered as a client in the network, enabling interaction with the ledger. Upon saving, temporary patient credentials are generated and stored in our **firebase** database. This newly created patient profile is then added to the "Patients Lists" in the ledger, making it available when querying for all patients.

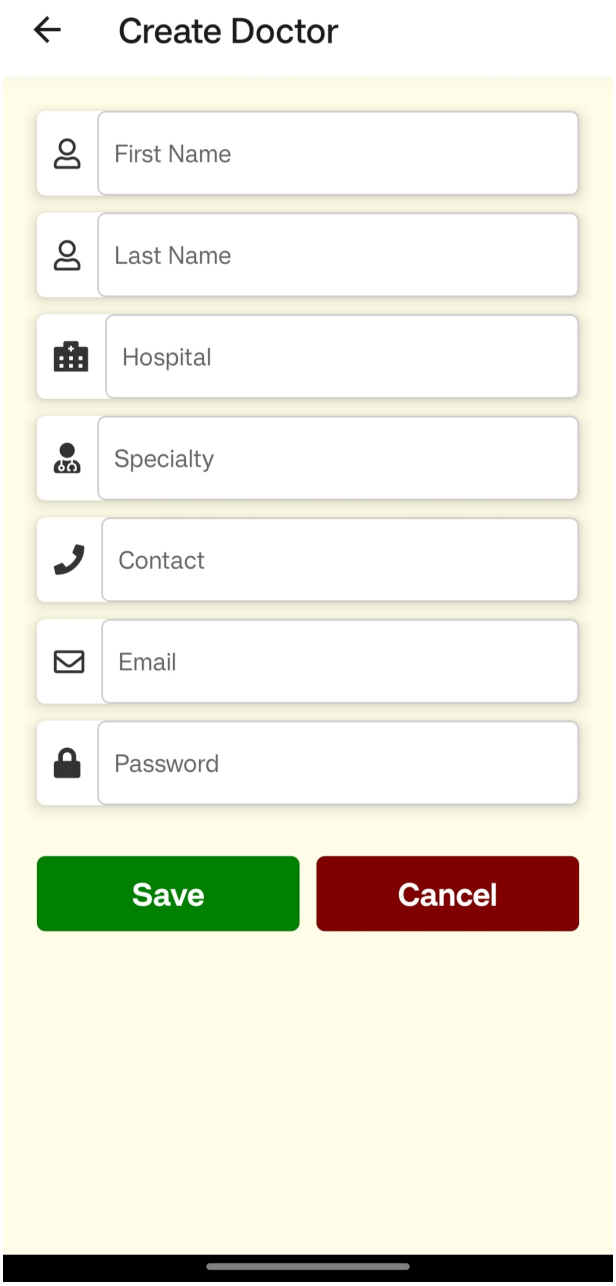
3.2.4 Patients List





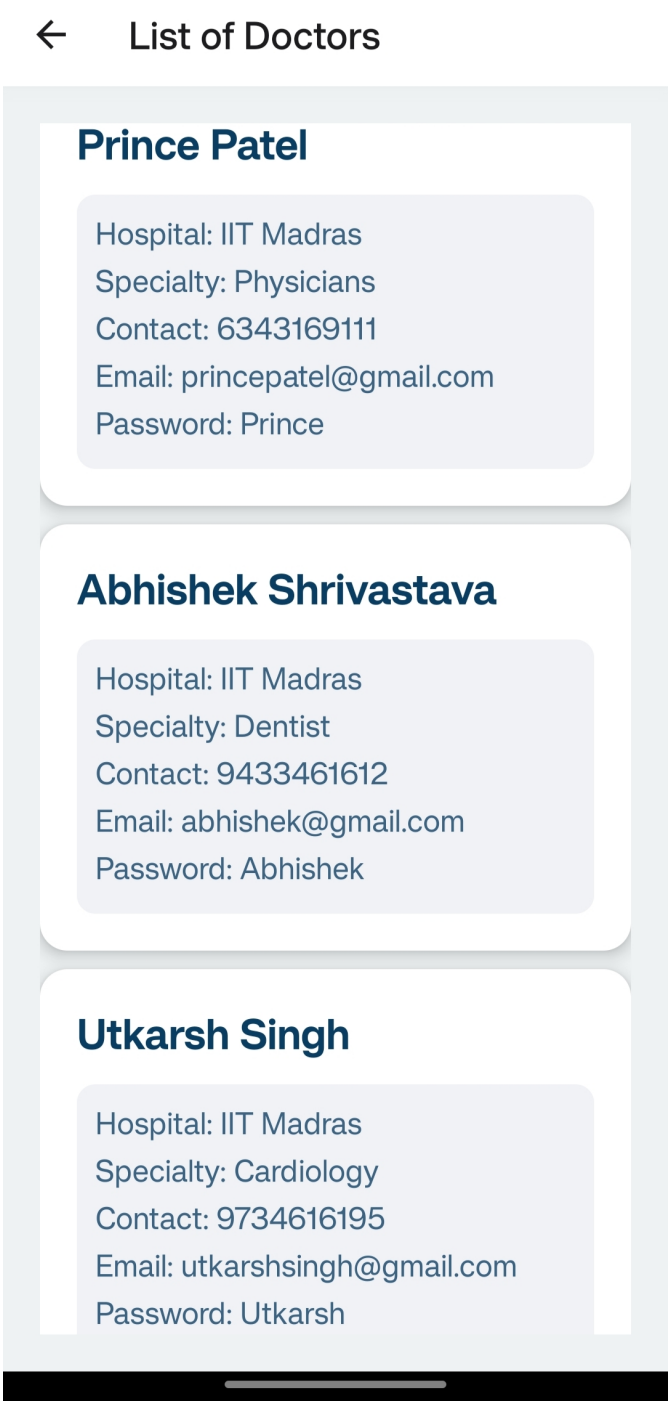
The patients we create are stored in Firebase. Once a patient is created, their information is added to the ledger under "Patients Lists," allowing their data to be accessed whenever a query for all patients is made.

3.2.5 Create Doctor



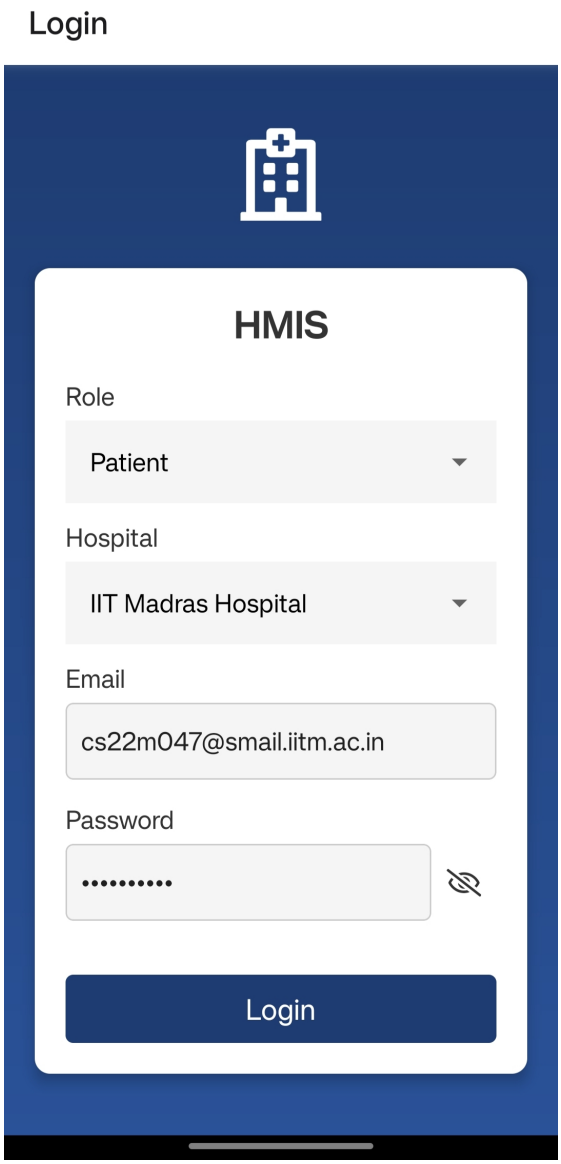
The admin has the ability to add a doctor profile. Although the doctor is not an object stored in the ledger, a client within the network is set up to enable the doctor to interact with the ledger. There is no direct interaction with the ledger when creating the doctor. The admin must provide the doctor's details as illustrated in the figure above, and upon clicking **"save,"** a client for the doctor is created within the network.

3.2.6 Doctors List



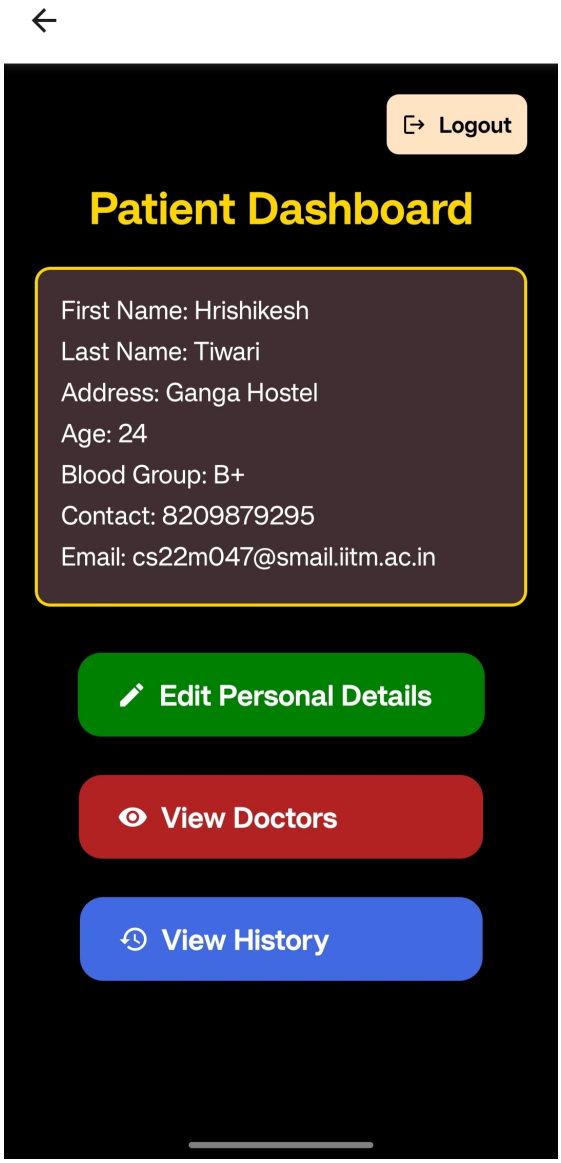
The patients we have created are stored in Firebase. After a patient is created, their details are added to the ledger under "Doctors Lists," making their information retrievable when all patients are queried again.

3.2.7 Patient Login



After a successful login, a patient can access their personal and medical details as depicted in the figure. These details are retrieved by invoking the patient contract. The patient login page is also shown in the figure. Each hospital that joins the network has patients created for it by the admin. The data for these patients is stored in the ledger and can be retrieved. The admin details, including email and password, are located in the fabric-ca-server-config.yaml file under the respective patient's CA configuration. These details are then authenticated against those recorded in the blockchain network, as configured in the Patient CA config YAML file.

3.2.8 Patient Dashboard

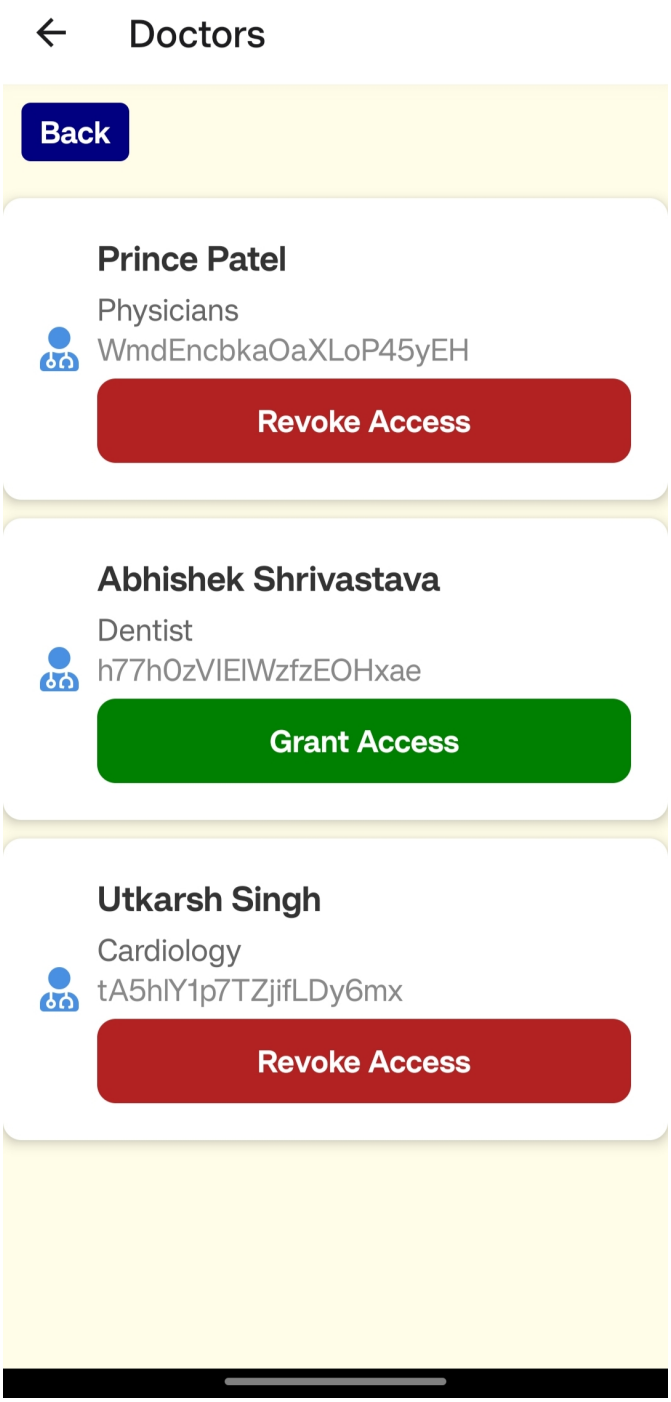


Once the patient successfully logs in, the **Patient Dashboard** appears. This dashboard includes options to **Edit Patient Details**, **View Doctors**, and **View History**, as depicted in the figure above. Clicking the **Logout** button allows the user to log out from this page.

3.2.9 Edit Patient Details

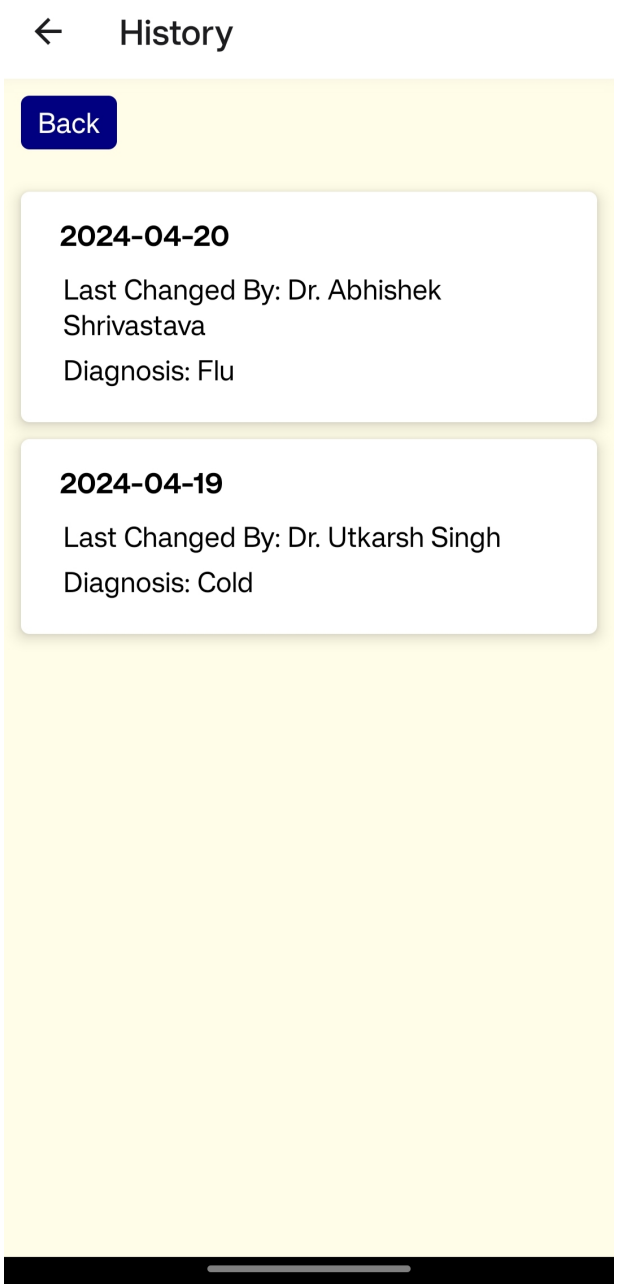
The patient has the ability to modify their personal details as illustrated in the figure. If any modifications are made, the updated details are logged in the ledger through the patient contract. When the **Save Changes** button is clicked, the user is redirected to the patient details page.

3.2.10 View Doctors



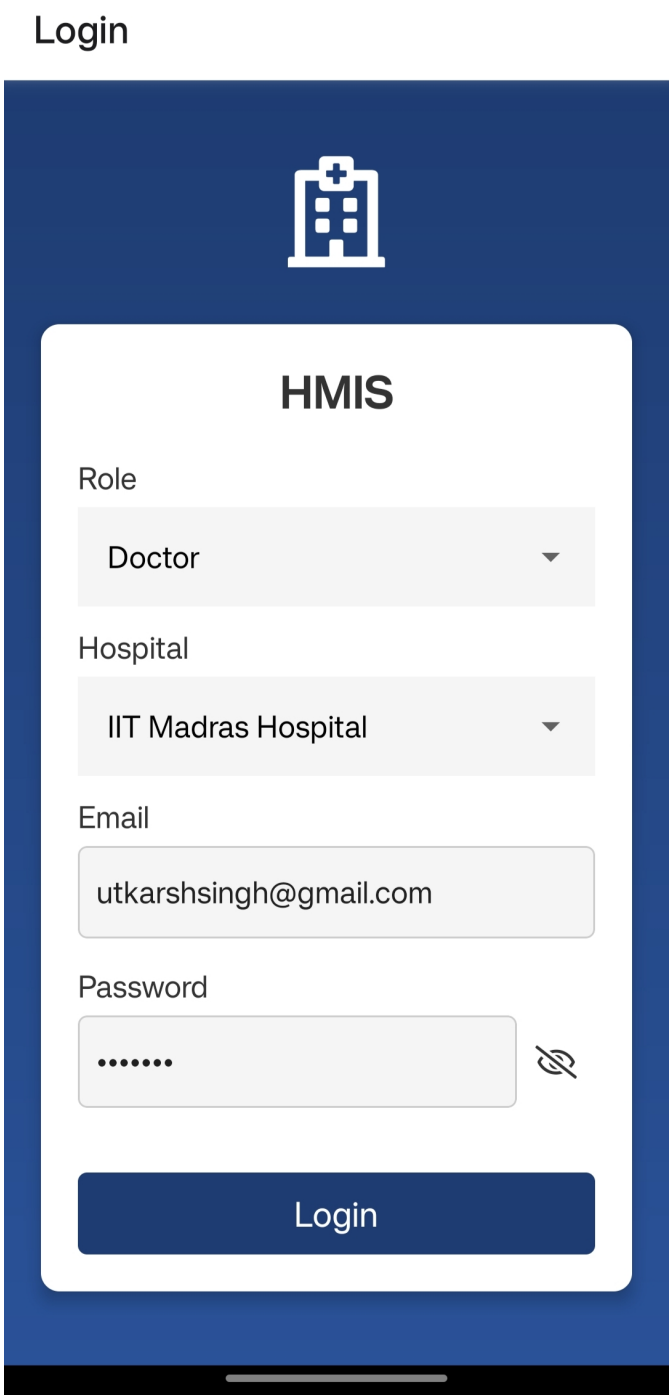
Selecting "View Doctors" presents a list of hospital doctors, as depicted in the figure. The patient can either **grant** or **revoke** a access to the doctor to patient's medical details. Access is required for a doctor to view the patient's medical details. If the patient revokes access, the doctor will no longer be able to view the patient's medical records.

3.2.11 View History



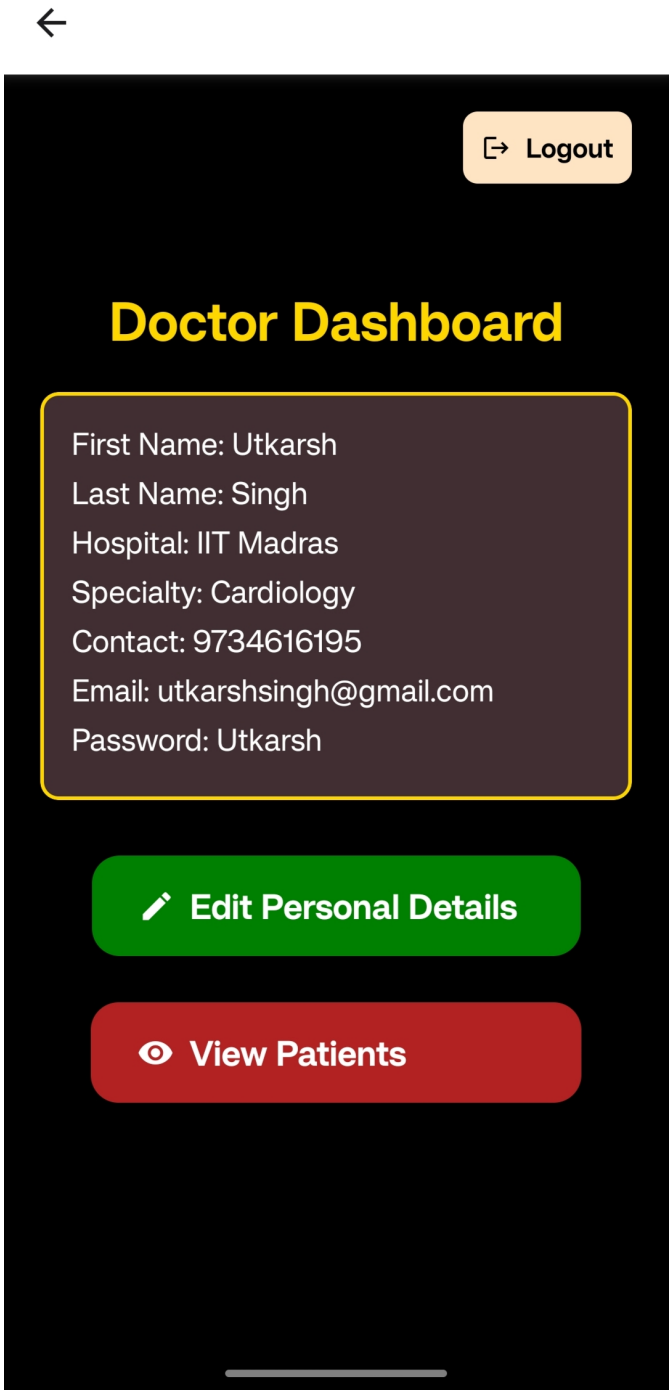
The patient can access a comprehensive view of all records, both personal and medical, from their first to their most recent visit with any doctor, as illustrated in the figure. This capability is enabled by the **getHistoryForKey** API in the Hyperledger Fabric, which can retrieve the patient's history. This feature ensures full transparency of the transactions for the patient.

3.2.12 Doctor Login



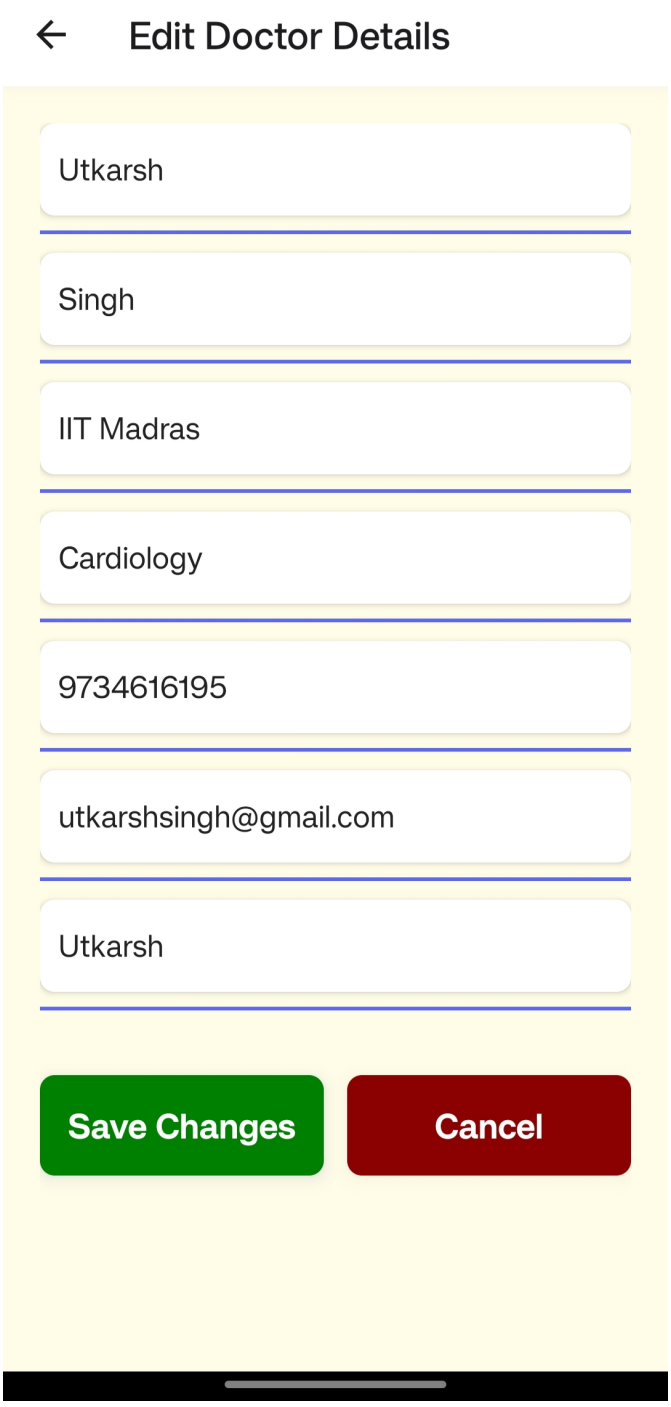
For log in as a doctor, the user must select **'Doctor'** as their role and specify the hospital to which they belong, along with the correct credentials on the login page, as shown in the figure.

3.2.13 Doctor Dashboard



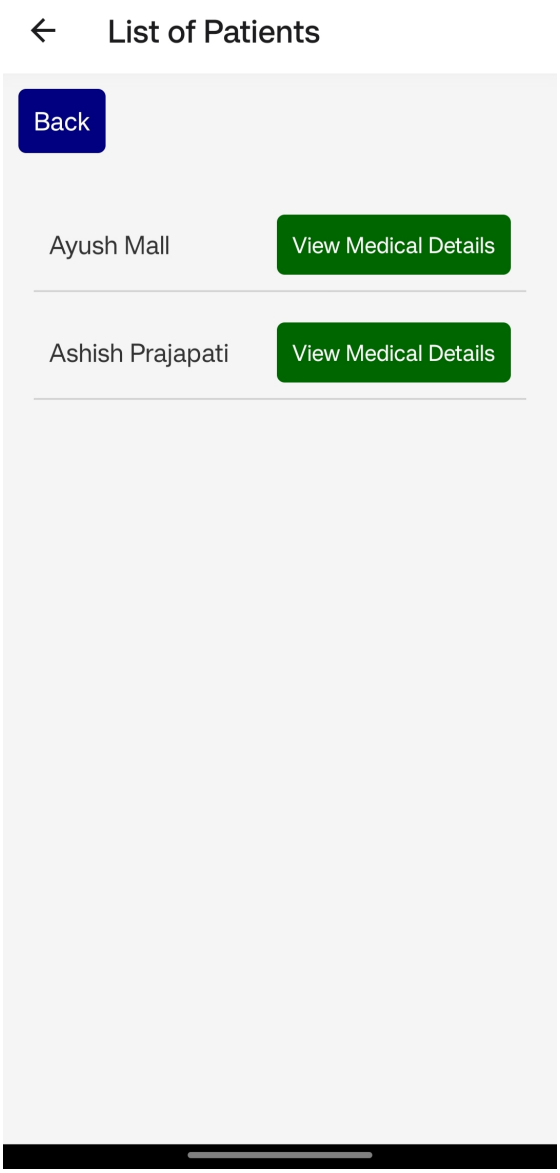
Once the patient successfully logs in, the **Doctor Dashboard** is displayed. This dashboard includes options to **Edit Personal Details** and **View Patients**, as depicted in the figure above. Clicking the Logout button allows the user to log out from this page.

3.2.14 Edit Doctor Details



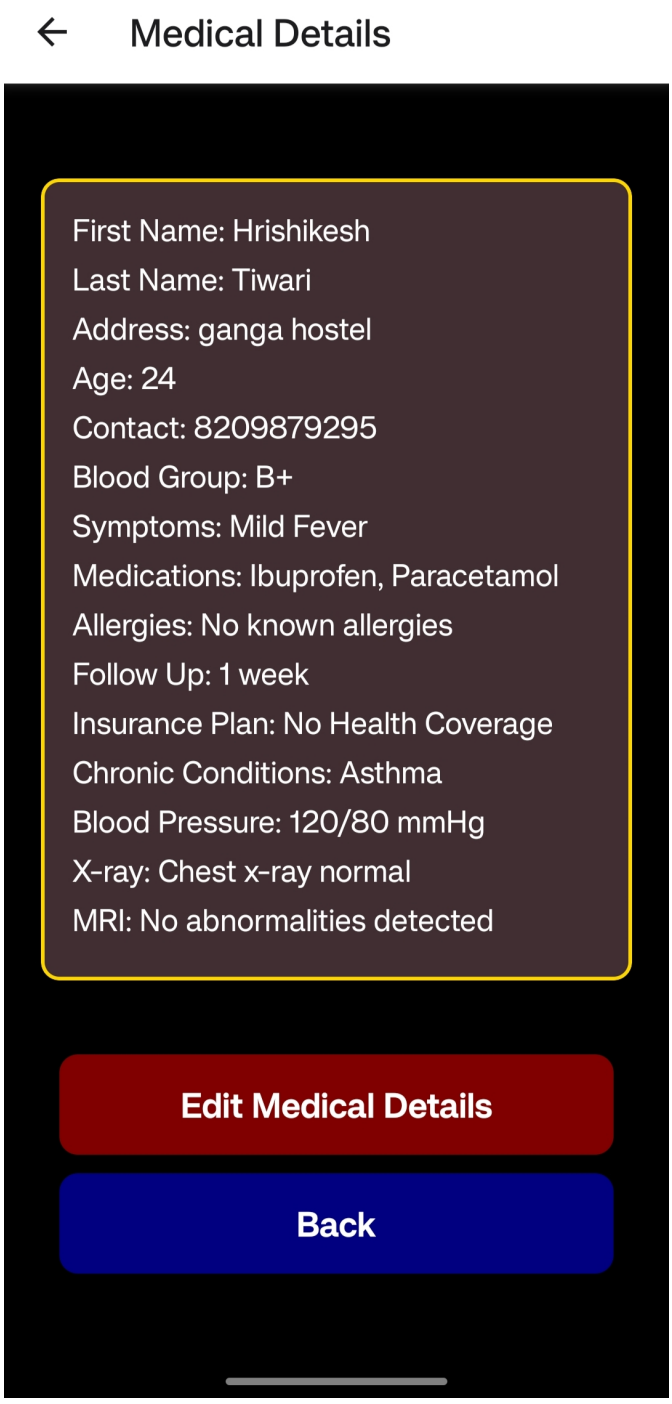
The doctor has the option to update their personal details as displayed in the figure. Any modifications made are recorded in the ledger through the doctor contract. After clicking the **Save Changes** button, the system redirects to the doctor details page.

3.2.15 List of Patients



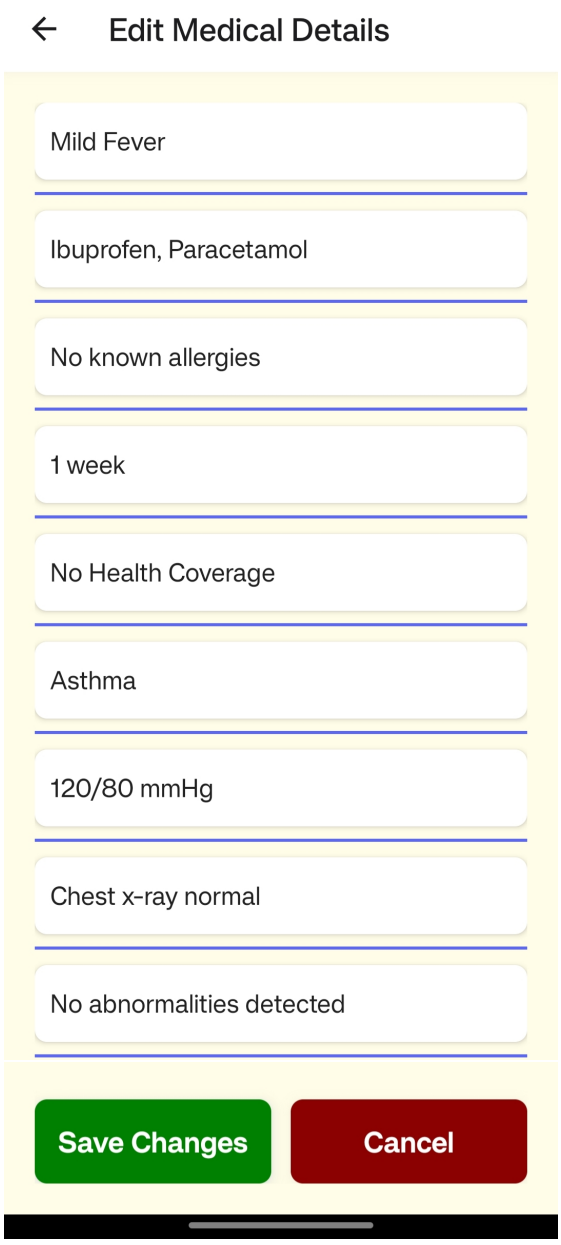
On clicking "view patients" doctors can view a list of patients who have granted them access; if no access is granted, the list will be empty. Each entry includes a "View More" button, as indicated in the figure, which directs the doctor to a page detailing the patient's medical information. However, only specific fields are visible to the doctor, as shown in the figure. This allows the doctor to review the patient's most current condition and information.

3.2.16 Medical Details



It contains the medical details of the patient which have given grant access to the doctor as shown in the figure above.

3.2.17 Edit Medical Details



Doctors can administer treatment by updating the patient's medical details. After clicking the **"Save Changes"** button, the page will redirect to the medical details page.

CHAPTER 4

FRONTEND

4.1 SYSTEM DESIGN

4.1.1 React Native

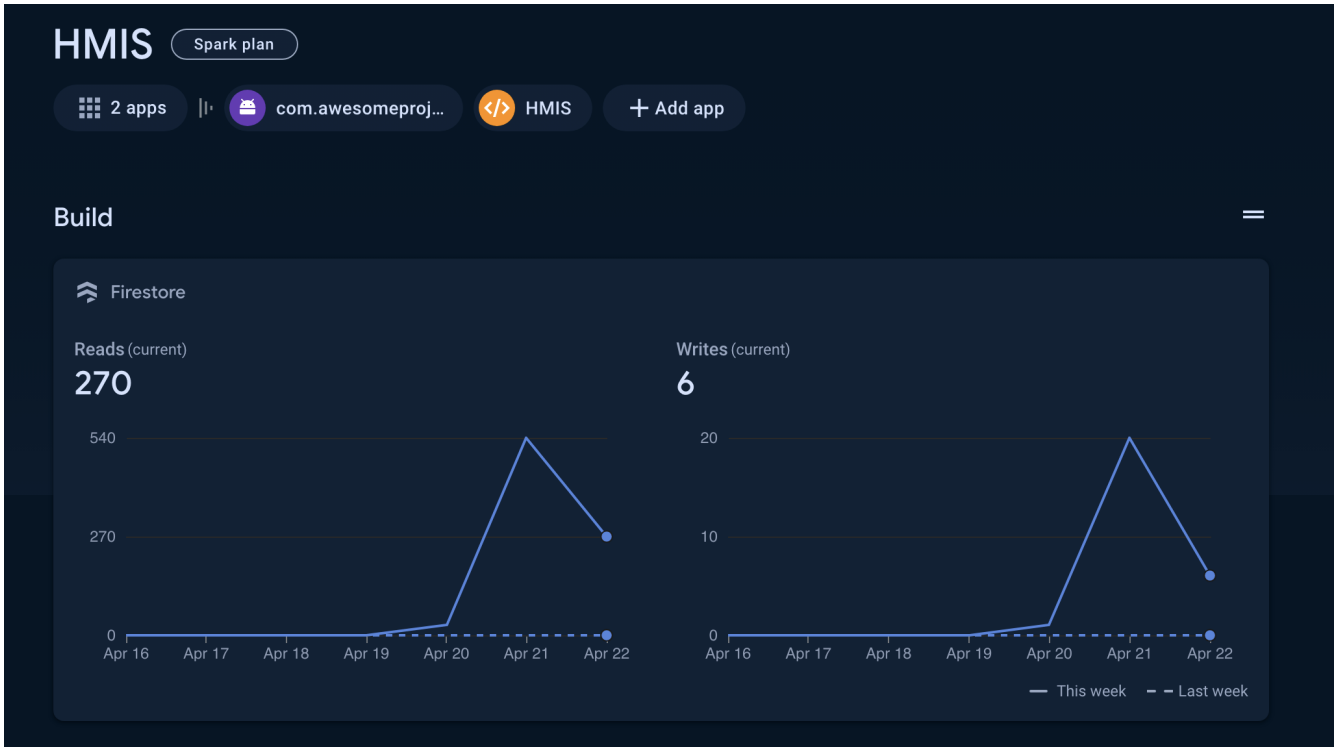
React Native is an open-source framework created by Facebook that allows for the development of native mobile apps using JavaScript and React. This framework enables developers to code once and deploy on both iOS and Android platforms, providing a native app experience on each. This framework facilitates the development of applications for multiple platforms using a single codebase.

4.1.2 Expo Go

Expo Go is a free app available on iOS and Android that allows developers to quickly view and test their React Native projects on real devices during development. It facilitates the viewing of projects without needing to compile the app binaries (APK or IPA files) every time a change is made. Developers can scan a QR code generated by the Expo CLI to open their project in Expo Go, streamlining the development and testing process.

4.1.3 Firebase

Firebase, developed by Google, is a platform designed for creating mobile and web applications. It provides a comprehensive suite of tools and services that aid developers in constructing high-quality apps, enhancing app performance, and expanding their user base. Firebase offers features such as analytics, databases, messaging, crash reporting, authentication, and file storage. Developers can synchronize data across devices in real-time using Firebase Realtime Database, authenticate users, and oversee app deployment and monitoring. Renowned for its user-friendly interface and quick configuration, Firebase facilitates swift application development.



(6)

```
export default function App() {
  const firebaseConfig = {
    apiKey: "AIzaSyCjZf6-sReT-3shhyVJrNMx48b0NeJT4io",
    authDomain: "hmis-3babd.firebaseapp.com",
    projectId: "hmis-3babd",
    storageBucket: "hmis-3babd.appspot.com",
    messagingSenderId: "436219627889",
    appId: "1:436219627889:web:7046a9fd214fce77fcced0",
    measurementId: "G-TPCD74N25H"
  };
}
```

4.1.4 initializeApp and getFirestore

1. **const app = initializeApp(firebaseConfig)**

    - This line initializes a Firebase application instance using the provided configuration object firebaseConfig. The configuration includes various keys and identifiers required for the Firebase services to work and to connect to the correct project in your Firebase account.

    - initializeApp is a function from the Firebase SDK that takes the configuration object and returns an initialized app instance.

2. **const db = getFirestore(app)**

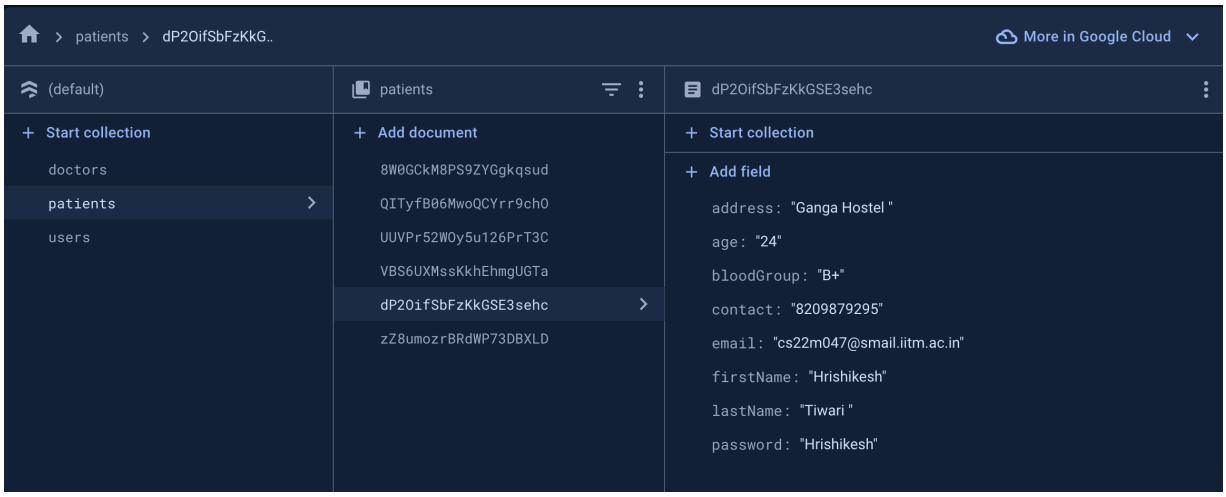
    - After the Firebase app has been initialized, this line gets an instance of Firestore, which is Firebase's NoSQL cloud database. getFirestore is a function from the Firestore SDK that takes the initialized Firebase app instance and returns a Firestore database instance.

    - The db constant is then used to interact with the Firestore database, allowing you to perform operations like reading, writing, updating, and listening to data in your Firestore collections.

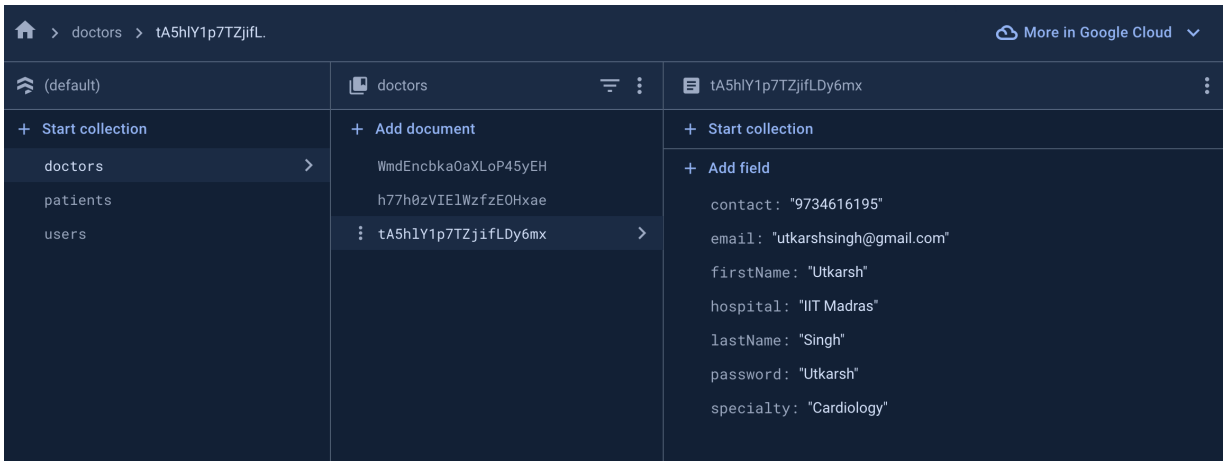
4.1.5 Firebase Database

The Firebase Realtime Database is a cloud-hosted NoSQL database that facilitates the storage and real-time synchronization of data among users. When data is updated, all connected clients receive these changes within milliseconds. Data is stored in JSON format and is continuously synced to every connected client. Access to the data is direct from client devices, including JavaScript for web apps, Android, and iOS clients. Firebase also offers a suite of declarative security rules that developers can employ to protect their data and manage access based on authentication and authorization levels. Additionally, the Realtime Database features offline capabilities, allowing client apps to write and read data even when offline. Any changes made are then synchronized once a network connection is re-established.

1. **Patient Firebase**

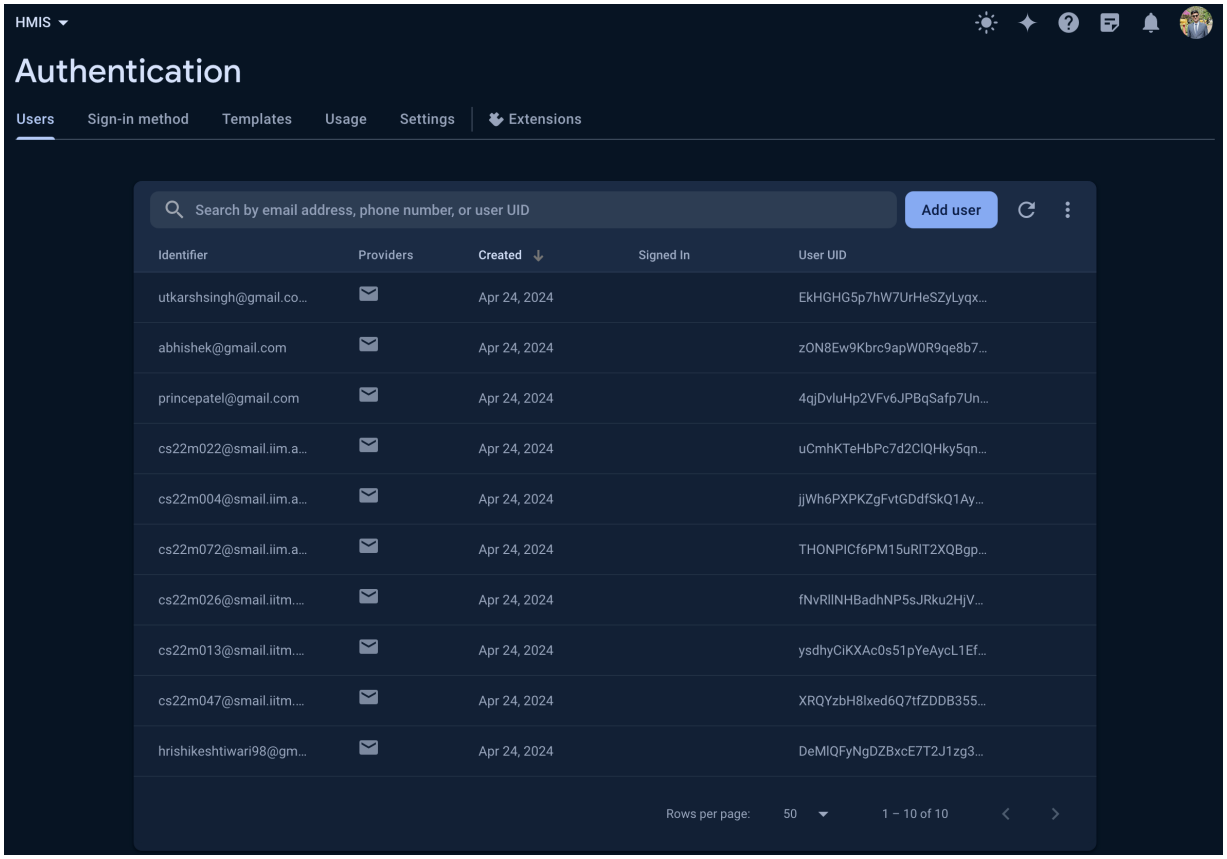


2. **Doctor Firebase**



4.1.6 Authentication

Firebase Authentication works by providing backend services and easy-to-use SDKs that allow developers to authenticate users with various methods including email/password, social media accounts, and phone numbers. It handles all aspects of user authentication, including security, token generation, and session management. Firebase Authentication integrates with other Firebase services and third-party providers, enabling developers to manage user authentication with minimal code.



(6)

```
function login(email, password) {
  signInWithEmailAndPassword(auth, email, password)
    .then((userCredential) => {
      // Signed in
      var user = userCredential.user;
      console.log('User logged in:', user);
    })
    .catch((error) => {
      var errorCode = error.code;
      var errorMessage = error.message;
      console.error('Login error:', errorCode, errorMessage);
    });
```

```
}
```

4.2 FLATLIST

In React Native, FlatList is a component used to efficiently display a scrolling list of data. It's highly optimized for lists of data that can change over time, rendering only those items visible on the screen. This approach enhances performance by not overloading the rendering process, especially for long lists.

```
const SomeComponent = () => {
  const data = [
    { id: '1', name: 'Hrishikesh Tiwari' },
  ];
  const renderItem = ({ item }) => (
    <View style={styles.item}>
      <Text style={styles.title}>{item.name}</Text>
    </View>
  );
  return (
    <FlatList
      data={data}
      renderItem={renderItem}
      keyExtractor={item => item.id}
    />
  );
};
```

4.3 ASYNC

In React, AsyncStorage is an asynchronous, unencrypted, persistent, key-value storage system that is globally available to all app components. It is commonly used for storing small amounts of data, such as user preferences or authentication tokens, locally on the user's device. AsyncStorage is typically used in React Native applications for storing data across app sessions.

4.3.1 Storing Data:

To store data using AsyncStorage, you can use the setItem method, which takes a key-value pair as arguments. The key is a string used to identify the data, and the value can be a string, number, boolean, or object that you want to store. AsyncStorage stores data as strings, so if you want to store complex data types like objects, you need to serialize them into JSON strings before storing and deserialize them when retrieving. AsyncStorage is used to store data such as user tokens and user IDs. For example, in the login process, after a successful login, the user's token and user ID are stored in AsyncStorage.

```
AsyncStorage.setItem('key', 'value')
  .then(() => {
    console.log('Data stored successfully');
  })
  .catch((error) => {
    console.error('Error storing data: ', error);
  });
```

4.3.2 Retrieving Data:

AsyncStorage is also used to retrieve stored data. For instance, in the index.tsx file, AsyncStorage is used to check if the user is already logged in by looking for the presence of a user ID.

```
AsyncStorage.getItem('key')
  .then((value) => {
    if (value !== null) {
      console.log('Retrieved value: ', value);
    } else {
      console.log('No data found');
    }
  })
  .catch((error) => {
    console.error('Error retrieving data: ', error);
  });
```

4.4 COMPONENTS AND SCREENS

This code renders a touchable opacity component (TouchableOpacity) that serves as a button. When pressed, it navigates to the screen with the name 'ListOfPatients'. The style prop applies two styles: styles.button and styles.listPatientsButton, which define the appearance of the button. Inside the component, there's an icon (Icon) displaying a list symbol, followed by text (Text) saying "Patients List". The navigation.navigate() function is called with the parameter 'ListOfPatients', indicating the screen to navigate to when the button is pressed.

```
<Animatable.View animation="fadeInUp" delay={700} style=
{styles.buttonsContainer}>
<TouchableOpacity style={[styles.button, styles.listPatientsButton]}
onPress={() => navigation.navigate('ListOfPatients')}>
    <Icon name="format-list-bulleted" size={24} color="#ffffff" />
    <Text style={styles.buttonText}>Patients List    </Text>
```

```
  </TouchableOpacity>
</Animatable.View>
```

The application is composed of various screens and components, each responsible for a specific part of the UI. For instance, the adminDashboard.js screen is responsible for admin dashboard, and actions like createPatient, patientsList, createDoctor and doctorsList. The patientDashboard.js component serves a similar purpose for patients, and The doctorDashboard.js component serves a similar purpose for doctors. These components are used within screens defined in the AppNavigation.js configuration.

4.5 HANDLE ACTIONS

handleSave, handleChange and handleCancel are likely functions used to handle actions such as saving changes, changing or canceling an operation in a React component.

1. **handleSave:** This function is typically called when a user confirms their changes and wants to save them. It might include logic to update data, make API calls, or trigger other necessary actions to save the changes.

```
const handleSave = async () => {
    if (Object.values(formData).some(value => value === '')) {
      Alert.alert('Error', 'All fields are required.');
      return;
    }

    try {
      const docRef = await addDoc(collection(db, "doctors"), formData);
      console.log("Document written with ID: ", docRef.id);
      Alert.alert('Success', 'Doctor data saved successfully!');
      navigation.goBack();
    } catch (e) {
      console.error("Error adding document: ", e);
      Alert.alert('Error', 'Failed to save the data.');
    }
  };
```

2. **handleChange:** handleChange is a common naming convention for a function used to handle changes in input fields or form elements in React.

```
const handleChange = (name, value) => {
    setFormData(prevState => ({ ...prevState, [name]: value }));
  };
```

3. **handleCancel:** This function is usually invoked when a user decides to discard their changes and revert to the previous state. It may involve resetting form fields, navigating away from the current page, or any other cleanup actions needed to cancel the operation.

```
const handleCancel = () => {
    navigation.goBack();
  };
```

(14)

4.6 NAVIGATION

In React Navigation, navigation refers to the object provided by the navigation container that allows you to navigate between screens in your app. It typically includes methods like navigate, goBack, push, pop, and others, depending on the navigation library being used.

```
const AppNavigator = () => {
  return (
    <NavigationContainer>
    <PatientProvider>
    <PatientDataProvider>
    <DoctorDataProvider>
    <DoctorProvider>
      <Stack.Navigator initialRouteName="LoginScreen">
        <Stack.Screen name="LoginScreen" component={LoginScreen} options=
        {{ title: 'Login' }} />
        <Stack.Screen name="AdminDashboard" component={AdminDashboard}
        options={{ title: '' }} />
```

```
    <Stack.Screen name="CreatePatientForm" component=
    {CreatePatientForm} options={{ title: 'Create Patient' }} />
    <Stack.Screen name="CreateDoctorForm" component={CreateDoctorForm}
    options={{ title: 'Create Doctor' }} />
    <Stack.Screen name="PatientDetails" component={PatientDetails}
    options={{ title: '' }} />
    <Stack.Screen name="EditPersonalDetails" component=
    {EditPersonalDetails} options={{ title: 'Edit Personal Details' }} />
    <Stack.Screen name="ViewDoctors" component={ViewDoctors} options={{
    title: 'Doctors' }} />
    <Stack.Screen name="ViewHistory" component={ViewHistory} options={{
    title: 'History' }} />
    <Stack.Screen name="DoctorDetails" component={DoctorDetails} options=
    {{ title: '' }} />
    <Stack.Screen name="PatientList" component={PatientList} options={{
    title: 'List of Patients' }} />
    <Stack.Screen name="EditMedicalDetails" component=
    {EditMedicalDetails} options={{ title: 'Edit Medical Details' }} />
    <Stack.Screen name="ViewMedicalDetails" component=
    {ViewMedicalDetails} options={{ title: 'Medical Details' }} />
    <Stack.Screen name="EditDoctorDetails" component={EditDoctorDetails}
    options={{ title: 'Edit Doctor Details' }} />
    <Stack.Screen name="ListOfPatients" component={ListOfPatients}
    options={{ title: 'List of Patients' }} />
    <Stack.Screen name="ListOfDoctors" component={ListOfDoctors} options=
    {{ title: 'List of Doctors' }} />
  </Stack.Navigator>
</DoctorProvider>
```

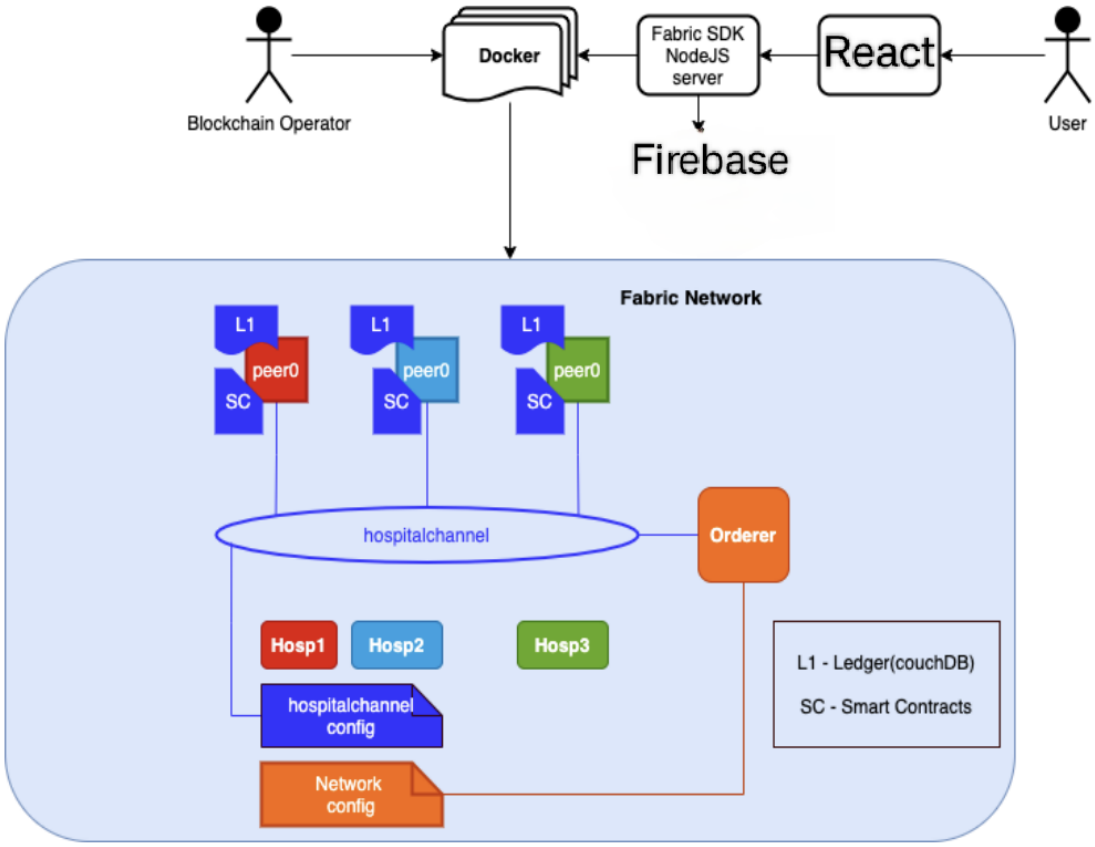
```
    </DoctorDataProvider>
    </PatientDataProvider>
    </PatientProvider>
    </NavigationContainer>
  );
```

In React Navigation, we might have a StackNavigator where each screen receives a navigation prop containing these methods. You can use these methods to programmatically navigate to different screens, go back to the previous screen, or perform other navigation actions.

CHAPTER 5

BACKEND

5.1 ARCHITECTURE FLOW



5.2 JSON WEB TOKEN

A JSON Web Token (JWT) is a compact and URL-safe method for representing claims shared between two parties. It comprises three components: a header, a payload, and a signature. The header usually includes the type of token and the hash algorithm in use.

The payload carries the claims, which are statements about an entity (usually the user) along with other data. The signature verifies that the token has not been modified. JWTs are commonly used in authentication and data exchange processes, their compact format making them ideal for inclusion in HTTP headers or URL query parameters.

5.2.1 Authorisation with JWT

Authorization with JSON Web Tokens (JWT) typically follows this process:

1. **User Authentication:** The user logs in with their credentials (like a username and password).

2. **Generate JWT:** Upon successful authentication, the server generates a JWT. This token includes claims about the user and additional data needed for authorization. The server signs the JWT with a secret key or a public/private key pair.

3. **Send JWT to Client:** The server sends the JWT back to the client, usually in an HTTP response.

4. **Client Stores JWT:** The client stores the JWT, often in local storage, session storage, or an HTTP-only cookie.

5. **Client Sends JWT in Subsequent Requests:** When the client makes subsequent requests to the server, it includes the JWT, typically in the Authorization header using the Bearer schema.

6. **Server Verifies JWT:** The server, upon receiving a request with a JWT, first verifies the token's signature to ensure its validity and integrity. This verification process checks that the token was indeed created by the server and has not been tampered with.

7. **Server Processes Claims:** If the token is valid, the server reads the token's claims to determine if the user is allowed to perform the requested action (this is the authorization part). For example, the claims might say that this user has admin privileges.

8. **Grant or Deny Access:** Based on the claims and possibly other information (like permissions stored in a database), the server decides to allow or deny the requested action.

9. **Respond to Client:** The server sends a response back to the client, which could be the requested resource for valid requests or an error message if the user is not authorized or if the token is invalid or expired.

```
const authenticateJWT = (req, res, next) => {
  const authHeader = req.headers.authorization;

  if (authHeader) {
    const token = authHeader.split(' ')[1];

    if (token === '' || token === 'null') {
      return res.status(401).send('Unauthorized request: Token is missing');
    }
    jwt.verify(token, jwtSecretToken, (err, user) => {
      if (err) {
        return res.status(403).send('Unauthorized request: Wrong or expired
        token found');
      }
      req.user = user;
      next();
    });
  } else {
    return res.status(401).send('Unauthorized request: Token is missing');
  }
};
```

5.2.2 Generate Access Token

The generateAccessToken function creates a JSON Web Token (JWT) that serves as an access token for authentication. It signs a payload containing user information and an expiration period with a secret key, and returns the signed token. This token is then sent to the client, which uses it to make authenticated requests to the server. The server verifies the token to grant access to protected resources.

```
function generateAccessToken(username, role) {
  return jwt.sign({username: username, role: role}, jwtSecretToken, {expiresIn: '
}
```

5.3 API'S

APIs (Application Programming Interfaces) are collections of rules and tools that enable communication between various software applications. They establish the methods and data formats that applications can utilize to request and exchange information. API methods are specific actions that can be performed, including GET (to retrieve data), POST (to create data), PUT (to update data), and DELETE (to remove data). These methods enable developers to send requests and receive responses through the API, usually over the internet.

1. **POST /createPatient:** This function is called when there is an HTTP request to create a new patient. It performs several actions:
   - Validates the user's role.
   - Connects to the Fabric network.
   - Optionally generates a new patient ID and a default password if not provided.
   - Calls the createPatient smart contract function.
   - Registers the user with the Certificate Authority (CA) and adds them to the wallet.
   - Sends back a response with the status code 201 (Created) or 400 (Bad Request) depending on the result.

```
exports.createPatient = async (req, res) => {
  await validateRole([ROLE_ADMIN], userRole, res);
  const networkObj = await network.connectToNetwork(req.headers.username);

  if (!('patientId' in req.body) || req.body.patientId === null ||
  req.body.patientId === '') {
    const lastId = await network.invoke(networkObj, true,
    capitalize(userRole) + 'Contract:getLatestPatientId');
```

```
    req.body.patientId = 'PID' + (parseInt(lastId.slice(3)) + 1);
  }

  if (!('password' in req.body) || req.body.password === null ||
  req.body.password === '') {
    req.body.password = Math.random().toString(36).slice(-8);
  }

  req.body.changedBy = req.headers.username;

  const data = JSON.stringify(req.body);
  const args = [data];
  const createPatientRes = await network.invoke(networkObj, false,
  capitalize(userRole) + 'Contract:createPatient', args);
  if (createPatientRes.error) {
    res.status(400).send(response.error);
  }

  const userData = JSON.stringify({hospitalId:
  (req.headers.username).slice(4, 5), userId: req.body.patientId});
  const registerUserRes = await network.registerUser(userData);

  if (registerUserRes.error) {
    await network.invoke(networkObj, false, capitalize(userRole) +
    'Contract:deletePatient', req.body.patientId);
    res.send(registerUserRes.error);
  }

  res.status(201).send(getMessage(false, 'Successfully registered Patient.',
  req.body.patientId, req.body.password));
};
```

2. **POST /createDoctor:** This function is called when there is an HTTP request to create a new doctor. It:

   - Validates the user's role.
   - Creates a Firebase client and stores the doctor's credentials.
   - Registers the doctor with the CA and adds them to the wallet.
   - Sends back a response with the status code 201 (Created) or 400 (Bad Request) depending on the result.

```
exports.createDoctor = async (req, res) => {
```

```
  const userRole = req.headers.role;
  let {hospitalId, username, password} = req.body;
  hospitalId = parseInt(hospitalId);

  await validateRole([ROLE_ADMIN], userRole, res);

  req.body.userId = username;
  req.body.role = ROLE_DOCTOR;
  req.body = JSON.stringify(req.body);
  const args = [req.body];
  const firebaseClient = createFirebaseClient(hospitalId);
  (await firebaseClient).SET(username, password);
  const response = await network.registerUser(args);
  if (response.error) {
    (await firebaseClient).DEL(username);
    res.status(400).send(response.error);
  }
  res.status(201).send(getMessage(false, response, username, password));
};
```

3. **GET /getAllPatients:** This function retrieves all patient assets from the ledger. It:
   - Validates the user's role.
   - Connects to the Fabric network.
   - Calls the queryAllPatients smart contract function.
   - Parses and sends the list of all patients in a response with the status code 200 (OK).

```
exports.getAllPatients = async (req, res) => {
  const userRole = req.headers.role;
  await validateRole([ROLE_ADMIN, ROLE_DOCTOR], userRole, res);
  const networkObj = await network.connectToNetwork(req.headers.username);
  const response = await network.invoke(networkObj, true,
  capitalize(userRole) + 'Contract:queryAllPatients',
    userRole === ROLE_DOCTOR ? req.headers.username : '');
  const parsedResponse = await JSON.parse(response);
  res.status(200).send(parsedResponse);
};
```

4. **GET /getAllDoctors:** This function retrieves all doctor assets from the ledger. It:
   - Validates the user's role.

- Connects to the Fabric network.
- Calls the queryAllDoctors smart contract function.
- Parses and sends the list of all doctors in a response with the status code 200 (OK).

5. **PUT /updatePatientMedicalDetails:**
   - Validates the user role to ensure only authorized roles (doctors) can execute the action.
   - Retrieves the patient's ID from the URL parameters and other details from the request body.
   - Connects to a blockchain network.
   - Invokes a smart contract function to update patient medical details on the blockchain.
   - Returns a response with status 200 (OK) if successful, or 500 (Server Error) if there's an error.

```
exports.updatePatientMedicalDetails = async (req, res) => {
  const userRole = req.headers.role;
  await validateRole([ROLE_DOCTOR], userRole, res);
  let args = req.body;
  args.patientId = req.params.patientId;
  args.changedBy = req.headers.username;
  args= [JSON.stringify(args)];
  const networkObj = await network.connectToNetwork(req.headers.username);
  const response = await network.invoke(networkObj, false,
  capitalize(userRole) + 'Contract:updatePatientMedicalDetails', args);
  (response.error) ? res.status(500).send(response.error) :
  res.status(200).send(getMessage(false, 'Successfully Updated Patient.'));
};
```

6. **GET /getDoctorById:**
   - Validates the user role.
   - Depending on the hospitalId from the URL, it assigns a specific admin user ID for network connection.
   - Connects to a blockchain network.

- Retrieves and filters the list of doctors enrolled by the Certificate Authority to find a specific doctor by ID.

- Returns the doctor's details in the response with a 200 status if found, or 500 if there's an error or the doctor is not found.

```
exports.getDoctorById = async (req, res) => {
  const userRole = req.headers.role;
  await validateRole([ROLE_DOCTOR], userRole, res);
  const hospitalId = parseInt(req.params.hospitalId);
  const userId = hospitalId === 1 ? 'hosp1admin' : hospitalId === 2 ?
  'hosp2admin' : 'hosp3admin';
  const doctorId = req.params.doctorId;
  const networkObj = await network.connectToNetwork(userId);
  const response = await network.getAllDoctorsByHospitalId(networkObj,
  hospitalId);
  (response.error) ? res.status(500).send(response.error) :
  res.status(200).send(response.filter(
    function(response) {
      return response.id === doctorId;
    },
  )[0]);
};
```

7. **GET /getPatientById:** Retrieves a patient's details from the ledger using their ID.

```
exports.getPatientById = async (req, res) => {
  const userRole = req.headers.role;
  await validateRole([ROLE_DOCTOR, ROLE_PATIENT], userRole, res);
  const patientId = req.params.patientId;
  const networkObj = await network.connectToNetwork(req.headers.username);
  const response = await network.invoke(networkObj, true,
  capitalize(userRole) + 'Contract:readPatient', patientId);
  (response.error) ? res.status(400).send(response.error) :
  res.status(200).send(JSON.parse(response));
};
```

8. **PUT /updatePatientPersonalDetails:** Updates personal details of a patient. This method is accessible only by the patient.

```
exports.updatePatientPersonalDetails = async (req, res) => {
  const userRole = req.headers.role;
  await validateRole([ROLE_PATIENT], userRole, res);
  let args = req.body;
  args.patientId = req.params.patientId;
```

```
  args.changedBy = req.params.patientId;
  args= [JSON.stringify(args)];
  const networkObj = await network.connectToNetwork(req.headers.username);
  const response = await network.invoke(networkObj, false,
  capitalize(userRole) + 'Contract:updatePatientPersonalDetails', args);
  (response.error) ? res.status(500).send(response.error) : res.status(200).
};
```

9. **GET /getPatientHistoryById:** Retrieves the transaction history of a patient's record from the ledger.

```
exports.getPatientHistoryById = async (req, res) => {
  const userRole = req.headers.role;
  await validateRole([ROLE_DOCTOR, ROLE_PATIENT], userRole, res);
  const patientId = req.params.patientId;
  const networkObj = await network.connectToNetwork(req.headers.username);
  const response = await network.invoke(networkObj, true,
  capitalize(userRole) + 'Contract:getPatientHistory', patientId);
  const parsedResponse = await JSON.parse(response);
  (response.error) ? res.status(400).send(response.error) : res.status(200).
};
```

10. **GET /getDoctorsByHospitalId:** Fetches all doctors associated with a specific hospital ID.

```
exports.getDoctorsByHospitalId = async (req, res) => {
  const userRole = req.headers.role;
  await validateRole([ROLE_PATIENT, ROLE_ADMIN], userRole, res);
  const hospitalId = parseInt(req.params.hospitalId);
  userId = hospitalId === 1 ? 'hosp1admin' : hospitalId === 2 ? 'hosp2admin'
  : 'hosp3admin';
  const networkObj = await network.connectToNetwork(userId);
  const response = await network.getAllDoctorsByHospitalId(networkObj,
  hospitalId);
  (response.error) ? res.status(500).send(response.error) :
  res.status(200).send(response);
};
```

11. **POST /grantAccessToDoctor:** Allows a patient to grant access to their records to a specific doctor.

```
exports.grantAccessToDoctor = async (req, res) => {
  const userRole = req.headers.role;
  await validateRole([ROLE_PATIENT], userRole, res);
  const patientId = req.params.patientId;
```

```
  const doctorId = req.params.doctorId;
  let args = {patientId: patientId, doctorId: doctorId};
  args= [JSON.stringify(args)];
  const networkObj = await network.connectToNetwork(req.headers.username);
  const response = await network.invoke(networkObj, false,
  capitalize(userRole) + 'Contract:grantAccessToDoctor', args);
  (response.error) ? res.status(500).send(response.error) :
  res.status(200).send(getMessage(false, `Access granted to ${doctorId}`));
};
```

12. **POST /revokeAccessToDoctor:** Allows a patient to revoke access from a doctor previously granted access to view their records.

```
exports.revokeAccessFromDoctor = async (req, res) => {
  const userRole = req.headers.role;
  await validateRole([ROLE_PATIENT], userRole, res);
  const patientId = req.params.patientId;
  const doctorId = req.params.doctorId;
  let args = {patientId: patientId, doctorId: doctorId};
  args= [JSON.stringify(args)];
  const networkObj = await network.connectToNetwork(req.headers.username);
  const response = await network.invoke(networkObj, false,
  capitalize(userRole) + 'Contract:revokeAccessFromDoctor', args);
  (response.error) ? res.status(500).send(response.error) :
  res.status(200).send(getMessage(false, `Access revoked from ${doctorId}`))
};
```

13. **POST /login:** Authenticates users (doctors, admins, or patients) and returns access and refresh tokens.

14. **POST /token:** Generates a new access token using a valid refresh token.

```
app.post('/token', (req, res) => {
  const {token} = req.body;

  if (!token) {
    return res.sendStatus(401);
  }
  if (!refreshTokens.includes(token)) {
    return res.sendStatus(403);
  }
  jwt.verify(token, refreshSecretToken, (err, username) => {
    if (err) {
      return res.sendStatus(403);
    }
    const accessToken = generateAccessToken({username: username, role:
```

```
    req.headers.role});
    res.json({
      accessToken,
    });
  });
});
```

15. **DELETE /logout:** Invalidates the refresh token to end the session.

```
app.delete('/logout', (req, res) => {
  refreshTokens = refreshTokens.filter((token) => token !==
  req.headers.token);
  res.sendStatus(204);
});
```

16. **POST /patients/register:** Registers a new patient, accessible only by authenticated admins.

```
app.post('/patients/register', authenticateJWT, adminRoutes.createPatient);
```

17. **POST /doctors/register:** Registers a new doctor, accessible only by authenticated users with proper roles.

```
app.post('/doctors/register', authenticateJWT, adminRoutes.createDoctor);
```

18. **GET /patients/_all:** Retrieves all registered patients, accessible only by authenticated admins.

```
app.get('/patients/_all', authenticateJWT, adminRoutes.getAllPatients);
```

19. **GET /doctors/_all:** Retrieves all registered doctors, accessible only by authenticated admins.

```
app.get('/doctors/_all', authenticateJWT, adminRoutes.getAllDoctors);
```

20. **PATCH /patients/:patientId/details/medical:** Updates medical details of a specific patient, accessible only by authenticated doctors.

```
app.patch('/patients/:patientId/details/medical', authenticateJWT,
doctorRoutes.updatePatientMedicalDetails);
```

21. **GET /doctors/:hospitalId/:doctorId:** Retrieves a specific doctor's details by ID, accessible only by authenticated doctors.

```
app.get('/doctors/:hospitalId([0-9]+)/:doctorId(HOSP[0-9]+\-DOC[0-9]+)',
authenticateJWT, doctorRoutes.getDoctorById);
```

22. **GET /patients/:patientId** Retrieves details of a specific patient, accessible only by authenticated users.

```
app.get('/patients/:patientId', authenticateJWT,
patientRoutes.getPatientById);
```

23. **PATCH /patients/:patientId/details/personal:** Updates personal details of a specific patient, accessible only by the patient themselves.

```
app.patch('/patients/:patientId/details/personal', authenticateJWT,
patientRoutes.updatePatientPersonalDetails);
```

24. **GET /patients/:patientId/history:** Retrieves the transaction history of a specific patient's record from the ledger.

```
app.get('/patients/:patientId/history', authenticateJWT,
patientRoutes.getPatientHistoryById);
```

25. **GET /doctors/:hospitalId/_all:** Retrieves all doctors within a specific hospital, accessible by authenticated patients.

```
app.get('/doctors/:hospitalId([0-9]+)/_all', authenticateJWT,
patientRoutes.getDoctorsByHospitalId);
```

26. **PATCH /patients/:patientId/grant/:doctorId:** Grants a specific doctor access to a patient's records, accessible only by the patient.

```
app.patch('/patients/:patientId/grant/:doctorId', authenticateJWT,
patientRoutes.grantAccessToDoctor);
```

27. **PATCH /patients/:patientId/revoke/:doctorId:** Revokes a specific doctor's access to a patient's records.

```
app.patch('/patients/:patientId/revoke/:doctorId', authenticateJWT,
patientRoutes.revokeAccessFromDoctor);
```

5.4 PROCESS OF MAKING API'S PUBLIC

Ngrok is a powerful tool that creates secure tunnels to localhost, allowing you to expose your local server to the internet. It provides a unique URL that can be used to access your local server from anywhere, making it ideal for testing webhooks, APIs, and other server applications during development.

1. **Install Ngrok:** Download and install Ngrok.

2. **Run Ngrok:** Start Ngrok by running the command ngrok http <8081> in your terminal, where <port> is the port number on which your local server is running.

3. **Access Ngrok URL:** Ngrok will generate a unique URL that tunnels traffic to our local server. Using this URL we cant access our API from external clients.

4. **Testing and Development:** Share the Ngrok URL with our clients to test the API's in a real-world environment. Any requests made to the Ngrok URL will be forwarded to our local server, allowing us to debug and develop our API with ease.

5. **Secure Communication:** Ngrok provides HTTPS support, ensuring that traffic between clients and our local server is encrypted. This helps to maintain security while testing and developing our API.

6. **Monitoring and Management:** Ngrok offers a web-based dashboard where we can monitor traffic, inspect requests, and manage tunnels. Using this dashboard, we can track usage and debug any issues with our API.

5.5 PROCESS OF CONNECTION BETWEEN WEB SERVER AND PEER

1. Begin by loading the TLS certificate, which is essential for establishing secure communication.

2. Create a certificate pool, referred to as certPool, to store trusted certificates for verification purposes.

3. Incorporate the TLS certificate, usually issued by a Certificate Authority (CA), into the certificate pool to enable authentication.

4. Configure transport security for the gRPC connection, specifying the use of TLS for secure transmission.

5. Establish a gRPC connection to the peer's endpoint, utilizing the configured transport credentials.

6. Construct a client identity for the Gateway connection, utilizing an X.509 certificate as a key component.

7. Develop a digital signature function by extracting the private key from a designated directory.

8. Complete the connection setup by ensuring that the gRPC connection, client

identity, and digital signature function are properly configured. This setup enables the API server to communicate securely with the Hyperledger Fabric network, facilitating transaction proposals and ledger queries.

```
const { Gateway, client } = require('hyperledger-fabric-client');
async function initialize(setup) {
    console.log(`Initializing connection for ${setup.orgName}...`);
    const clientConnection = setup.newGrpcConnection();
    const id = setup.newIdentity();
    const sign = setup.newSign();
    try {
        const gateway = new Gateway();
        await gateway.connect({
            identity: id,
            signer: sign,
            clientConnection: clientConnection,
            evaluateTimeout: 5000,
            endorseTimeout: 15000,
            submitTimeout: 5000,
            commitStatusTimeout: 60000
        });
        setup.gateway = gateway;
        console.log("Initialization complete");
        return setup;
    } catch (error) {
        console.error(error);
        throw error;
    }
}
```

The above code initializes a connection for an organization in the Hyperledger Fabric network. It's using the hyperledger-fabric-client library to interact with the Fabric network. The initialize function is an asynchronous function that performs the connection setup.

CHAPTER 6

DEPLOYMENT OF CHAINCODE ON 4 VM AND API SERVER

6.1 TOOLS

Tools and technologies used for deployment of Hyperledger Fabric:

- **Docker:** Docker is a widely used open-source platform for developing, deploying, and managing applications. It allows you to decouple your applications from your infrastructure, facilitating rapid software delivery. Docker utilizes containers to ensure that an application operates consistently across any environment. These containers encapsulate software and all its dependencies into a standardized unit for software development, which includes the code, runtime, system tools, and libraries. This encapsulation ensures that the application runs identically in any environment.

- **Docker Compose:** Docker Compose is a tool designed to define and manage multi-container Docker applications. It uses a YAML file to set up your application's services, allowing you to launch and run all the services from your configuration with just one command. This tool streamlines the configuration and maintenance of intricate Docker environments, making it especially beneficial in development and testing settings.

- **Docker Swarm:** Docker Swarm is a native clustering and orchestration tool for Docker, designed to manage a cluster of Docker hosts and deploy applications throughout the cluster. With Docker Swarm, you can establish a "swarm" of Docker hosts and utilize the Docker CLI to deploy services to this swarm. It offers features such as load balancing, service discovery, and auto-scaling, simplifying the management and scaling of Docker applications in a production setting.

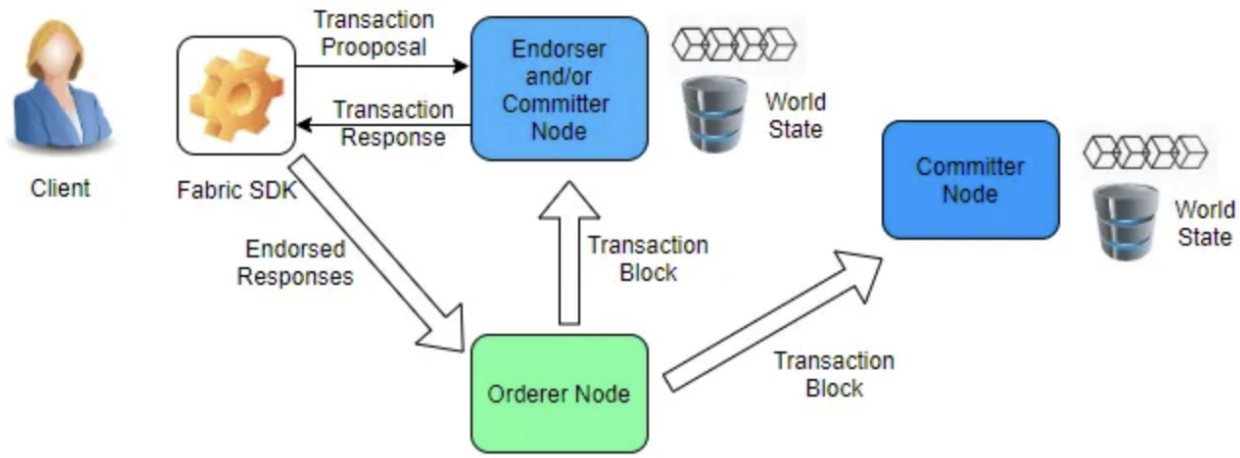
6.2 VM'S USED FOR DEPLOYMENT

1. VM1 ccd079@192.168.25.102

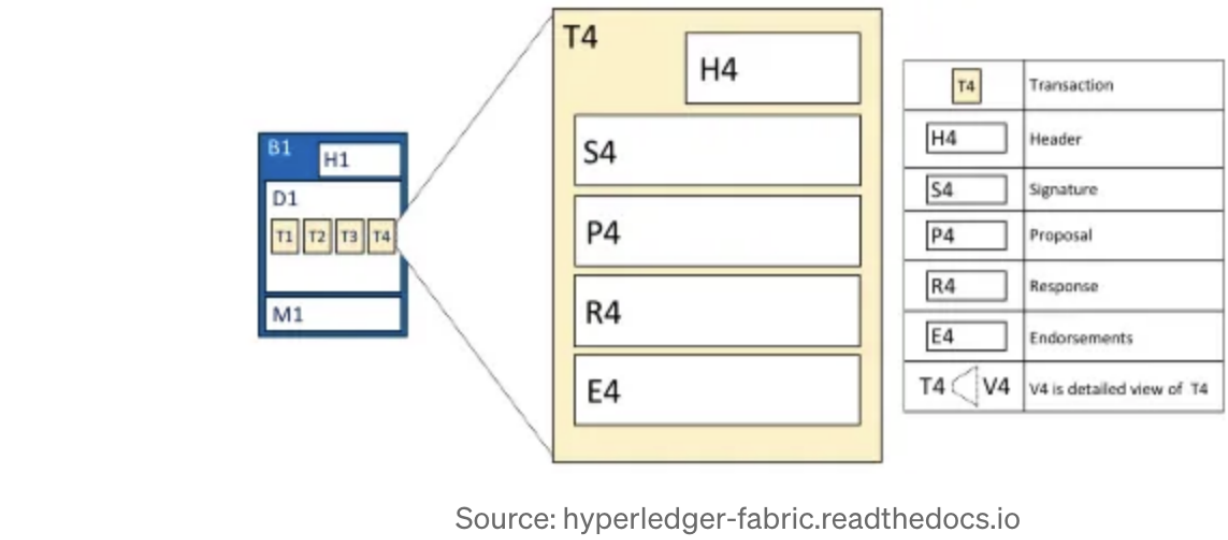
2. VM2 ccd080@192.168.25.103

3. VM3 ccd081@192.168.25.104

4. VM4 ccd082@192.168.25.105



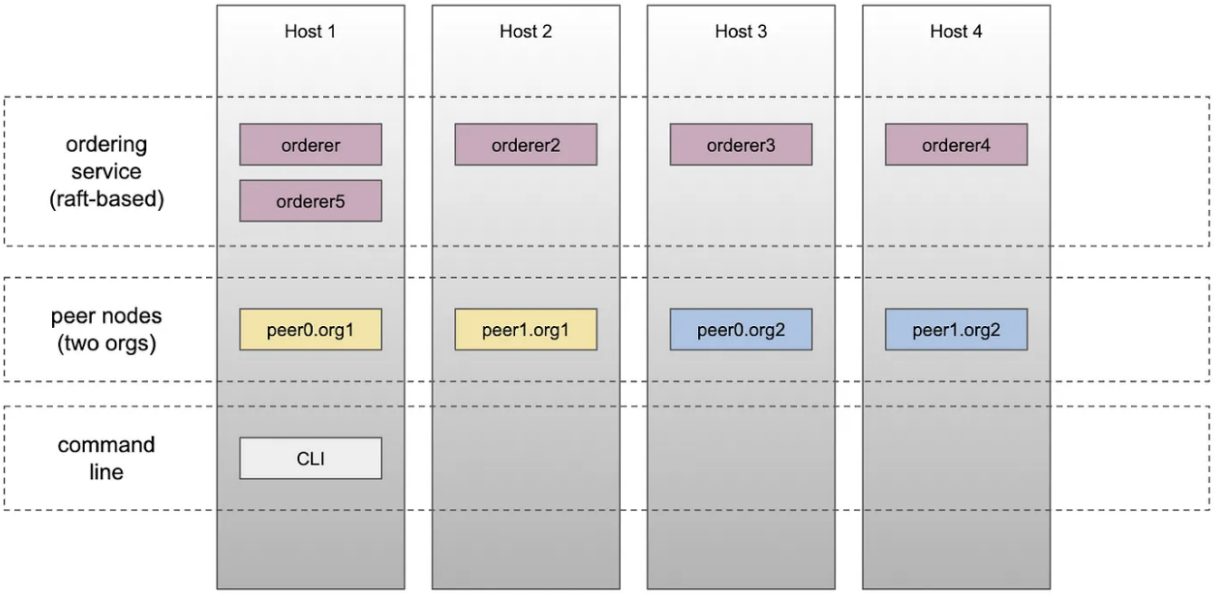
Hyperledger Fabric Basic Transaction Flow



6.3 ARCHITECTURE SETUP

(12)

First Network is made up of one orderer organization and two peer organizations. The orderer organization utilizes a raft-based ordering service cluster comprising five ordering service nodes (orderers). Within each peer organization, Org1 and Org2, there are two peers, specifically named peer0 and peer1. They all join a channel known as "mychannel," which is created to facilitate communication and transactions between the peers of both organizations.



1. **Ordering Service (Raft-based):**

   - The ordering service manages the sequencing of transactions and disseminates blocks filled with these ordered transactions to the peer nodes.
   - It utilizes the Raft consensus protocol to maintain fault tolerance and high availability, even amidst orderer node failures.
   - The orderer nodes (orderer, orderer2, orderer3, orderer4, orderer5) constitute a Raft cluster, with one node functioning as the leader and the rest as followers.
   - The leader node manages transaction ordering and replication to follower nodes.
   - In the event of leader node failure, a new leader is elected from the remaining followers, ensuring uninterrupted operation of the ordering service.

2. **Peer Nodes (Two Organizations):**

   - Peer nodes uphold the decentralized ledger and execute smart contracts (chaincode) on behalf of their corresponding organizations.
   - Each organization contributes its own array of peer nodes to the network, fostering a decentralized and distributed ledger.
   - The diagram illustrates two organizations: org1 and org2.

- Org1 hosts two peer nodes (peer0.org1 and peer1.org1) deployed across distinct hosts (Host 2 and Host 3).

- Org2 also features two peer nodes (peer0.org2 and peer1.org2) situated on separate hosts (Host 3 and Host 4).

- Peer nodes within an organization can validate transactions and maintain ledger consistency using a gossip protocol.

- Transactions are added to the ledger once they're endorsed by the requisite number of peer nodes from various organizations.

3. **Command Line Inferface(CLI):**

   - The Command Line Interface (CLI) is a utility enabling administrators, developers, and users to engage with the Hyperledger Fabric network.

   - It furnishes a range of directives for administering the network's lifecycle, such as deploying and revising smart contracts, interrogating the ledger, and triggering transactions.

   - While the image depicts the CLI operating on Host 2, it's accessible from any host linked to the Fabric network.

6.4 DEPLOYEMENT STEPS

(12)

6.4.1 Bring up the 4 VM's

1. **Steps to install docker and docker compose:**

```
sudo apt-get install ca-certificates curl && sudo install -m 0755 -d
/etc/apt/keyrings && sudo curl -fsSL

https://download.docker.com/linux/ubuntu/gpg -o /etc/apt/keyrings/
docker.asc && sudo chmod a+r /etc/apt/keyrings/docker.asc

echo \ "deb [arch=$(dpkg --print-architecture) signedby=/etc/apt/keyrings/
docker.asc] https://download.docker.com/linux/ubuntu \
$(. /etc/os-release && echo "$VERSION_CODENAME") stable" | \
sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
sudo apt-get update
sudo apt-get install docker-ce docker-ce-cli containerd.io docker-
```

```
buildxplugin docker-compose-plugin

sudo chmod 777 /var/run/docker.sock
sudo curl -L
"https://github.com/docker/compose/releases/download/1.29.2/dockercompose-
$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
sudo chmod +x /usr/local/bin/docker-compose
```

2. **Open necessary ports for Docker Swarm:**

```
sudo apt -y install firewalld
sudo systemctl start firewalld
sudo systemctl enable firewalld
sudo firewall-cmd --add-port=2376/tcp --permanent
sudo firewall-cmd --add-port=2377/tcp --permanent
sudo firewall-cmd --add-port=7946/tcp --permanent
sudo firewall-cmd --add-port=7946/udp --permanent
sudo firewall-cmd --add-port=4789/udp --permanent
sudo firewall-cmd --add-port=3001/tcp --permanent
sudo firewall-cmd --reload
sudo systemctl restart docker
```

3. **Install Hyperledger Fabric:**

```
curl -sSLO
https://raw.githubusercontent.com/hyperledger/fabric/main/scripts/
installfabric.sh && chmod +x install-fabric.sh
./install-fabric.sh docker samples binary
```

6.4.2 Creating an overlay with Docker Swarm

1. **On VM1, execute the following commands:**

```
docker swarm init --advertise-addr <VM-1 IP address>
docker swarm join-token manager
```

2. **Make sure to record the output from docker swarm join-token manager as it will be needed immediately in the following steps. Then, on VMs 2, 3, and 4, run:**

```
<output from join-token manager> --advertise-addr <VM n IP>
```

3. **Back on VM1, create an overlay network called 'first-network', which will be used for the network components (refer to each hostn.yaml in the repository):**

```
docker network create --attachable --driver overlay first-network
```

4. **Now we can check each VM. All are sharing the same overlay (note the same network ID in all VM).**

6.4.3 Clone the repository on all hosts

On each host,

```
cd fabric-samples
git clone https://github.com/kctam/4host-swarm.git
cd 4host-swarm
```

6.4.4 Bringing up each host

Clone repository Crypto Material Docker Configs on 4 VM On each host, bring up the host with script hostnup.sh. This script is just a docker-compose up with the corresponding configuration file.

```
/hostnup.sh
```

6.4.5 Bring up mychannel and join all peers to mychannel

This standard process begins by generating the genesis block for mychannel. All peers then join the network using this block file, and updates to the anchor peer transactions are made. To streamline these operations, all commands are run through the CLI on Host 1, and a script has been implemented to facilitate these actions.

On Host 1, run:

```
../mychannelup.sh
```

Once all peers have joined the channel, they should all share the same ledger. To verify this across each host, the following command is used. It checks that all peers are at the same blockchain height (3) and share the same block hash.

```
docker exec peerx.orgy.example.com peer channel getinfo -c mychannel
```

It's crucial to highlight that we are using the docker exec command directly on each peer, instead of going through the CLI. Now that all peers are connected to mychannel, the network is set up and ready for the deployment of chaincode.

6.4.6 Deploy Chaincode

Go inside cli container by running command

```
docker exec -it cli bash

Install git

apt-get update

apt install git-all
```

1. **Package chaincode**

   ```
   pushd ../chaincode/fabcar/go
   GO111MODULE=on go mod vendor
   popd

   docker exec cli peer lifecycle chaincode package fabcar.tar.gz --path github
   ```

2. **Install chaincode package to all peers**

   ```
   # peer0.org1
   docker exec cli peer lifecycle chaincode install fabcar.tar.gz



   # peer1.org1
   docker exec -e
   CORE_PEER_ADDRESS=peer1.org1.example.com:8051 -e
   CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/
   fabric/peer/crypto/peerOrganizations/org1.example.com/peers/
   peer1.org1.example.com/tls/
   ca.crt cli peer lifecycle chaincode install fabcar.tar.gz

   # peer0.org2
   ```

```
docker exec -e
CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/
hyperledger/fabric/peer/crypto/peerOrganizations/org2.example.com/
users/Admin@org2.example.com/
msp -e
CORE_PEER_ADDRESS=peer0.org2.example.com:9051 -e
CORE_PEER_LOCALMSPID="Org2MSP" -e
CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/
peer/crypto/peerOrganizations/org2.example.com/peers/
peer0.org2.example.com/tls/ca.crt cli peer lifecycle chaincode install
fabcar.tar.gz

# peer1.org2
docker exec -e
CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/
crypto/peerOrganizations/org2.example.com/users/Admin@org2.example.com/
msp -e
CORE_PEER_ADDRESS=peer1.org2.example.com:10051 -e
CORE_PEER_LOCALMSPID="Org2MSP" -e
CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/
peer/crypto/peerOrganizations/org2.example.com/peers/
peer1.org2.example.com/tls/ca.crt cli peer lifecycle chaincode install
fabcar.tar.gz
```

6.5 API SERVER DEPLOYMENT

1. Ngrok serves as a robust tool that establishes a secure pathway to localhost, rendering our API endpoints accessible over the internet. This functionality is pivotal for both testing and development endeavors, as well as for seamlessly integrating our API with external services.

2. Ngrok furnishes a distinct domain, employed in the frontend to execute API calls.

3. To ensure continuous operation of our Go web server even after closing the terminal, we employ the nohup command. This command enables the process to disregard the HUP (hangup) signal, typically sent upon terminal closure.

4. Environment variables can be supplied either directly within the command or via a .env file. Utilizing a .env file offers enhanced security and manageability, particularly for sensitive data such as API keys.

5. To enable the API server functionality, we require the NGROK TOKEN for making local APIs public and the secret key utilized in JWT generation.

6.6 CHALLENGES ENCOUNTERED DURING DEPLOYMENT

1. **Complexity in Setup and Management:** Setting up and managing a Docker Swarm cluster can be complex, especially for me as I was new to Docker or orchestration systems. Configuring networks, managing secrets, and setting up service discovery require a good understanding of Docker concepts.

2. **Docker Swarm Leader Election Issue:** The Docker Swarm network deployment was completed successfully, and all systems connected to the network. However, they faced an issue with the election of a raft leader, which prevented the creation of a blockchain channel. Upon thorough investigation, it was found that the swarm communication ports had not been properly configured. The problem was addressed by configuring the firewalld package to allow traffic through the essential ports needed for swarm communication.

3. **Docker Container Communication Issue:** The Docker containers set up for the blockchain were running but could not establish communication, which hindered the creation of the blockchain. The issue was traced back to TCP checksums. By using the ethtool package to disable TCP checksums on each Docker container, communication was re-established, allowing for the successful creation of the blockchain.

CHAPTER 7

CHALLENGES

1. **Architectural Complexity:**
   The Hyperledger Fabric architecture is notably intricate. In healthcare scenarios, where there is a need to store MRI reports and high-resolution images, Fabric offers limited database support.

2. **Limited Database Support:**
   Hyperledger Fabric primarily supports only onlyLevelDB and CouchDB as its state databases. We have used **firebase** here. We have used firebase because it supports Real-Time Data Synchronization, Built-in Authentication and Security, and as a managed service provided by Google, it takes care of scalability, maintenance, and security patches automatically. Hyperledger Fabric does not natively support Firebase. However we can integrate Firebase with a Hyperledger Fabric application for certain functionalities like user authentication, event notifications, or external data storage, we can do so programmatically. This typically involves writing custom application logic within the client application or chaincode that interacts with Firebase through its API. This sort of integration would require careful architectural planning to ensure security and efficiency, particularly given the different nature of data handled by Firebase and the requirements of a blockchain application in terms of immutability and auditability.

3. **Limited API's Support:**

   a) **Native Cryptocurrency APIs:** Unlike blockchain platforms like Ethereum, Hyperledger Fabric does not have a native cryptocurrency or token. Therefore, it does not support APIs related to cryptocurrency transactions, wallet management, or token generation and distribution.

   b) **Public Internet-Facing APIs:** Hyperledger Fabric is typically used in permissioned, consortium blockchain environments and does not provide public APIs for open internet-facing applications without appropriate gateways and access controls designed by the application developer.

   c) **Mobile-Specific APIs:** There are no specific APIs provided by Hyperledger Fabric for mobile device management or mobile-specific functionalities. Any mobile integration requires external tools or services to interface with the blockchain network.

4. **Access User Attributes Using Client:**
   In this situation, since the doctor is not an asset on the ledger, their name and speciality are saved in the identity attributes. However, patients on the blockchain network are unable to access these attributes because they do not have permission to read the doctor's details. Only the admin user can retrieve this information. This limitation arises because the system does not allow for distinct permission settings for different clients, treating all clients uniformly.

5. **Challenges in Developing Application**

   a) Significant challenges arise in securing patient data at the peer level within the ledger. The private data collection method cannot be used due to the issues with the fabric infrastructure previously mentioned. Additionally, the data re-encryption strategy is hindered by two main problems. Firstly, Node.js lacks robust libraries for re-encryption. Although several libraries are suggested, but they prove ineffective. One library that does function does not support the typical formats used for system-generated private and public keys, which it cannot interpret. While Node-RSA is effective for public-key cryptography and accepts standard key formats, it lacks functionality for re-encryption. This gap presents an opportunity to develop and implement a re-encryption algorithm, which, although time-consuming, could be a valuable project that contributes to the community.

   b) Another issue arises during user creation; certificates and private keys are generated successfully, but the public key remains elusive within the Fabric framework, necessitating further investigation to resolve this problem and enable effective re-encryption.

   c) A further challenge involves scaling peers within hospital organizations. For unspecified reasons, one fabric-peer container consistently fails when the network is activated. Additionally, not all peer configurations are captured when generating ccp files, indicating that improvements are needed in documentation and configuration processes.

CHAPTER 8

FUTURE SCOPE

1. Numerous enhancements can be made to advance the solution toward a production-grade application. Despite the inherent security features provided by blockchain and Hyperledger Fabric, there are still security challenges that need to be addressed to ensure the protection of patient records. While the Fabric framework is inherently modular, the underlying code could be further refined to enhance its adaptability, particularly when integrating additional hospitals and their peers. As the network expands, incorporating more organizations and peers into the channel, the system will require additional ordering peers to manage the growing volume of transaction requests and approvals efficiently. Apache Kafka, an open-source distributed event streaming platform, shows great potential for managing multiple ordering nodes effectively.

2. Network communications within the network should utilize HTTPS to provide transport-level security (TLS). Currently, passwords are transmitted from the front end to the back end in plain text, which can be significantly secured by implementing HTTPS. Additionally, using email for temporary password delivery is an effective solution. Incorporating a password reset feature is also advisable. Enhancing the user interface and user experience (UI/UX) can significantly improve user interactions. The addition of search functionality could be beneficial as patient data volume increases. However, it is necessary to investigate whether Hyperledger Fabric supports wildcard searches. It is also crucial to recognize that since the data does not originate from a single database, frequent search queries may not be feasible.

3. Construct the consortium policy to meet the minimum requirements of the consensus algorithm. Hyperledger Fabric employs pBFT, wherein a transaction requires approval from all involved peers, such as a 75% approval rate—meaning approval from three out of four peers is enough to execute a transaction. Allocate the wallet according to the distribution of the organization's peers. Users generated within the wallet should be associated with their respective organizational peers. Additionally, the wallet could be housed in a no-SQL database and replicated across multiple nodes to prevent data loss.

4. The thesis may explore further applications of blockchain technology within the hospital management system, such as extending its use to other areas of healthcare data management and operations.

5. The system might aim to enhance interoperability with other health information systems, facilitating seamless data exchange and integration across different platforms and healthcare providers.

6. Given the emphasis on security against attacks like ransomware, network sniffing, and DDOS, future work could involve developing more robust security protocols or enhancing the existing blockchain framework to better protect against these threats.

7. The thesis might look into integrating additional emerging technologies, such as AI and machine learning, to enhance decision-making processes and predictive analytics in the hospital management system.

REFERENCES

[1] *Blockchain denial of service*.

[2] *Blockchain-Enabled Telehealth Services Using Smart Contracts*.

[3] *A Decentralized Electronic Prescription Management System*.

[4] *Distributed Denial of Service (DDoS) Mitigation Using Blockchain*.

[5] *Docker Documentation*.

[6] *Firebase*.

[7] *HapiChain: A Blockchain-based Framework for Patient-Centric Telemedicine*.

[8] *HealthBlock: A secure blockchain-based healthcare data management system*.

[9] *Healthchain: Framework on privacy preservation of EHR using blockchain*.

[10] *Hyperledger Fabric Documentation*.

[11] *MedicalChain Whitepaper*.

[12] *Multi-Host Deployment for First Network*.

[13] *A Novel EMR Integrity Management Based on a Medical Blockchain Platform*.

[14] *React native*.

[15] *Security against Ransomware Attack in Medical Healthcare Records*.

[16] *Security framework in blockchain against ransomware attacks for smart healthcare*.

[17] *Shaping the health data future – the DayOne Scenarios*.

[18] *Utilization of blockchain for mitigating the distributed denial of service attacks*.

(15) (16) (8) (4) (1) (18) (11) (17) (7) (3) (13) (2) (9) (10) (12) (5)