

INDEX

1. Introduction
 - 1.1. Machine Learning
 - 1.2. HyperLedger Fabric
 - 1.3. Federated Learning
 - 1.4. Federated Learning steps
 - 1.5. MNIST Data
 - 1.6. Flower Framework
2. Objective
3. Method/Procedure
 - 3.1. Technology Used
 - 3.2. Federating learning Model using Flower Framework
 - 3.3. Configuration of Network
 - 3.4. Creation of Container in docker desktop
 - 3.5. Creating Channel
 - 3.6. Deploying Chaincode
 - 3.7. Smart Contract
 - 3.8. API
 - 3.9. Working
 - 3.10. Network Configuration
4. Result
 - 4.1. Federated Averaging
 - 4.2. Federated Proximal
 - 4.3. Comparison
5. Conclusion
6. Future Prospect
7. References

1. Introduction

1.1. Machine learning

Machine learning (ML) is a branch of artificial intelligence (AI) and computer science that focuses on the using data and algorithms to enable AI to imitate the way that humans learn, gradually improving its accuracy.

The machine learning algorithm can be broken down into three parts

1. A Decision Process: In general, machine learning algorithms are used to make a prediction or classification. Based on some input data, which can be labeled or unlabeled, your algorithm will produce an estimate about a pattern in the data.
2. An Error Function: An error function evaluates the prediction of the model. If there are known examples, an error function can make a comparison to assess the accuracy of the model.
3. A Model Optimization Process: If the model can fit better to the data points in the training set, then weights are adjusted to reduce the discrepancy between the known example and the model estimate. The algorithm will repeat this iterative “evaluate and optimize” process, updating weights autonomously until a threshold of accuracy has been met.

The various methods of machine learning are-

1. Supervised Machine learning-Supervised Learning, also known as supervised machine learning, is defined by its use of labeled datasets to train algorithms to classify data or predict outcomes accurately. As input data is fed into the model, the model adjusts its weights until it has been fitted appropriately. This occurs as part of the cross validation process to ensure that the model avoid overfitting or underfitting. Supervised learning helps organizations solve a variety of real-world problems at scale, such as classifying spam in a separate folder from your inbox. Some methods used in supervised learning include neural networks, naïve bayes, linear regression, logistic regression, random forest, and support vector machine (SVM).

2. Unsupervised machine learning- It is also known as unsupervised machine learning, uses machine learning algorithms to analyze and cluster unlabeled datasets (subsets called clusters). These algorithms discover hidden patterns or data groupings without the need for human intervention. This method's ability to discover similarities and differences in information make it ideal for exploratory data analysis, cross-selling strategies, customer segmentation, and image and pattern recognition. It's also used to reduce the number of features in a model through the process of dimensionality reduction. Principal component analysis (PCA) and singular value decomposition (SVD) are two common approaches for this. Other algorithms used in unsupervised learning include neural networks, k-means clustering, and probabilistic clustering methods.

3- Semi-supervised learning - Semi-supervised learning offers a happy medium between supervised and unsupervised learning. During training, it uses a smaller labeled data set to guide classification and feature extraction from a larger, unlabeled data set. Semi-supervised learning can solve the problem of not having enough labeled data for a supervised learning algorithm. It also helps if it's too costly to label enough data.

The various machine learning algorithms are-

- Neural networks: Neural networks simulate the way the human brain works, with a huge number of linked processing nodes. Neural networks are good at recognizing patterns and play an important role in applications including natural language translation, image recognition, speech recognition, and image creation.
- Linear regression: This algorithm is used to predict numerical values, based on a linear relationship between different values. For example, the technique could be used to predict house prices based on historical data for the area.
- Logistic regression: This supervised learning algorithm makes predictions for categorical response variables, such as "yes/no" answers to questions. It can be used for applications such as classifying spam and quality control on a production line.

- **Clustering:** Using unsupervised learning, clustering algorithms can identify patterns in data so that it can be grouped. Computers can help data scientists by identifying differences between data items that humans have overlooked.
- **Decision trees:** Decision trees can be used for both predicting numerical values (regression) and classifying data into categories. Decision trees use a branching sequence of linked decisions that can be represented with a tree diagram. One of the advantages of decision trees is that they are easy to validate and audit, unlike the black box of the neural network.
- **Random forests:** In a random forest, the machine learning algorithm predicts a value or category by combining the results from a number of decision trees.

1.2. HyperLedger Fabric

Hyper Leger Fabric is a permission blockchain which is made up of various unique organization that interact through each other to serve a specific purpose. The member of a hyperledger fabric is enrolled through a trusted Membership Service provider. In this, a group of participants can create a separate a ledger of transaction known as channel. This gives participants an option of private transactions. Hyperledger smart contracts are written in chaincode.

Features of Hyperledger Fabric-

1. **Assets-** Assets is any tangible or intangible material which the hyperledger fabric can modify using chaincode. They are represented as key-value pair.

2. **Chaincode-** Chaincode is software which defines an asset and the instruction for modifying them. It executes on a database ledger and is initiated through a transaction proposal.

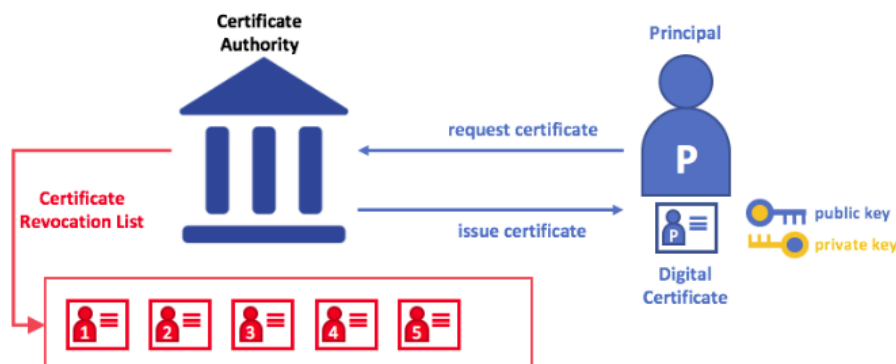
3. Ledger- The ledger comprises of block to store immutable , sequenced record in blocks, as well as a state database to maintain current fabric state. There is one ledger per channel. Every peer on the channel contain the copy of ledger of that channel.

4. Consensus- Consensus is an algorithm which is responsible for the verification of the correctness of a set of transactions comprising a block. It includes the usage of endorsement policies to dictate which specific member can endorse the transaction as well as chaincode to ensure that these policies are enforced and upheld.

5. Ledger- Hyperledger fabric employs ledger on a channel basis so that the ledger can be modified only by the participants of that channel. When a subset of organizations on that channel need to keep their transaction data confidential, a private data collection (collection) is used to segregate this data in a private database, logically separate from the channel ledger, accessible only to the authorized subset of organizations.

6. Identity- A hyperledger fabric includes various peers, orderers, client application etc. Each of these elements has a digital identity encapsulated in a digital certificate. This identity is used to determine the permissions over a resource and the access to a network element in the blockchain. The union of an identity and the associated attribute it has forms a principal.

Public Key Infrastructure



Public key infrastructure (PKI) is a collection of technologies that provide secure communication. It is a set of roles, policies and procedures which are used to create, manage, distribute, store and revoke digital certificates and manage public key encryption. Hyperledger fabric uses PKI to ensure secure communication between various network participants, and to ensure that messages posted on the blockchain are properly authenticated.

There are four key elements to PKI

- Digital Certificate- A digital certificate is a document which holds a set of attributes relating to the holder of the certificate. This certificate is generated by a certificate authority. This certificate allows a participant to prove his identity to other party so long as the other party trust on the certificate issuer.
- Public and Private key- The receiving party should be able to authenticate the certificate to ensure that the certificate comes from correct sender. For authentication, the certificate should be digitally signed. Digital signature mechanism requires two party to hold cryptographically connected key(Public and Private key). The sender signed the message using its private key. The receipt can check the authenticity of the message using the public key.
- Certificate Authority- A certificate authority is used to dispense the certificate to various actors. These certificates are signed by the CA and bind actor with its public key.

There are two type of CA-

- Root CA- Which issues certificate to both actors and intermediate CA
- Intermediate CA- Which issues certificate to actor only.
- Certificate Revocation List- It contains a list of certificate that CA has to revoke for some reason.

Membership Service Provider

Membership service provide play an important role in Hyperledger fabric. It identifies which certificate authority are accepted to define the member of a channel. It then turn the identity of the member to role by indentifying privileges and permission the member has over the channel. The various role can be a admin, peer, client, orderer

MSP has two type of scope in a blockchain-

- Local MSP
- Channel MSP

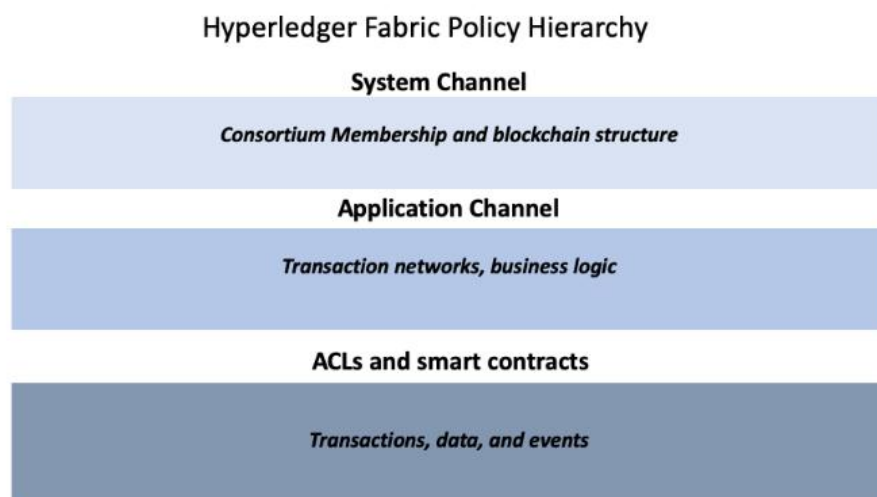
Local MSP- They are defined for peers, orderers and client. Every node in the chain should has a MSP define for it. It defines who has administrative or participatory right at that level. It also defines the organization admins.

Channel MSP- Unlike local MSP, channel MSP is defined at the channel level. Peer and ordering node share the same view of the channel MSP, and is responsible for correctly authenticating the channel participants. Every organization participating in a channel should have a MSP defined for it.

Channel MSP is instantiated on the file system of every node in the channel

Policy

Policy is a set of rules that define how the system should works and how decisions are made. It defines what rights and access an individual have over some asset. They are used for infrastructure management of a fabric. Since the fabric is a permission blockchain, these policies defines which users have the ability to decide on the governance of the system and which can change the system. It also defines how many organization and individual need to agree on a proposal to update a proposal.

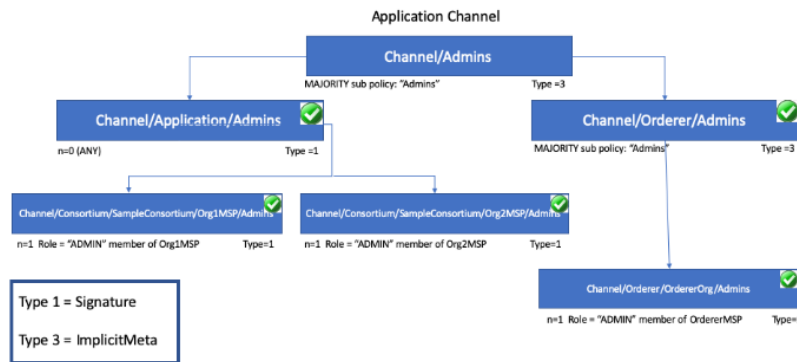


Policies can be implemented at different levels in the chain.

- **System Channel Configuration-** This is also known as ordering system channel. There must be exactly one ordering system channel for an ordering system. The policies in this system govern the consensus used by the ordering service and how new blocks are created. This also defines which member of the consortium is allowed to make the channel.
- **Application Channel Configuration-** The policies at this level govern the ability to add or remove member from the channel. It also defines which organization has to approve the chaincode before it is committed to channel.
- **Access Control List-** The policies in this are used to define the access to resources by different member. This defines whether a member has access to a resource or not.
- **Smart Contract Endorsement Policy-** This policy is stored inside the chaincode of the respective smart contract. This defines how many peers belonging to different channel should endorse a transaction before it can make changes in the ledger and be considered valid. It also defines which peer can endorse a transaction.
- **Modification Policy-** Modification policies are a special kind of policy that defines how a policy can be updated. It defines the group of organization which must sign to update the policy.

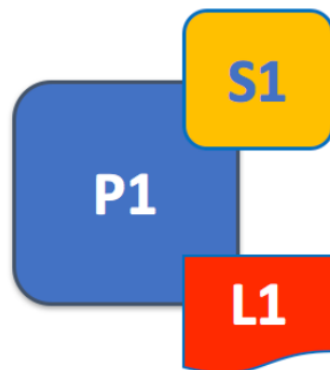
Types of Policies

- **Signature Policies-** These are explicitly signed policies. This defines the specific type of user who must sign in order for a policy to be satisfied. They allow for the construction of extremely specific rules.
- **Implicit Meta Policies-** They are implicitly signed policies. These policies are valid in the context of channel configuration. They aggregate the result of policies deeper in the configuration tree. The policies at the bottom are generally signature policies.



Peers

Peers are a fundamental element of a blockchain and host a ledger and a smart contracts. Peers can be created, started, stopped, configured and even deleted. Peers communicate with each other with the help of channel. A peer can only communicate to only those peers which are connected to it through channel. The peers are contributed by different organization which comes together to form this network. When a peer connects to a channel, its digital certificate identifies its owning organization via a channel MSP.



An application must interact with peer if they want to change the ledger or invoke a smart contract. For any operation on peer, applications need to connect to peers, invoke chaincodes to generate transactions, submit transactions to the network that will get ordered, validated and committed to the distributed ledger, and receive events when this process is complete. There are two type of operations that can be performed on peer

- Query- In query operation, we try to send some response from peer to application. In this, the application invoke a chaincode to query the

ledger. The peer responds by generating a proposal response that contains the query result. This proposal is then sent back to the application.

- Update- In this, the application invoke a chaincode to update the ledger. The peer response with a proposal response. Application send the code to ordering node. Ordering node collect the transaction across the network into block and distribute these to all peers. The peers then validate this node and after validation, commit the changes to the ledger.

Application that wants to update the ledger are involved in a 3-phase process

- Phase 1 (Proposal)- Phase 1 of the transaction workflow involves an interaction between an application and a set of peers. Phase 1 is only concerned with an application asking different organizations' endorsing peers to agree to the results of the proposed chaincode invocation. Application generate a transaction protocol which they send to each of the required node for endorsement. Each of this node then executes a chaincode to generate a transaction protocol response. It does not apply this update to the ledger, but rather simply signs it and returns it to the application. Once the application has received a sufficient number of signed proposal responses, it moves to phase 2.
- Phase 2 (Packing into Blocks)- In Phase 2, the application send the transaction responses it has received from peers to ordering nodes. The ordering node receives responses from various application concurrently. It arrange the batches of submitted transaction into a well defined sequence. Then they create and pack the transaction into blocks which will ultimately transferred to all the peers on the channel for final validation.
- Phase 3 (Validation)- It begins with the ordering nodes distributing blocks to all peers connected to it. Each peer will validate the transaction independently but in a determined manner. Validation involves checking whether it has been endorsed by required organization peers, its endorsement match and whether it has not become invalidated by other recently committed transaction.

Ledger

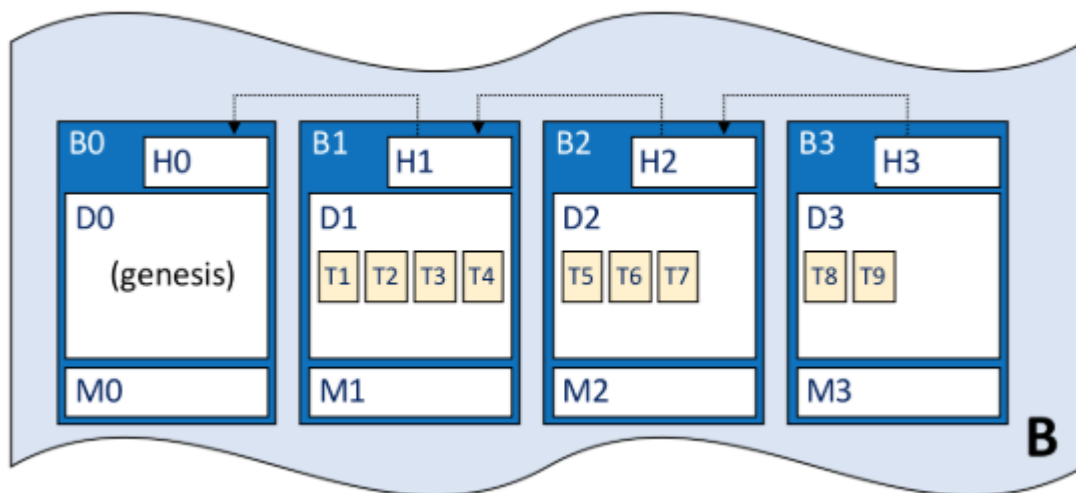
A ledger is a database which is used to information. In hyperledger fabric , it is used to store both mutable and immutable data. Mutable data consist of facts and immutable data consists of history of these facts. The data is stored in the form of key-value pair in ledger. Ledger work on distributed ledger technology where every node in the channel maintain the ledger

In hyperledger fabric, ledger consists of two related parts

- World State(Mutable)
- Blockchain(Immutable)

World State- The world state holds the current value of the attributes of a business object as a unique ledger state. This prevent the problem of traversing the whole ledger to find the value of an object. The worldstate is implemented as a database. The world state can be accessed and updated only through smart contracts.

Blockchain- Blockchain consists the historical record of the world state. This is of immutable type means it cannot be changed.



Each block header consists of a hash of the block transaction, hash of the previous block transaction, block data which consists of the transaction and block metadata.

The transaction further contain various fields like-

- Header

- Signature
- Proposal
- Response
- Endorsement

Ordering

Unlike permission blockchain which are based on probabilistic consensus algorithm, hyperledger fabric is based on deterministic consensus algorithm. It features a node called as orderer that does the transaction ordering. Any block validated by the peer is guaranteed to be final and correct. Ledger cannot fork them. Ordering nodes belong to an organization. Ordering nodes also maintain a list of organization that are allowed to create channel. This list is called consortium and is kept in the configuration of the orderer system channel. It enforces basic access control of the channel, restricting who can read and write data to them and who can configure them. It plays an important role in updating the ledger. In Phase 2, it collects the transaction proposal it has received from applications, puts them in a block and sends them to all the nodes for validation and updating of the ledger.

Smart Contracts

Smart contract defines the executable logic that generates new facts that are added to the ledger. It defines the rules between different organizations in executable code. Applications invoke a smart contract to generate transactions that are recorded on the ledger. A smart contract is defined within a chaincode. Multiple smart contracts can be defined within the same chaincode. When a chaincode is deployed, all smart contracts within it are made available to applications.

Smart contracts primarily put, get and delete states in the world state, and can also query the immutable blockchain record of transactions.

- A get typically represents a query to retrieve information about the current state of a business object.
- A put typically creates a new business object or modifies an existing one in the ledger world state.
- A delete typically represents the removal of a business object from the current state of the ledger, but not its history.

Every chaincode has an endorsement policy that should be fulfilled for the successful execution of a smart contract. It defines which organization in a blockchain network must sign a transaction generated by a given smart contract.

A Smart Contract can call other smart contracts both within the same channel and across different channels. In this way, they can read and write world state data to which they would not otherwise have access due to smart contract namespaces.

Chain Code

A chaincode is typically used by administrators to group related smart contracts for deployment, but can also be used for low level system programming of Fabric. Channel members can only execute a smart contract after the chaincode has been defined on a channel. The chaincode definition is a struct that contains the parameters that govern how a chaincode operates. These parameters include the chaincode name, version, and the endorsement policy. Each channel member agrees to the parameters of a chaincode by approving a chaincode definition for their organization. When a sufficient number of organizations (a majority by default) have approved to the same chaincode definition, the definition can be committed to the channel. The chaincode definition provides a way for channel members to agree on the governance of a chaincode before they start using the smart contract to transact on the channel.

Channel

A channel provides a completely separate communication mechanism between a set of organizations. When a chaincode definition is committed to a channel, all the smart contracts within the chaincode are made available to the applications on that channel.

Working of Hyper Ledger Fabric

The Hyperledger fabric working is a three phase process in which each process has defined role.

- Phase 1 (Proposal)- Phase 1 of the transaction workflow involves an interaction between an application and a set of peers. Phase 1 is only concerned with an application asking different organizations' endorsing

peers to agree to the results of the proposed chaincode invocation. Application generate a transaction protocol which they send to each of the required node for endorsement. Each of this node than executes a chaincode to generate a transaction protocol response. It does not apply this update to the ledger, but rather simply signs it and returns it to the application. Once the application has received a sufficient number of signed proposal responses, it moves to phase 2.

- Phase 2 (Packing into Blocks)- In Phase 2, the application send the transaction responses it has received from peers to ordering nodes. The ordering node receives responses from various application concurrently. It arrange the batches of submitted transaction into a well defined sequence. Then they create and pack the transaction into blocks which will ultimately transferred to all the peers on the channel for final validation.
- Phase 3 (Validation)- The third phase of the transaction workflow involves the distribution and subsequent validation of blocks from the orderer to the peers, where they can be committed to the ledger It begins with the ordering nodes distributing blocks to all peers connected to it. Each peer will validate the transaction independently but in a determined manner. Validation involves checking whether it has been endorsed by required organization peers, its endorsement match and whether it has not become invalidated by other recently committed transaction. In summary, phase three sees the blocks generated by the ordering service applied consistently to the ledger

1.3. Federated Learning

Federated learning is a distributed machine learning paradigm that enables machine learning models to be trained on decentralized data sources without compromising privacy or security. In traditional machine learning, the data is centralized, meaning that the data is collected, stored, and processed in a single location. This data centralization can create privacy and security issues, as sensitive information is often collected, stored, and processed in this manner.

In contrast, federated learning enables machine learning models to be trained on decentralized data sources by allowing multiple parties to train models on their local data while sharing the learned parameters with the central server. This allows the server to aggregate the models and update the global model while preserving the privacy and security of the data.

Advantages of Federated Learning

- Privacy- In federated learning, the data is not centralized, that a single machine don't has access to all data. This makes federated learning secure for sensitive data.
- Decentralization- The data is decentralized on the local system. The global system has access only to model not to data.

Disadvantages of Federated Learning

- Latency- It take long to model to update. It makes the system slow.

Types of Federated Learning

- Horizontal Federated Learning- Data samples are distributed across devices or services, and the model is trained collaboratively.
- Vertical Federated Learning- Features are divided between devices and the model is trained on complementary features.
- Federated Transfer Learning- Pre-trained models are fine- tuned on decentralized data for specific tasks, reducing the need for extensive local data.

Federated Learning Algorithm

- **Federated stochastic gradient descent** - It's a distributed form of Stochastic Gradient Descent (SGD), tailored to the federated learning setting where multiple clients collaboratively train a global model without sharing their private data.
- **Federated Averaging**- Federated averaging (FedAvg) allows local nodes to perform more than one batch update on local data and exchanges the updated weights rather than the gradients. The rationale behind this generalization is that in FedSGD, if all local nodes start from the same initialization, averaging the gradients is strictly equivalent to averaging the weights themselves. Further, averaging tuned weights coming from the same initialization does not necessarily hurt the resulting averaged model's performance.
- **Federated Learning with Dynamic Regularization- FedDyn** (Federated Dynamic Regularization) is an advanced optimization algorithm designed to address the challenges in federated learning, particularly when dealing with non-IID (non-independent and identically distributed) data across clients. FedDyn introduces a dynamic regularization term that adjusts based on the difference between the local and global models, effectively aligning the local training objectives with the global objective. Each client in FedDyn performs local updates by minimizing a dynamically regularized loss function. After a few local iterations, clients send their updated model parameters to a central server. The server aggregates these parameters, incorporating the regularization effect, to update the global model.
- **Federated Adaptive Moment Estimation- Federated Adaptive Moment Estimation**, is an optimization algorithm tailored for federated learning environments. FedAdam addresses heterogeneous and non-IID data distributions by leveraging Adam's adaptive learning rates and moment estimates, which combine the advantages of AdaGrad and RMSProp. In FedAdam, each client performs local training on its data using the Adam optimizer, computes gradients, and periodically sends these gradients to a central server. The server aggregates the gradients, updates the moment estimates, and applies the Adam update rules to refine the global model.

IID's and Non-IID's

IID stands for Independent and Identically Distributed data. These data points are generated independently of each other. The occurrence of one data point does not influence the occurrence of another. There is no correlation between data points. All data points are drawn from the same probability distribution. Each data point has the same statistical properties, such as mean and variance.

Non-IID stands for Non-Independent and Identically Distributed data. These data points are not generated independently. There may be correlations or dependencies between them. Data points are drawn from different distributions, which means the statistical properties can vary across subsets of the data. This can happen due to different sources, contexts, or conditions under which the data is collected. The presence of dependencies and varying distributions complicates the analysis and design of algorithms.

1.4. Federated learning in five steps

Step 0: Initialize global model

We start by initializing the model on the server. This is exactly the same in classic centralized learning: we initialize the model parameters, either randomly or from a previously saved checkpoint.

Step 1: Send model to a number of connected organizations/devices (client nodes)

Next, we send the parameters of the global model to the connected client nodes (think: edge devices like smartphones or servers belonging to organizations). This is to ensure that each participating node starts their local training using the same model parameters. We often use only a few of the connected nodes instead of all nodes. The reason for this is that selecting more and more client nodes has diminishing returns.

Step 2: Train model locally on the data of each organization/device (client node)

Now that all (selected) client nodes have the latest version of the global model parameters, they start the local training. They use their own local dataset to train their own local model. They don't train the model until full convergence, but they only train for a little while. This could be as little as one epoch on the local data, or even just a few steps (mini-batches).

Step 3: Return model updates back to the server

After local training, each client node has a slightly different version of the model parameters they originally received. The parameters are all different because each client node has different examples in its local dataset. The client nodes then send those model updates back to the server. The model updates they send can either be the full model parameters or just the gradients that were accumulated during local training.

Step 4: Aggregate model updates into a new global model

The server receives model updates from the selected client nodes. If it selected 100 client nodes, it now has 100 slightly different versions of the original global model, each trained on the local data of one client. But didn't we want to have one model that contains the learnings from the data of all 100 client nodes?

In order to get one single model, we have to combine all the model updates we received from the client nodes. This process is called aggregation, and there are many different ways to do it. The most basic way to do it is called Federated Averaging ([McMahan et al., 2016](#)), often abbreviated as FedAvg. FedAvg takes the 100 model updates and, as the name suggests, averages them. To be more precise, it takes the weighted average of the model updates, weighted by the number of examples each client used for training. The weighting is important to make sure that each data example has the same "influence" on the resulting global model. If one client has 10 examples, and another client has 100 examples, then - without weighting - each of the 10 examples would influence the global model ten times as much as each of the 100 examples.

Step 5: Repeat steps 1 to 4 until the model converges

Steps 1 to 4 are what we call a single round of federated learning. The global model parameters get sent to the participating client nodes (step 1), the client nodes train on their local data (step 2), they send their updated models to the server (step 3), and the server then aggregates the model updates to get a new version of the global model (step 4).

During a single round, each client node that participates in that iteration only trains for a little while. This means that after the aggregation step (step 4), we have a model that has been trained on all the data of all participating client nodes, but only for a little while. We then have to repeat this training process over and over again to eventually arrive at a fully trained model that performs well across the data of all client nodes.

1.5. MNIST Dataset

The MNIST (Modified National Institute of Standards and Technology) dataset is a renowned and widely used dataset in the field of machine learning and computer vision, specifically for training and testing image classification algorithms. It consists of a large collection of handwritten digits, with a total of 70,000 images divided into two main subsets: a training set of 60,000 images and a test set of 10,000 images. Each image in the dataset is a grayscale image of 28x28 pixels, where each pixel value ranges from 0 to 255, indicating the intensity of the pixel. These images are typically represented as one-dimensional arrays of 784 values (28x28) for ease of processing by machine learning models.

The MNIST dataset was created by modifying two datasets from the original NIST dataset, and it has become a standard benchmark for evaluating the performance of various machine learning algorithms, particularly those involved in image classification. The simplicity and cleanliness of the dataset make it an excellent starting point for those new to machine learning, while its widespread use allows for easy comparison of different algorithms and models. The dataset contains a diverse range of handwritten digits, ensuring that models trained on it can generalize well to different styles of handwriting. Overall, the MNIST dataset is a foundational resource in the machine learning community, enabling researchers and practitioners to develop, test, and refine their algorithms effectively.

1.6. Flower Framework

Flower is an open source framework use to create the federated leaning pipeline. It provides services for various task in federated learning such as data and model partitioning, inter device communication and model averaging. Flower also provide services for various federated learning algorithm like federated averaging, federated stochastic gradient descent, federated learning with dynamic regularization.

Federated learning, federated evaluation, and federated analytics require infrastructure to move machine learning models back and forth, train and evaluate them on local data, and then aggregate the updated models. Flower provides the infrastructure to do exactly that in an easy, scalable, and secure way. In short, Flower presents a unified approach to federated learning, analytics, and evaluation. It allows the user to federate any workload, any ML framework, and any programming language.

2. Objective-

To implement a federated learning model using HyperLedger Fabric Blockchain and to compare the accuracy and loss of different federated learning aggregation methods on different type of identical and independent data(IID's) and non-identical and independent data(Non-IID's).

3. Method

Link of Repository used for reference

<https://github.com/adhavpavan/BasicNetwork-2.0>

Link of my Repository - <https://github.com/Adityagarg8384/Federated-Learning-Using-Hyperledger-fabric>

3.1. Technology Used

- Windows Subsystem Linux(WSL): WSL is used to run the linux operating system on the windows. It is necessary to run the various Linux operating system based command for creating the artifacts, creating the channel, deploying chaincode and creating the container on docker desktop.
- Docker-Desktop :Docker is used to run various peers, orderers and certificate authority(ca) component in isolated container. These containers are used to store the data and information of the channel and network.
- Postman- Postman is used to make various api call for function like registering the user, registering the admin, fetching specific data, etc.
- Python- The Federated Learning model is developed using the python. All various api has been written in python .Python modules used in the project are torch, flwr, flask, pickle, Dictconfig, requests.
- Go- Go is used to write the smartcontract of the system.
- Javascript- The Api for registering the user, registering admin, fetching user and other various function for invoking the smartcode function is written in javascript.

3.2. Federating learning Model using Flower Framework-

Step-1 Loading the data-

We simulate having multiple datasets from multiple organizations (also called the “cross-silo” setting in federated learning) by splitting the original MNIST dataset into multiple partitions. Each partition will represent the data from a single organization.

```
def get_mnist(datapath: str = "./data"):
    logging.info("Downloading MNIST dataset")
    transform = Compose([ToTensor(), Normalize((0.1307,), (0.3081,))])
    trainset = MNIST(datapath, train=True, download=True, transform=transform)
    testset = MNIST(datapath, train=False, download=True, transform=transform)
    logging.info("MNIST dataset downloaded")
    return trainset, testset
```

```
def preparedatasets(num_clients: int, batch_size: int, val_ratio: float = 0.1):
    logging.info(f"Preparing dataset for {num_clients} clients with batch size {batch_size} and validation ratio {val_ratio}")
    trainset, testset = get_mnist()

    num_images = len(trainset) // num_clients # Use integer division
    remainder = len(trainset) % num_clients # Calculate any remainder
    logging.info(f"Each client will have approximately {num_images} images, with {remainder} images distributed to some clients")

    partition_lengths = [num_images] * num_clients
    for i in range(remainder): # Distribute the remainder images
        partition_lengths[i] += 1

    trainsets = random_split(trainset, partition_lengths, torch.Generator().manual_seed(2023))
    logging.info(f"Train dataset partitioned into {num_clients} subsets")

    trainloaders = []
    valloaders = []
    for idx, subset in enumerate(trainsets):
        num_total = len(subset)
        num_val = int(val_ratio * num_total)
        num_train = num_total - num_val
        train_subset, val_subset = random_split(subset, [num_train, num_val], torch.Generator().manual_seed(2023))
        trainloaders.append(DataLoader(train_subset, batch_size=batch_size, shuffle=True, num_workers=2))
        valloaders.append(DataLoader(val_subset, batch_size=batch_size, shuffle=False, num_workers=2))
        logging.info(f"Client {idx}: {num_train} training samples, {num_val} validation samples")

    testloader = DataLoader(testset, batch_size=128)
    logging.info("Test dataset prepared")

    return trainloaders, valloaders, testloader
```

We now have a list of training sets and validation sets (trainloaders and valloaders) representing the data of different organizations. Each trainloader/valloader pair contains 4000 training examples and 1000 validation examples.

Step-2 Defining the model and training the data on it-

The next step in federated learning model is to define the model. In this project I have defined a CNN model.

```
class Net(nn.Module):  
  
    def __init__(self, num_classes: int) -> None:  
        super(Net, self).__init__()  
        self.conv1 = nn.Conv2d(1, 6, 5)  
        self.pool = nn.MaxPool2d(2, 2)  
        self.conv2 = nn.Conv2d(6, 16, 5)  
        self.fc1 = nn.Linear(16 * 4 * 4, 120)  
        self.fc2 = nn.Linear(120, 84)  
        self.fc3 = nn.Linear(84, num_classes)  
  
    def forward(self, x: torch.Tensor) -> torch.Tensor:  
        x = self.pool(F.relu(self.conv1(x)))  
        x = self.pool(F.relu(self.conv2(x)))  
        x = x.view(-1, 16 * 4 * 4)  
        x = F.relu(self.fc1(x))  
        x = F.relu(self.fc2(x))  
        x = self.fc3(x)  
        return x
```

Once the model is defined we need to define the function for training and validation of the dataset.

```
def train(net, trainloader, optimizer, epochs, device: str):  
  
    logging.info("Starting training")  
    criterion = torch.nn.CrossEntropyLoss()  
    net.train()  
    net.to(device)  
    for epoch in range(epochs):  
        running_loss = 0.0  
        for i, (images, labels) in enumerate(trainloader, 0):  
            images, labels = images.to(device), labels.to(device)  
            optimizer.zero_grad()  
            outputs = net(images)  
            loss = criterion(outputs, labels)  
            loss.backward()  
            optimizer.step()  
  
            # Print statistics  
            running_loss += loss.item()  
            if i % 100 == 99: # Print every 100 mini-batches  
                logging.info(f"[Epoch {epoch + 1}, Batch {i + 1}] loss: {running_loss / 100:.3f}")  
                running_loss = 0.0  
    logging.info("Finished training")
```

```
def test(net, testloader, device: str):
    logging.info("Starting evaluation")
    criterion = torch.nn.CrossEntropyLoss()
    correct, total, loss = 0, 0, 0.0
    net.eval()
    net.to(device)
    with torch.no_grad():
        for data in testloader:
            images, labels = data[0].to(device), data[1].to(device)
            outputs = net(images)
            loss += criterion(outputs, labels).item()
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
    accuracy = correct / total
    logging.info(f"Test loss: {loss:.3f}, Accuracy: {accuracy:.3f}")
    return loss, accuracy
```

We now have all the basic building blocks we need: a dataset, a model, a training function, and a test function. Let's put them together to train the model on the dataset of one of our organizations (trainloaders[0]). This simulates the reality of most machine learning projects today: each organization has their own data and trains models only on this internal data.

```
def evaluate(self, parameters: NDArrays, config: Dict[str, Scalar]):
    logging.info("Starting evaluation")
    self.set_parameters(parameters)
    loss, accuracy = test(self.model, self.valloader, self.device)
    logging.info(f"Evaluation completed with loss={loss}, accuracy={accuracy}")
    return float(loss), len(self.valloader), {"accuracy": accuracy}
```

Step-3 Defining the model parameter-

In federated learning, the server sends the global model parameters to the client, and the client updates the local model with the parameters received from the server. It then trains the model on the local data (which changes the model parameters locally) and sends the updated/changed model parameters back to the server (or, alternatively, it sends just the gradients back to the server, not the full model parameters).

We need two helper functions to update the local model with parameters received from the server and to get the updated model parameters from the local

model: `set_parameters` and `get_parameters`. The following two functions do just that for the PyTorch model above.

```
def set_parameters(self, parameters):
    params_dict = zip(self.model.state_dict().keys(), parameters)
    state_dict = OrderedDict({k: torch.Tensor(v) for k, v in params_dict})
    self.model.load_state_dict(state_dict, strict=True)
    logging.info("Model parameters set")

def get_parameters(self, config: Dict[str, Scalar]):
    logging.info("Getting model parameters")
    return [val.cpu().numpy() for _, val in self.model.state_dict().items()]
```

Step-4 Implementing a Flower Client

Federated learning systems consist of a server and multiple clients. In Flower, we create clients by implementing Subclasses of `flwr.client.Client` and `flwr.client.NumPyClient`.

To implement the Flower client, we create a subclass of `flwr.client.NumPyClient` and implement the three methods `get_parameters`, `fit`, and `evaluate`:

- `get_parameters`: Return the current local model parameters
- `fit`: Receive model parameters from the server, train the model parameters on the local data, and return the (updated) model parameters to the server
- `evaluate`: Receive model parameters from the server, evaluate the model parameters on the local data, and return the evaluation result to the server

```
class FlowerClient(fl.client.NumPyClient):

    def __init__(self, trainloader, valloader, num_classes) -> None:
        super().__init__()

        self.trainloader = trainloader
        self.valloader = valloader

        self.model = Net(num_classes)

        self.device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

        logging.info(f"FlowerClient initialized with device {self.device}")

    def set_parameters(self, parameters):
        params_dict = zip(self.model.state_dict().keys(), parameters)
        state_dict = OrderedDict({k: torch.Tensor(v) for k, v in params_dict})
        self.model.load_state_dict(state_dict, strict=True)
        logging.info("Model parameters set")
```

```

def get_parameters(self, config: Dict[str, Scalar]):
    logging.info("Getting model parameters")
    return [[val.cpu().numpy() for _, val in self.model.state_dict().items()]]

def fit(self, parameters, config):
    logging.info("Starting training")
    self.set_parameters(parameters)

    lr = 0.01
    momentum = 0.9
    epochs = 1
    logging.info(f"Training with lr={lr}, momentum={momentum}, epochs={epochs}")

    optim = torch.optim.SGD(self.model.parameters(), lr=lr, momentum=momentum)
    train(self.model, self.trainloader, optim, epochs, self.device)

    logging.info("Training completed")
    return self.get_parameters({}), len(self.trainloader), {}

def evaluate(self, parameters: NDArrays, config: Dict[str, Scalar]):
    logging.info("Starting evaluation")
    self.set_parameters(parameters)
    loss, accuracy = test(self.model, self.valloader, self.device)
    logging.info(f"Evaluation completed with loss={loss}, accuracy={accuracy}")
    return float(loss), len(self.valloader), {"accuracy": accuracy}

```

Our class `FlowerClient` defines how local training/evaluation will be performed and allows Flower to call the local training/evaluation through `fit` and `evaluate`. Each instance of `FlowerClient` represents a *single client* in our federated learning system. Federated learning systems have multiple clients (otherwise, there's not much to federate), so each client will be represented by its own instance of `FlowerClient`. If we have, for example, three clients in our workload, then we'd have three instances of `FlowerClient`. Flower calls `FlowerClient.fit` on the respective instance when the server selects a particular client for training (and `FlowerClient.evaluate` for evaluation).

Step-5 Defining the virtual engine class-

In this notebook, we want to simulate a federated learning system with 10 clients on a single machine. This means that the server and all 10 clients will live on a single machine and share resources such as CPU, GPU, and memory. Having 10 clients would mean having 10 instances of `FlowerClient` in memory. Doing this on a single machine can quickly exhaust the available memory resources, even if only a subset of these clients participates in a single round of federated learning.

In addition to the regular capabilities where server and clients run on multiple machines, Flower, therefore, provides special simulation capabilities that create `FlowerClient` instances only when they are actually necessary for training or evaluation. To enable the Flower framework to create clients when necessary,

we need to implement a function called `client_fn` that creates a `FlowerClient` instance on demand. Flower calls `client_fn` whenever it needs an instance of one particular client to call `fit` or `evaluate` (those instances are usually discarded after use, so they should not keep any local state). Clients are identified by a client ID, or short `cid`. The `cid` can be used, to load different local data partitions for different clients

```
def generate_client_fn(trainloaders, valloaders, num_classes):  
  
    def client_fn(cid: str):  
        logging.info(f"Generating client for cid={cid}")  
        return FlowerClient(  
            trainloader=trainloaders[int(cid)],  
            valloader=valloaders[int(cid)],  
            num_classes=num_classes,  
        )  
  
    return client_fn
```

Step-6 Server for combining the model

Once the model for small datasets has been created, then the next step is to combine them. **Federated Evaluation** (or *client-side evaluation*) is more complex, but also more powerful: it doesn't require a centralized dataset and allows us to evaluate models over a larger set of data, which often yields more realistic evaluation results. In fact, many scenarios require us to use **Federated Evaluation** if we want to get representative evaluation results at all. But this power comes at a cost: once we start to evaluate on the client side, we should be aware that our evaluation dataset can change over consecutive rounds of learning if those clients are not always available. Moreover, the dataset held by each client can also change over consecutive rounds. This can lead to evaluation results that are not stable, so even if we would not change the model, we'd see our evaluation results fluctuate over consecutive rounds. To overcome this difficulty, flower framework gives us a method where we can config the clients for proper result.

```
def get_on_fit_config(config: DictConfig):

    def fit_config_fn(server_round: int):
        logging.info(f"Preparing fit config for server round {server_round}")

        return {
            "lr": 0.01,
            "momentum": 0.9,
            "local_epochs": 1,
        }

    return fit_config_fn
```

Flower module also provide the features to aggregate the model based on different parameters. For this we need to define a evaluation class in the server.

```
def get_evaluate_fn(num_classes: int, testloader):

    def evaluate_fn(server_round: int, parameters, config):
        logging.info(f"Evaluating global model at server round {server_round}")

        model = Net(num_classes)
        device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

        logging.info(f"Using device: {device}")

        params_dict = zip(model.state_dict().keys(), parameters)
        state_dict = OrderedDict({k: torch.Tensor(v) for k, v in params_dict})
        model.load_state_dict(state_dict, strict=True)

        loss, accuracy = test(model, testloader, device)

        logging.info(f"Evaluation results - Loss: {loss}, Accuracy: {accuracy}")

        return loss, {"accuracy": accuracy}

    return evaluate_fn
```

Step-7 Server side parameter initialization

The strategy uses for aggregating the models accepts a number of arguments, amongst them the `client_fn` used to create `FlowerClient` instances, the number of clients to simulate `num_clients`, the number of rounds `num_rounds`, and the strategy.

```
# strategy = fl.server.strategy.FedAvg(  
#     fraction_fit=0.0,  
#     min_fit_clients=2,  
#     fraction_evaluate=0.0,  
#     min_evaluate_clients=2,  
#     min_available_clients=10,  
#     on_fit_config_fn=get_on_fit_config({  
#         "lr":0.01,  
#         "momentum":0.9,  
#         "local_epochs":1  
#     })),  
#     evaluate_fn=get_evaluate_fn(10, testloader),  
#     initial_parameters= cfg.initial_params  
  
# )  
strategy= instantiate(cfg.strategy, evaluate_fn=get_evaluate_fn(10, testloader),)  
  
print("Strategy generated successfully")  
print("Starting simulation")  
history = fl.simulation.start_simulation(  
    client_fn=client_fn,  
    num_clients=10,  
    config=fl.server.ServerConfig(num_rounds=5),  
    strategy=strategy,  
    client_resources={'num_cpus':2}  
)
```

Note- Flower module provide method for creating the custom strategy as per need . For more reference visit –

<https://flower.ai/docs/framework/tutorial-series-build-a-strategy-from-scratch-pytorch.html>

3.3. Configuration of Network

Configuration of Organisation Node

NOTE- (The base code of creating the organization Node can be found in the above two repository at path “artifacts/channel/config/configtx.yaml”. However in the repository we have created another file at path “artifacts/channel/configtx.yaml” where we can change the configuration of organization in an easy manner).

```
Organizations:

# SampleOrg defines an MSP using the sampleconfig. It should never be used
# in production but may be used as a template for other definitions
- &OrdererOrg

# DefaultOrg defines the organization which is used in the sampleconfig
# of the fabric.git development environment
Name: OrdererOrg

# ID to load the MSP definition as
ID: OrdererMSP

# MSPDir is the filesystem path which contains the MSP configuration
MSPDir: crypto-config/ordererOrganizations/example.com/msp

# Policies defines the set of policies at this level of the config tree
# For organization policies, their canonical path is usually
# /Channel/<Application|Orderer>/<OrgName>/<PolicyName>
Policies:
  Readers:
    Type: Signature
    Rule: "OR('OrdererMSP.member')"
  Writers:
    Type: Signature
    Rule: "OR('OrdererMSP.member')"
  Admins:
    Type: Signature
    Rule: "OR('OrdererMSP.admin')"

- &Org1

# DefaultOrg defines the organization which is used in the sampleconfig
# of the fabric.git development environment
Name: Org1MSP

# ID to load the MSP definition as
ID: Org1MSP
```

```

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

# Policies defines the set of policies at this level of the config tree
# For organization policies, their canonical path is usually
# /Channel/<Application|Orderer>/<OrgName>/<PolicyName>
Policies:
  Readers:
    Type: Signature
    Rule: "OR('Org1MSP.admin', 'Org1MSP.peer', 'Org1MSP.client')"
  Writers:
    Type: Signature
    Rule: "OR('Org1MSP.admin', 'Org1MSP.client')"
  Admins:
    Type: Signature
    Rule: "OR('Org1MSP.admin')"
  Endorsement:
    Type: Signature
    Rule: "OR('Org1MSP.peer')"

# leave this flag set to true.
AnchorPeers:
  # AnchorPeers defines the location of peers which can be used
  # for cross org gossip communication. Note, this value is only
  # encoded in the genesis block in the Application section context
  - Host: peer0.org1.example.com
    Port: 7051

```

In the code above , we have define the configuration of one of the organisation which can be change for proper use.

For creation of the organisation, run the command “./create-artifacts.sh” in the Window subsystem linux.

Configuration of Peer Node

NOTE- (The base code of creating the Peer Node can be found in the above two repository at path “artifacts/channel/config/core.yaml”).

To change the configuration we have to go to the above path. However to add the peer, we can do it directly in the docker file.

Configuration of Channel

```
Channel: &ChannelDefaults
  # Policies defines the set of policies at this level of the config tree
  # For Channel policies, their canonical path is
  #   /Channel/<PolicyName>
  Policies:
    # Who may invoke the 'Deliver' API
    Readers:
      Type: ImplicitMeta
      Rule: "ANY Readers"
    # Who may invoke the 'Broadcast' API
    Writers:
      Type: ImplicitMeta
      Rule: "ANY Writers"
    # By default, who may modify elements at this config level
    Admins:
      Type: ImplicitMeta
      Rule: "MAJORITY Admins"

  # Capabilities describes the channel level capabilities, see the
  # dedicated Capabilities section elsewhere in this file for a full
  # description
  Capabilities:
    <<: *ChannelCapabilities
```

The above code shows the channel configuration written in configtx.yaml file.

Configuration of Orderers

(The base code of creating the Orderer Node can be found in the above two repository at path “artifacts/channel/config/orderer.yaml”. However in the repository we have created another file at path “artifacts/channel/configtx.yaml” under the Orderer section where we can change the configuration of organization in an easy manner)

Orderer: &OrdererDefaults

Orderer Type: The orderer implementation to start

OrdererType: etcdraft

EtcdRaft:

Consenters:

- Host: orderer.example.com

Port: 7050

ClientTLSCert: crypto-config/ordererOrganizations/example.com/orderers/orderer.example.com/tls/server.crt

ServerTLSCert: crypto-config/ordererOrganizations/example.com/orderers/orderer.example.com/tls/server.crt

Addresses:

- orderer.example.com:7050

Batch Timeout: The amount of time to wait before creating a batch

BatchTimeout: 2s

Batch Size: Controls the number of messages batched into a block

BatchSize:

Max Message Count: The maximum number of messages to permit in a batch

MaxMessageCount: 10

Absolute Max Bytes: The absolute maximum number of bytes allowed for

the serialized messages in a batch.

AbsoluteMaxBytes: 99 MB

Preferred Max Bytes: The preferred maximum number of bytes allowed for

the serialized messages in a batch. A message larger than the preferred

max bytes will result in a batch larger than preferred max bytes.

PreferredMaxBytes: 512 KB

Organizations is the list of orgs which are defined as participants on

the orderer side of the network

Organizations:

Policies defines the set of policies at this level of the config tree

For Orderer policies, their canonical path is

/Channel/Orderer/<PolicyName>

Policies:

Readers:

Type: ImplicitMeta

Rule: "ANY Readers"

Writers:

Type: ImplicitMeta

Rule: "ANY Writers"

Admins:

Type: ImplicitMeta

Rule: "MAJORITY Admins"

BlockValidation specifies what signatures must be included in the block

from the orderer for the peer to validate it.

BlockValidation:

Type: ImplicitMeta

Rule: "ANY Writers"

```

OrdererOrgs:
# -----
# Orderer
# -----
- Name: Orderer
  Domain: example.com
  EnableNodeOUs: true

# -----
# "Specs" - See PeerOrgs below for complete description
# -----
Specs:
- Hostname: orderer
  SANS:
    - "localhost"
    - "127.0.0.1"
- Hostname: orderer2
  SANS:
    - "localhost"
    - "127.0.0.1"
- Hostname: orderer3
  SANS:
    - "localhost"
    - "127.0.0.1"

```

Configuration of Profile

```

Profiles:
  BasicChannel:
    Consortium: SampleConsortium
    <<: *ChannelDefaults
    Application:
      <<: *ApplicationDefaults
      Organizations:
        - *Org1
        - *Org2
      Capabilities:
        <<: *ApplicationCapabilities

  OrdererGenesis:
    <<: *ChannelDefaults
    Capabilities:
      <<: *ChannelCapabilities
    Orderer:
      <<: *OrdererDefaults
      OrdererType: etcdraft
      EtdcRaft:
        Consenters:
          - Host: orderer.example.com
            Port: 7050
            ClientTLS Cert: crypto-config/ordererOrganizations/example.com/orderers/orderer.example.com/tls/server.crt
            ServerTLS Cert: crypto-config/ordererOrganizations/example.com/orderers/orderer.example.com/tls/server.crt
          - Host: orderer2.example.com
            Port: 8050
            ClientTLS Cert: crypto-config/ordererOrganizations/example.com/orderers/orderer2.example.com/tls/server.crt
            ServerTLS Cert: crypto-config/ordererOrganizations/example.com/orderers/orderer2.example.com/tls/server.crt
          - Host: orderer3.example.com
            Port: 9050
            ClientTLS Cert: crypto-config/ordererOrganizations/example.com/orderers/orderer3.example.com/tls/server.crt
            ServerTLS Cert: crypto-config/ordererOrganizations/example.com/orderers/orderer3.example.com/tls/server.crt
          # - Host: orderer4.example.com
          #   Port: 10050

```

```

# - Host: orderer4.example.com
#   Port: 10050
#   ClientTLSCert: crypto-config/ordererOrganizations/example.com/orderers/orderer4.example.com/tls/server.crt
#   ServerTLSCert: crypto-config/ordererOrganizations/example.com/orderers/orderer4.example.com/tls/server.crt
# - Host: orderer5.example.com
#   Port: 11050
#   ClientTLSCert: crypto-config/ordererOrganizations/example.com/orderers/orderer5.example.com/tls/server.crt
#   ServerTLSCert: crypto-config/ordererOrganizations/example.com/orderers/orderer5.example.com/tls/server.crt
Addresses:
- orderer.example.com:7050
- orderer2.example.com:8050
- orderer3.example.com:9050
# - orderer4.example.com:10050
# - orderer5.example.com:11050

Organizations:
- *OrdererOrg
Capabilities:
  <<: *OrdererCapabilities
Consortiums:
  SampleConsortium:
    Organizations:
      - *Org1
      - *Org2

```

Creation of Cryptogenic Material

Once the configuration for organization, peer, orderer has been decided, we should create the cryptogenic material for each channel. For this in the repository there is a file create-artifacts.sh at path “artifacts/channel/create-artifacts.sh”. This file creates the cryptogenic material in a directory “crypto-config”. In the create-artifacts.sh file we should give the name of the channel. Alongside creating cryptogenic material, it also create a genesis block, channel configuration(used during channel creation), MSP anchors for organization etc.

To run the creater-artifacts.sh file, go to Window subsystem linux and run the command ./create-artifacts.sh by going to proper file location is WSL.

```

chmod -R 0755 ./crypto-config
# Delete existing artifacts
rm -rf ./crypto-config
rm genesis.block mychannel.tx
rm -rf ../../channel-artifacts/*

#Generate Crypto artifacts for organizations
cryptogen generate --config=./crypto-config.yaml --output=./crypto-config/

# System channel
SYS_CHANNEL="sys-channel"

# channel name defaults to "mychannel"
CHANNEL_NAME="mychannel"

echo $CHANNEL_NAME

# Generate System Genesis block
configtxgen -profile OrdererGenesis -configPath . -channelID $SYS_CHANNEL -outputBlock ./genesis.block

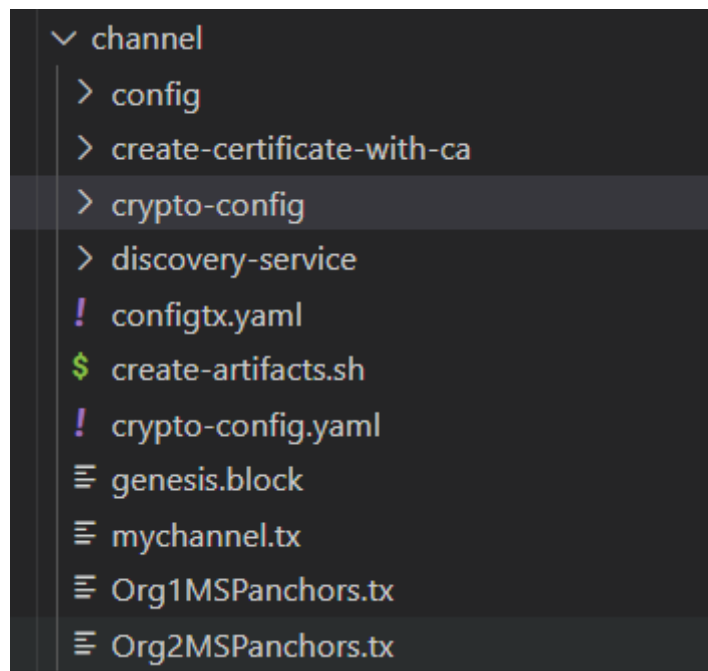
# Generate channel configuration block
configtxgen -profile BasicChannel -configPath . -outputCreateChannelTx ./mychannel.tx -channelID $CHANNEL_NAME

echo "##### Generating anchor peer update for Org1MSP #####"
configtxgen -profile BasicChannel -configPath . -outputAnchorPeersUpdate ./Org1MSPanchors.tx -channelID $CHANNEL_NAME -asOrg Org1MSP

echo "##### Generating anchor peer update for Org2MSP #####"
configtxgen -profile BasicChannel -configPath . -outputAnchorPeersUpdate ./Org2MSPanchors.tx -channelID $CHANNEL_NAME -asOrg Org2MSP

```

After the command is run successfully, the channel directory under artifacts directory will look something like this



3.4. Creation of Container in docker desktop

To run and save the data of various organization, peers, orderers, certificate authority , we have created the containers in the docker desktop.

Docker is used to run various peers, orderers and certificate authority(ca) component in isolated container. This containers are used to store the data and information of the channel and network.

Creation of Certificate authority container

```
ca-org1:
image: hyperledger/fabric-ca
environment:
  - FABRIC_CA_HOME=/etc/hyperledger/fabric-ca-server
  - FABRIC_CA_SERVER_CA_NAME=ca.org1.example.com
  - FABRIC_CA_SERVER_CA_CERTFILE=/etc/hyperledger/fabric-ca-server-config/ca.org1.example.com-cert.pem
  - FABRIC_CA_SERVER_CA_KEYFILE=/etc/hyperledger/fabric-ca-server-config/priv_sk
  - FABRIC_CA_SERVER_TLS_ENABLED=true
  - FABRIC_CA_SERVER_TLS_CERTFILE=/etc/hyperledger/fabric-ca-server-tls/tlsca.org1.example.com-cert.pem
  - FABRIC_CA_SERVER_TLS_KEYFILE=/etc/hyperledger/fabric-ca-server-tls/priv_sk
ports:
  - "7054:7054"
command: sh -c 'fabric-ca-server start -b admin:adminpw -d'
volumes:
  - ./channel/crypto-config/peerOrganizations/org1.example.com/ca:/etc/hyperledger/fabric-ca-server-config
  - ./channel/crypto-config/peerOrganizations/org1.example.com/tlsca:/etc/hyperledger/fabric-ca-server-tls
container_name: ca.org1.example.com
hostname: ca.org1.example.com
networks:
  - test
```

```
ca-org2:
image: hyperledger/fabric-ca
environment:
  - FABRIC_CA_HOME=/etc/hyperledger/fabric-ca-server
  - FABRIC_CA_SERVER_CA_NAME=ca.org2.example.com
  - FABRIC_CA_SERVER_CA_CERTFILE=/etc/hyperledger/fabric-ca-server-config/ca.org2.example.com-cert.pem
  - FABRIC_CA_SERVER_CA_KEYFILE=/etc/hyperledger/fabric-ca-server-config/priv_sk
  - FABRIC_CA_SERVER_TLS_ENABLED=true
  - FABRIC_CA_SERVER_TLS_CERTFILE=/etc/hyperledger/fabric-ca-server-tls/tlsca.org2.example.com-cert.pem
  - FABRIC_CA_SERVER_TLS_KEYFILE=/etc/hyperledger/fabric-ca-server-tls/priv_sk
ports:
  - "8054:7054"
```

```
command: sh -c 'fabric-ca-server start -b admin:adminpw -d'
volumes:
  - ./channel/crypto-config/peerOrganizations/org2.example.com/ca:/etc/hyperledger/fabric-ca-server-config
  - ./channel/crypto-config/peerOrganizations/org2.example.com/tlsca:/etc/hyperledger/fabric-ca-server-tls
container_name: ca.org2.example.com
hostname: ca.org2.example.com
networks:
  - test
```

In the above code , we have created the container for two certificate authority container for the two organization respectively. We have to specify the port number on which they will be running and the path of cryptogenic material for the organization for which they are associated.

Creation of Orderer container

```
orderer.example.com:
  container_name: orderer.example.com
  image: hyperledger/fabric-orderer:2.1
  dns_search: .
  environment:
    - ORDERER_GENERAL_LOGLEVEL=info
    - FABRIC_LOGGING_SPEC=INFO
    - ORDERER_GENERAL_LISTENADDRESS=0.0.0.0
    - ORDERER_GENERAL_GENESISMETHOD=file
    - ORDERER_GENERAL_GENESISFILE=/var/hyperledger/orderer/genesis.block
    - ORDERER_GENERAL_LOCALMSPID=OrdererMSP
    - ORDERER_GENERAL_LOCALMSPDIR=/var/hyperledger/orderer/msp
    - ORDERER_GENERAL_TLS_ENABLED=true
    - ORDERER_GENERAL_TLS_PRIVATEKEY=/var/hyperledger/orderer/tls/server.key
    - ORDERER_GENERAL_TLS_CERTIFICATE=/var/hyperledger/orderer/tls/server.crt
    - ORDERER_GENERAL_TLS_ROOTCAS=[/var/hyperledger/orderer/tls/ca.crt]
    - ORDERER_KAFKA_VERBOSE=true
    - ORDERER_GENERAL_CLUSTER_CLIENTCERTIFICATE=/var/hyperledger/orderer/tls/server.crt
    - ORDERER_GENERAL_CLUSTER_CLIENTPRIVATEKEY=/var/hyperledger/orderer/tls/server.key
    - ORDERER_GENERAL_CLUSTER_ROOTCAS=[/var/hyperledger/orderer/tls/ca.crt]
    - ORDERER_METRICS_PROVIDER=prometheus
    - ORDERER_OPERATIONS_LISTENADDRESS=0.0.0.0:8443
    - ORDERER_GENERAL_LISTENPORT=7050
  working_dir: /opt/gopath/src/github.com/hyperledger/fabric/orderers
  command: orderer
  ports:
    - 7050:7050
    - 8443:8443
  networks:
    - test
  volumes:
    - ./channel/genesis.block:/var/hyperledger/orderer/genesis.block
    - ./channel/crypto-config/ordererOrganizations/example.com/orderers/orderer.example.com/msp:/var/hyperledger/orderer/msp
    - ./channel/crypto-config/ordererOrganizations/example.com/orderers/orderer.example.com/tls:/var/hyperledger/orderer/tls
```

In the above code, I have created the orderer for the channel. We should define the port on which the orderer will be running. Also in the volumes, we have added the path of the genesis block which we have created during the generation of the cryptogenic material. This is necessary to associate each orderer node to a particular channel as the genesis block for each channel is unique. In the project I have associated three orderer to the channel which has almost the same configuration as the above code..

Creation of CouchDB container

```

couchdb0:
  container_name: couchdb0
  image: hyperledger/fabric-couchdb
  environment:
    - COUCHDB_USER=
    - COUCHDB_PASSWORD=
  ports:
    - 5984:5984
  networks:
    - test

couchdb1:
  container_name: couchdb1
  image: hyperledger/fabric-couchdb
  environment:
    - COUCHDB_USER=
    - COUCHDB_PASSWORD=
  ports:
    - 6984:5984
  networks:
    - test

couchdb2:
  container_name: couchdb2
  image: hyperledger/fabric-couchdb
  environment:
    - COUCHDB_USER=
    - COUCHDB_PASSWORD=
  ports:
    - 7984:5984
  networks:
    - test

couchdb3:
  container_name: couchdb3
  image: hyperledger/fabric-couchdb
  environment:
    - COUCHDB_USER=
    - COUCHDB_PASSWORD=
  ports:
    - 8984:5984
  networks:
    - test

```

Couchdb container are created to act as a worldstate database. It is also used to store the chaincode definition. Each couchdb container has a Port Number associated with it on which it will be running.

Creation of Peer Node container

```

peer0.org1.example.com:
  container_name: peer0.org1.example.com
  extends:
    file: base.yaml
    service: peer-base
  environment:
    - FABRIC_LOGGING_SPEC=info
    - ORDERER_GENERAL_LOGLEVEL=info
    - CORE_PEER_LOCALMSPID=Org1MSP

    - CORE_VM_DOCKER_HOSTCONFIG_NETWORKMODE=artifacts_test

    - CORE_PEER_ID=peer0.org1.example.com
    - CORE_PEER_ADDRESS=peer0.org1.example.com:7051
    - CORE_PEER_LISTENADDRESS=0.0.0.0:7051
    - CORE_PEER_CHAINCODEADDRESS=peer0.org1.example.com:7052
    - CORE_PEER_CHAINCODELISTENADDRESS=0.0.0.0:7052
    # Peer used to bootstrap gossip within organisation
    - CORE_PEER_GOSSIP_BOOTSTRAP=peer1.org1.example.com:8051
    # Exposed for discovery Service
    - CORE_PEER_GOSSIP_EXTERNALENDPOINT=peer0.org1.example.com:7051







































































































    # - CORE_OPERATIONS_LISTENADDRESS=0.0.0.0:9440

    - CORE_LEDGER_STATE_STATEDATABASE=CouchDB
    - CORE_LEDGER_STATE_COUCHDBCONFIG_COUCHDBADDRESS=couchdb0:5984
    - CORE_LEDGER_STATE_COUCHDBCONFIG_USERNAME=
    - CORE_LEDGER_STATE_COUCHDBCONFIG_PASSWORD=
    - CORE_METRICS_PROVIDER=prometheus
    - CORE_PEER_TLS_ENABLED=true
    - CORE_PEER_TLS_CERT_FILE=/etc/hyperledger/crypto/peer/tls/server.crt
    - CORE_PEER_TLS_KEY_FILE=/etc/hyperledger/crypto/peer/tls/server.key
    - CORE_PEER_TLS_ROOTCERT_FILE=/etc/hyperledger/crypto/peer/tls/ca.crt
    - CORE_PEER_MSPCONFIGPATH=/etc/hyperledger/crypto/peer/msp
  depends_on:
    - couchdb0
  ports:
    - 7051:7051
  volumes:
    - ./channel/crypto-config/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/msp:/etc/hyperledger/crypto/peer/ms
    - ./channel/crypto-config/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls:/etc/hyperledger/crypto/peer/tl
    - /var/run:/host/var/run/
    - ./channel:/etc/hyperledger/channel/
  networks:
    - test

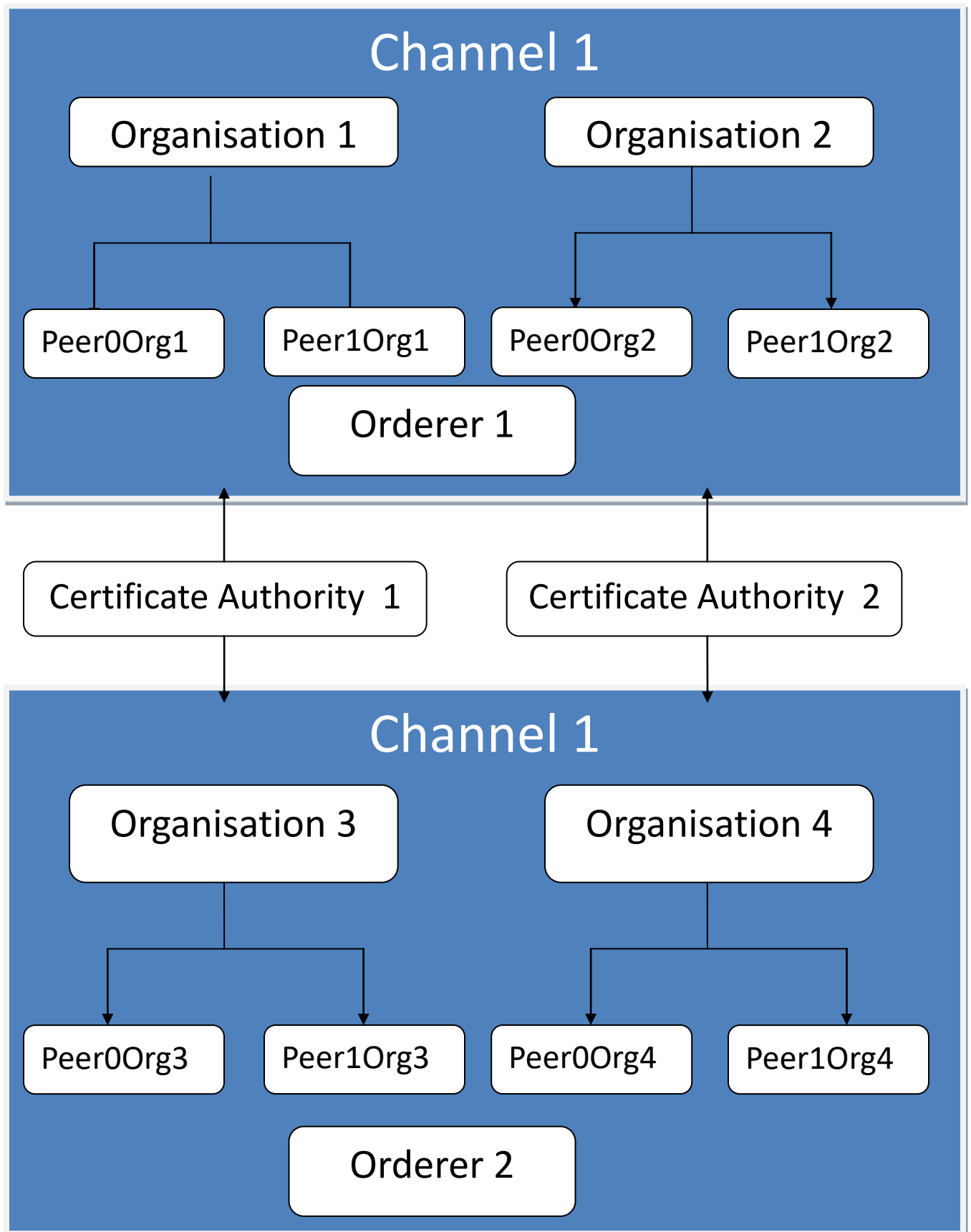
```

The above code is used to create the container for the peer node in the docker. I have only shown the code for one of the containers. However, for other peer nodes, only some configuration changes are needed. Each peer node should have a unique port number on which it is running. Also, each peer node container should define the couchdb container it is associated with. CouchDB is used to store the worldstate data of the peer node. In the environment and volume, we associate various cryptographic material

To create the containers defined in the docker-compose.yaml file, open the Window Subsystem linux, go to file path and run the command docker-compose up -d . Make sure the docker desktop is open while running this command. When the command has successfully executed, go to docker-desktop and something like this will be seen.

<input type="checkbox"/>		dev-peer1.org1.example.com 52961b9a6549 	dev-peer1.org1.example.com-fabcar	Exited (255)		0%	19 days ago			
<input type="checkbox"/>		dev-peer0.org2.example.com ac0b15942eff 	dev-peer0.org2.example.com-fabcar	Exited (255)		0%	19 days ago			
<input type="checkbox"/>		dev-peer1.org2.example.com 93357c08e4b1 	dev-peer1.org2.example.com-fabcar	Exited (255)		0%	19 days ago			
<input type="checkbox"/>		dev-peer0.org1.example.com 5abfccdfa37f 	dev-peer0.org1.example.com-fabcar	Exited (255)		0%	19 days ago			
<input type="checkbox"/>		artifacts		Running (13/1)		21.72%	48 minutes ago			
<input type="checkbox"/>		couchdb3 1d07d17e96e5 	hyperledger/fabric-couchdb	Running	8984:5984 	2.49%	48 minutes ago			
<input type="checkbox"/>		couchdb1 cce8e00bc9de 	hyperledger/fabric-couchdb	Running	6984:5984 	0.73%	48 minutes ago			
<input type="checkbox"/>		couchdb0 3d43a01ec00d 	hyperledger/fabric-couchdb	Running	5984:5984 	2.27%	48 minutes ago			
Showing 18 items										
<input type="checkbox"/>		couchdb2 50bcb04062d3 	hyperledger/fabric-couchdb	Running	7984:5984 	4.57%	48 minutes ago			
<input type="checkbox"/>		ca.org1.example.com e448004214f8 	hyperledger/fabric-ca	Running	7054:7054 	0%	48 minutes ago			
<input type="checkbox"/>		peer1.org1.example.com 3f7abf383086 	hyperledger/fabric-peer:2.1	Running	8051:8051 	3.35%	48 minutes ago			
<input type="checkbox"/>		orderer3.example.com 6c745c058dc5 	hyperledger/fabric-orderer:2.1	Running	8445:8443  Show all ports (2)	0.51%	48 minutes ago			
<input type="checkbox"/>		ca.org2.example.com 351def37e802 	hyperledger/fabric-ca	Running	8054:7054 	0%	48 minutes ago			
<input type="checkbox"/>		orderer2.example.com 2eeff4be0d9e 	hyperledger/fabric-orderer:2.1	Running	8050:8050  Show all ports (2)	0.22%	48 minutes ago			
<input type="checkbox"/>		peer1.org2.example.com 4c119c31286d 	hyperledger/fabric-peer:2.1	Running	10051:10051 	3.19%	48 minutes ago			
<input type="checkbox"/>		orderer.example.com b196ae9b2def 	hyperledger/fabric-orderer:2.1	Running	7050:7050  Show all ports (2)	0.31%	48 minutes ago			
Showing 18 items										
<input type="checkbox"/>		peer0.org1.example.com 358d6ed4dd64 	hyperledger/fabric-peer:2.1	Running	7051:7051 	2.96%	48 minutes ago			
<input type="checkbox"/>		peer0.org2.example.com bf12210ee980 	hyperledger/fabric-peer:2.1	Running	9051:9051 	3.28%	48 minutes ago			

Docker container Architecture



3.5. Creating Channel

```
export CORE_PEER_TLS_ENABLED=true
export ORDERER_CA=${PWD}/artifacts/channel/crypto-config/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacert.pem
export PEER0_ORG1_CA=${PWD}/artifacts/channel/crypto-config/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/ca.pem
export PEER0_ORG2_CA=${PWD}/artifacts/channel/crypto-config/peerOrganizations/org2.example.com/peers/peer0.org2.example.com/tls/ca.pem
export FABRIC_CFG_PATH=${PWD}/artifacts/channel/config/

export CHANNEL_NAME=mychannel

# setGlobalsForOrderer(){
#   export CORE_PEER_LOCALMSPID="OrdererMSP"
#   export CORE_PEER_TLS_ROOTCERT_FILE=${PWD}/artifacts/channel/crypto-config/ordererOrganizations/example.com/orderers/orderer.example.com/tls/ca.pem
#   export CORE_PEER_MSPCONFIGPATH=${PWD}/artifacts/channel/crypto-config/ordererOrganizations/example.com/users/Admin@example.com/msp
# }

setGlobalsForPeer0Org1(){
  export CORE_PEER_LOCALMSPID="Org1MSP"
  export CORE_PEER_TLS_ROOTCERT_FILE=${PEER0_ORG1_CA}
  export CORE_PEER_MSPCONFIGPATH=${PWD}/artifacts/channel/crypto-config/peerOrganizations/org1.example.com/users/Admin@org1.example.com/msp
  export CORE_PEER_ADDRESS=localhost:7051
}

setGlobalsForPeer1Org1(){
  export CORE_PEER_LOCALMSPID="Org1MSP"
  export CORE_PEER_TLS_ROOTCERT_FILE=${PEER0_ORG1_CA}
  export CORE_PEER_MSPCONFIGPATH=${PWD}/artifacts/channel/crypto-config/peerOrganizations/org1.example.com/users/Admin@org1.example.com/msp
  export CORE_PEER_ADDRESS=localhost:8051
}

setGlobalsForPeer0Org2(){
  export CORE_PEER_LOCALMSPID="Org2MSP"
  export CORE_PEER_TLS_ROOTCERT_FILE=${PEER0_ORG2_CA}
  export CORE_PEER_MSPCONFIGPATH=${PWD}/artifacts/channel/crypto-config/peerOrganizations/org2.example.com/users/Admin@org2.example.com/msp
  export CORE_PEER_ADDRESS=localhost:9051
}
```

```
setGlobalsForPeer1Org2(){
  export CORE_PEER_LOCALMSPID="Org2MSP"
  export CORE_PEER_TLS_ROOTCERT_FILE=${PEER0_ORG2_CA}
  export CORE_PEER_MSPCONFIGPATH=${PWD}/artifacts/channel/crypto-config/peerOrganizations/org2.example.com/users/Admin@org2.example.com/msp
  export CORE_PEER_ADDRESS=localhost:10051
}

createChannel(){
  rm -rf ./channel-artifacts/*
  setGlobalsForPeer0Org1

  peer channel create -o localhost:7050 -c $CHANNEL_NAME \
    --ordererTLSHostnameOverride orderer.example.com \
    -f ./artifacts/channel/${CHANNEL_NAME}.tx --outputBlock ./channel-artifacts/${CHANNEL_NAME}.block \
    --tls $CORE_PEER_TLS_ENABLED --cafile $ORDERER_CA
}

removeOldCrypto(){
  rm -rf ./api-1.4/crypto/*
  rm -rf ./api-1.4/fabric-client-kv-org1/*
  rm -rf ./api-2.0/org1-wallet/*
  rm -rf ./api-2.0/org2-wallet/*
}

joinChannel(){
  setGlobalsForPeer0Org1
  peer channel join -b ./channel-artifacts/${CHANNEL_NAME}.block

  setGlobalsForPeer1Org1
  peer channel join -b ./channel-artifacts/${CHANNEL_NAME}.block

  setGlobalsForPeer0Org2
  peer channel join -b ./channel-artifacts/${CHANNEL_NAME}.block
}
```

```

    setGlobalsForPeer0Org2
    peer channel join -b ./channel-artifacts/${CHANNEL_NAME}.block

    setGlobalsForPeer1Org2
    peer channel join -b ./channel-artifacts/${CHANNEL_NAME}.block
}

updateAnchorPeers(){
    setGlobalsForPeer0Org1
    peer channel update -o localhost:7050 --ordererTLSHostnameOverride orderer.example.com -c $CHANNEL_NAME -f ./artifacts/channel/

    setGlobalsForPeer0Org2
    peer channel update -o localhost:7050 --ordererTLSHostnameOverride orderer.example.com -c $CHANNEL_NAME -f ./artifacts/channel/
}

removeOldCrypto

createChannel
joinChannel
updateAnchorPeers

```

Once the container has been created, the next steps is to create the channel. For creating the channel, you should have the `geneseisblock.tx` and `channelname.tx` file created during the generation of cryptogenic material.

Step-1 The channel contains orderer node and organisation. This configuration is defined in the consortium at path “`artifacts/channel/configtx.yaml`”. We need to bring the required certificate for the orderer node and anchor peer node of the organisations in the channel file. This certificate are generated by the certificate authority and MSP.

Step-2 After this we need to bring the certificate of all the peers that are there in the organisations. For this we have defined a function `setGlobalsForPeer1Org1`.

Step-3 After we have get all the certificate for each of the we write a function to create the channel. One of the main thing in this function is that we have to choose a peer that will be working as the anchor of the channel. For the above code I have choosen Peer0Org1 as the anchor of the channel. We also have to tell the orderer for that channel.

Step-4 After the creation of channel, we have to make ensure that other peer also join the channel. For this we have written the function `JoinChannel`.

Step-5 After this, we need to set the anchor node for each of the organisation in the channel. For this we have written the function `updateChannel`.

3.6. Deploying Chain Code

Step-1

```
export CORE_PEER_TLS_ENABLED=true
export ORDERER_CA=${PWD}/artifacts/channel/crypto-config/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacert.pem
export PEER0_ORG1_CA=${PWD}/artifacts/channel/crypto-config/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/ca.crt
export PEER1_ORG1_CA=${PWD}/artifacts/channel/crypto-config/peerOrganizations/org1.example.com/peers/peer1.org1.example.com/tls/ca.crt
export PEER0_ORG2_CA=${PWD}/artifacts/channel/crypto-config/peerOrganizations/org2.example.com/peers/peer0.org2.example.com/tls/ca.crt
export PEER1_ORG2_CA=${PWD}/artifacts/channel/crypto-config/peerOrganizations/org2.example.com/peers/peer1.org2.example.com/tls/ca.crt
export FABRIC_CFG_PATH=${PWD}/artifacts/channel/config/

export PRIVATE_DATA_CONFIG=${PWD}/artifacts/private-data/collections_config.json

export CHANNEL_NAME=mychannel

setGlobalsForOrderer() {
    export CORE_PEER_LOCALMSPID="OrdererMSP"
    export CORE_PEER_TLS_ROOTCERT_FILE=${PWD}/artifacts/channel/crypto-config/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacert.pem
    export CORE_PEER_MSPCONFIGPATH=${PWD}/artifacts/channel/crypto-config/ordererOrganizations/example.com/users/Admin@example.com/msp
}

setGlobalsForPeer0Org1() {
    export CORE_PEER_LOCALMSPID="Org1MSP"
    export CORE_PEER_TLS_ROOTCERT_FILE=${PWD}/artifacts/channel/crypto-config/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/ca.crt
    export CORE_PEER_MSPCONFIGPATH=${PWD}/artifacts/channel/crypto-config/peerOrganizations/org1.example.com/users/Admin@org1.example.com/msp
    # export CORE_PEER_MSPCONFIGPATH=${PWD}/artifacts/channel/crypto-config/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/ca.crt
    export CORE_PEER_ADDRESS=localhost:7051
}

setGlobalsForPeer1Org1() {
    export CORE_PEER_LOCALMSPID="Org1MSP"
    export CORE_PEER_TLS_ROOTCERT_FILE=${PWD}/artifacts/channel/crypto-config/peerOrganizations/org1.example.com/peers/peer1.org1.example.com/tls/ca.crt
    export CORE_PEER_MSPCONFIGPATH=${PWD}/artifacts/channel/crypto-config/peerOrganizations/org1.example.com/users/Admin@org1.example.com/msp
    # export CORE_PEER_MSPCONFIGPATH=${PWD}/artifacts/channel/crypto-config/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/ca.crt
    export CORE_PEER_ADDRESS=localhost:8051
}

setGlobalsForPeer0Org2() {
    export CORE_PEER_LOCALMSPID="Org2MSP"
```

```
    export CORE_PEER_TLS_ROOTCERT_FILE=${PWD}/artifacts/channel/crypto-config/peerOrganizations/org2.example.com/peers/peer0.org2.example.com/tls/ca.crt
    export CORE_PEER_MSPCONFIGPATH=${PWD}/artifacts/channel/crypto-config/peerOrganizations/org2.example.com/users/Admin@org2.example.com/msp
    export CORE_PEER_ADDRESS=localhost:9051
}

setGlobalsForPeer1Org2() {
    export CORE_PEER_LOCALMSPID="Org2MSP"
    export CORE_PEER_TLS_ROOTCERT_FILE=${PWD}/artifacts/channel/crypto-config/peerOrganizations/org2.example.com/peers/peer1.org2.example.com/tls/ca.crt
    export CORE_PEER_MSPCONFIGPATH=${PWD}/artifacts/channel/crypto-config/peerOrganizations/org2.example.com/users/Admin@org2.example.com/msp
    export CORE_PEER_ADDRESS=localhost:10051
}
```

The above code is used to bring necessary certificate and other documents for the deployment.

Step-2

```
presetup() {  
    echo Vendoring Go dependencies ...  
    pushd ./artifacts/src/github.com/fabcar2/go  
    GO111MODULE=on go mod vendor  
    popd  
    echo Finished vendoring Go dependencies  
}  
# presetup
```

The shell function `presetup` is designed to prepare the Go dependencies. It then uses the `pushd` command to change the current directory to `./artifacts/src/github.com/fabcar2/go`, where the Go source code for the chaincode is located. The `pushd` command not only changes the directory but also saves the current directory on a stack, allowing for easy navigation back to the original directory later. Within the specified directory, the function sets the environment variable `GO111MODULE` to `on`, enabling module support in Go. It then executes the `go mod vendor` command. This command creates a vendor directory within the project and copies all the necessary dependencies into it. Vendoring is a process in Go that ensures all dependencies are included within the project itself, making it self-contained and ensuring that the specific versions of dependencies are used consistently. After completing the vendoring process, the function uses the `popd` command to return to the original directory that was saved on the stack.

Step-3

```
CHANNEL_NAME="mychannel"  
CC_RUNTIME_LANGUAGE="golang"  
VERSION="1"  
CC_SRC_PATH="./artifacts/src/github.com/fabcar2/go"  
CC_NAME="fabcar2"  
  
packageChaincode() {  
    rm -rf ${CC_NAME}.tar.gz  
    setGlobalsForPeer0Org1  
    peer lifecycle chaincode package ${CC_NAME}.tar.gz \  
        --path ${CC_SRC_PATH} --lang ${CC_RUNTIME_LANGUAGE} \  
        --label ${CC_NAME}_${VERSION}  
    echo "===== Chaincode is packaged on peer0.org1 ===== "  
}  
# packageChaincode
```

The `packageChaincode` function is used to package a Hyperledger Fabric chaincode into a deployable artifact. The function sets the environment variables necessary for interacting with the peer node of the first organization (Org1) by calling `setGlobalsForPeer0Org1`, which configures the necessary credentials and network details for communication with the peer. With the environment configured, the function then uses the `peer lifecycle chaincode package` command to create the chaincode package. This command packages the chaincode located in `${CC_SRC_PATH}`—which is set to `./artifacts/src/github.com/fabcar2/go`—and specifies the programming language for the chaincode using the `--lang` option with the value `"golang"`. It also assigns a label to the package using the `--label` option, which combines the chaincode name (`${CC_NAME}`) and version (`${VERSION}`, set to `"1"`). The resulting package is saved as `${CC_NAME}.tar.gz`, which will be used for deploying and installing the chaincode onto the network.

Step-4

```
installChaincode() {  
    setGlobalsForPeer0Org1  
    peer lifecycle chaincode install ${CC_NAME}.tar.gz  
    echo "===== Chaincode is installed on peer0.org1 ===== "  
  
    setGlobalsForPeer1Org1  
    peer lifecycle chaincode install ${CC_NAME}.tar.gz  
    echo "===== Chaincode is installed on peer1.org1 ===== "  
  
    setGlobalsForPeer0Org2  
    peer lifecycle chaincode install ${CC_NAME}.tar.gz  
    echo "===== Chaincode is installed on peer0.org2 ===== "  
  
    setGlobalsForPeer1Org2  
    peer lifecycle chaincode install ${CC_NAME}.tar.gz  
    echo "===== Chaincode is installed on peer1.org2 ===== "  
}  
  
# installChaincode
```

The `installChaincode` function is responsible for installing a packaged Hyperledger Fabric chaincode onto multiple peer nodes across different organizations within the network. The function begins by setting the necessary environment variables for interacting with the peer node of Org1's first peer, `peer0.org1`, by calling `setGlobalsForPeer0Org1`. With the environment configured, the function then uses the `peer lifecycle chaincode install` command

to install the chaincode package, `${CC_NAME}.tar.gz`, onto this peer node. The function then proceeds to install the same chaincode package on `peer1.org1`, Org1's second peer node. It updates the environment variables using `setGlobalsForPeer1Org1` before performing the installation and prints a confirmation message. It does the same for organization 2 nodes.

Step-5

```
queryInstalled() {
    setGlobalsForPeer0Org1
    peer lifecycle chaincode queryinstalled >&log.txt
    cat log.txt
    PACKAGE_ID=$(sed -n "/${CC_NAME}_${VERSION}/s/^Package ID: //; s/, Label:.*$/; p;" log.txt)
    echo PackageID is ${PACKAGE_ID}
    echo "===== Query installed successful on peer0.org1 on channel ===== "
}

# queryInstalled
```

The `queryInstalled` function is designed to query the installed chaincodes on a specific peer node and extract the package ID of the installed chaincode. The function starts by setting the appropriate environment variables for interacting with the peer node `peer0.org1` through the `setGlobalsForPeer0Org1` command. Next, the function uses the `peer lifecycle chaincode queryinstalled` command to retrieve a list of all installed chaincodes on `peer0.org1`. The function then uses the `cat` command to display the contents of `log.txt`, allowing the user to view the raw output from the `queryinstalled` command. To extract the package ID of the installed chaincode, the function utilizes the `sed` command. Specifically, it searches for a line in `log.txt` that contains the pattern `${CC_NAME}_${VERSION}`, which corresponds to the label of the chaincode package. It then extracts the portion of the line following "Package ID: " and before ", Label:.*\$" (which matches the label information), capturing the package ID. This extracted package ID is stored in the `PACKAGE_ID` variable.

Step-6


```

approveForMyOrg1() {
    setGlobalsForPeer0Org1
    # set -x
    peer lifecycle chaincode approveformyorg -o localhost:7050 \
        --ordererTLSHostnameOverride orderer.example.com --tls \
        --collections-config $PRIVATE_DATA_CONFIG \
        --cafile $ORDERER_CA --channelID $CHANNEL_NAME --name ${CC_NAME} --version ${VERSION} \
        --init-required --package-id ${PACKAGE_ID} \
        --sequence ${VERSION}

    # set +x

    echo "===== chaincode approved from org 1 ===== "
}

```

The `approveForMyOrg1` function is used to approve a chaincode for deployment on behalf of Org1. It first sets up the environment for interacting with Org1's peer node by calling `setGlobalsForPeer0Org1`. The function then uses the peer lifecycle chaincode `approveformyorg` command to send an approval request to the orderer, specifying details such as the orderer's address, TLS settings, the channel name, the chaincode name, version, and package ID. This approval includes configuration for private data collections if applicable. After the approval command is executed, the function prints a confirmation message indicating that the chaincode has been successfully approved by Org1.

Step-7

```

checkCommitReadiness() {
    setGlobalsForPeer0Org1
    peer lifecycle chaincode checkcommitreadiness \
        --collections-config $PRIVATE_DATA_CONFIG \
        --channelID $CHANNEL_NAME --name ${CC_NAME} --version ${VERSION} \
        --sequence ${VERSION} --output json --init-required
    echo "===== checking commit readiness from org 1 ===== "
}

```

The `checkCommitReadiness` function is used to verify whether a chaincode is ready to be committed to the channel. It first sets the environment for Org1's peer node using `setGlobalsForPeer0Org1`. It then executes the peer lifecycle chaincode `checkcommitreadiness` command, which checks the readiness of the chaincode for commit by specifying the channel, chaincode name, version, sequence number, and any private data configurations. The output is formatted

as JSON. After running the command, the function prints a message confirming that the readiness check has been performed for Org1.

The step-6 and Step-7 are done again for the remaining organization.

Step-8

```
commitChaincodeDefinition() {  
  setGlobalsForPeer0Org1  
  peer lifecycle chaincode commit -o localhost:7050 --ordererTLSHostnameOverride orderer.example.com \  
    --tls $CORE_PEER_TLS_ENABLED --cafile $ORDERER_CA \  
    --channelID $CHANNEL_NAME --name ${CC_NAME} \  
    --collections-config $PRIVATE_DATA_CONFIG \  
    --peerAddresses localhost:7051 --tlsRootCertFiles $PEER0_ORG1_CA \  
    --peerAddresses localhost:8051 --tlsRootCertFiles $PEER1_ORG1_CA \  
    --peerAddresses localhost:9051 --tlsRootCertFiles $PEER0_ORG2_CA \  
    --peerAddresses localhost:10051 --tlsRootCertFiles $PEER1_ORG2_CA \  
    --version ${VERSION} --sequence ${VERSION} --init-required  
}
```

The `commitChaincodeDefinition` function is used to commit the chaincode definition to the channel. It sets up the environment for Org1's peer node and then executes the `peer lifecycle chaincode commit` command. This command specifies details like the orderer's address, TLS settings, the channel name, chaincode name, version, and sequence. It also lists the peer nodes from Org1 and Org2 that will endorse the chaincode, along with their TLS certificates. The command commits the chaincode to the channel with the given configuration.

Step-9

```
queryCommitted() {  
  setGlobalsForPeer0Org1  
  peer lifecycle chaincode querycommitted --channelID $CHANNEL_NAME --name ${CC_NAME}  
}
```

The `queryCommitted` function is used to check which chaincode definitions have been committed to the specified channel. It sets up the environment for Org1's peer node and then executes the `peer lifecycle chaincode querycommitted` command to retrieve and display the chaincode details for the specified channel and chaincode name.

Step-10

```
chaincodeInvokeInit() {
    setGlobalsForPeer0Org1
    peer chaincode invoke -o localhost:7050 \
        --ordererTLSHostnameOverride orderer.example.com \
        --tls $CORE_PEER_TLS_ENABLED --cafile $ORDERER_CA \
        -C $CHANNEL_NAME -n ${CC_NAME} \
        --peerAddresses localhost:7051 --tlsRootCertFiles $PEER0_ORG1_CA \
        --peerAddresses localhost:8051 --tlsRootCertFiles $PEER1_ORG1_CA \
        --peerAddresses localhost:9051 --tlsRootCertFiles $PEER0_ORG2_CA \
        --peerAddresses localhost:10051 --tlsRootCertFiles $PEER1_ORG2_CA \
        --isInit -c '{"Args":[]}'
}
```

The `chaincodeInvokeInit` function is used to initialize a chaincode on a Hyperledger Fabric network by invoking its `init` function. The function starts by configuring the environment for Org1's `peer0` node through `setGlobalsForPeer0Org1`, ensuring that the subsequent commands are executed with the correct peer settings. It then executes the `peer chaincode invoke` command to send a transaction proposal to the orderer. This command specifies the orderer's address and TLS settings, including the CA file for certificate validation. It also sets the channel where the chaincode is deployed and identifies the chaincode to be initialized. The `--isInit` flag indicates that this invocation is for initialization purposes, and the `-c '{"Args":[]}'` parameter passes an empty argument list to the chaincode's initialization function. Additionally, the command includes the addresses and TLS certificates for all peer nodes that will be involved in endorsing the transaction, ensuring secure and validated communication throughout the process.

3.7. Smart Contract

A smart contract defines the rules between different organizations in executable code. Applications invoke a smart contract to generate transactions that are recorded on the ledger.

In the code given below, I have defined a structure `DataInfo` that have 3 parameters. The parameter `ClientId` is used to define the id of the dataset divided

in the python dataset file. The TrainCount and ValCount defines the number of data for training set and validation set respectively. The init go function is executed as soon as the package is imported and used when you need your application to initialize in a specific state.

```
type SmartContract struct{}

type DataInfo struct {
    ClientID    string `json:"clientID"`
    TrainCount  int    `json:"trainCount"`
    ValCount    int    `json:"valCount"`
    APIResponse string `json:"apiResponse,omitempty"`
}

func (s *SmartContract) Init(APIStub shim.ChaincodeStubInterface) peer.Response {
    fmt.Println("Init function called")
    return shim.Success(nil)
}
```

Whenever any client want to interact with smartcontract, they need to call the invoke. They have to pass the name of the function they want to call and the arguments they want to send with it. The invoke function decide which function to call.

```
func (s *SmartContract) Invoke(APIStub shim.ChaincodeStubInterface) peer.Response {
    fmt.Println("Invoke function called")
    function, args := APIStub.GetFunctionAndParameters()

    switch function {
    case "StoreDatasetInfo":
        fmt.Println("StoreDatasetInfo function called")
        return s.StoreDatasetInfo(APIStub, args)
    case "RetrieveDatasetInfo":
        fmt.Println("RetrieveDatasetInfo function called")
        return s.RetrieveDatasetInfo(APIStub, args)
    case "GetAllData":
        fmt.Println("GetAllData function called")
        return s.GetAllData(APIStub)
    default:
        fmt.Println("Invalid function name")
        return shim.Error("Invalid function name")
    }
}
```

StoreDatesetInfo is used to store the dataset info in the ledger. It forms a data Info variable, marshal the data and then call PutState to put the data into ledger.

```

func (s *SmartContract) StoreDatasetInfo(APIStub shim.ChaincodeStubInterface, args []string) peer.Response {
    fmt.Println("StoreDatasetInfo function called")
    if len(args) != 3 {
        return shim.Error("Incorrect number of arguments. Expecting 3")
    }

    clientID := args[0]
    trainCount, err := strconv.Atoi(args[1])
    if err != nil {
        return shim.Error("Train count must be a numeric value")
    }

    valCount, err := strconv.Atoi(args[2])
    if err != nil {
        return shim.Error("Validation count must be a numeric value")
    }

    dataInfo := DataInfo{
        ClientID:  clientID,
        TrainCount: trainCount,
        ValCount:  valCount,
    }

    dataInfoBytes, err := json.Marshal(dataInfo)
    if err != nil {
        return shim.Error(fmt.Sprintf("Failed to marshal dataset info: %s", err))
    }

    err = APIStub.PutState(clientID, dataInfoBytes)
    if err != nil {
        return shim.Error(fmt.Sprintf("Failed to store dataset info: %s", err))
    }

    fmt.Println("StoreDatasetInfo operation successful")
    return shim.Success([]byte("Operation Successful"))
}

```

RetrieveDatasetInfo function is used to retrieve the information stored by a particular client in the ledger. It takes the clientID as a parameter and APIStub.GetState to get the data from ledger.

```

func (s *SmartContract) RetrieveDatasetInfo(APIStub shim.ChaincodeStubInterface, args []string) peer.Response {
    fmt.Println("RetrieveDatasetInfo function called")
    if len(args) != 1 {
        return shim.Error("Incorrect number of arguments. Expecting 1")
    }

    clientID := args[0]
    dataInfoBytes, err := APIStub.GetState(clientID)
    if err != nil {
        return shim.Error(fmt.Sprintf("Failed to read from world state: %s", err))
    }
    if dataInfoBytes == nil {
        return shim.Error(fmt.Sprintf("Dataset info for clientID %s does not exist", clientID))
    }

    fmt.Println("RetrieveDatasetInfo operation successful")
    return shim.Success(dataInfoBytes)
}

```

GetAllData function is used to get all the data of a particular channel. In HyperledgerFabric we can retrieve the data only of the channel for which we are a member. It retrieves all the data using resultsIterator and store the data in clientIDs list.

```

func (s *SmartContract) GetAllData(APIStub shim.ChaincodeStubInterface) peer.Response {
    fmt.Println("GetAllData function called")
    startKey := ""
    endKey := ""
    resultsIterator, err := APIStub.GetStateByRange(startKey, endKey)
    if err != nil {
        return shim.Error(fmt.Sprintf("Failed to get state by range: %s", err))
    }
    defer resultsIterator.Close()
    var results []DataInfo
    var clientIDs []string
    for resultsIterator.HasNext() {
        queryResponse, err := resultsIterator.Next()
        if err != nil {
            return shim.Error(fmt.Sprintf("Error iterating over query results: %s", err))
        }
        var dataInfo DataInfo
        err = json.Unmarshal(queryResponse.Value, &dataInfo)
        if err != nil {
            return shim.Error(fmt.Sprintf("Failed to unmarshal data: %s", err))
        }
        results = append(results, dataInfo)
        clientIDs = append(clientIDs, dataInfo.ClientID)
    }
    clientIDsBytes, err := json.Marshal(clientIDs) // Marshal clientIDs slice into JSON bytes
    if err != nil {
        return shim.Error(fmt.Sprintf("Failed to marshal client IDs: %s", err))
    }

    fmt.Println("GetAllData operation successful")
    return shim.Success(clientIDsBytes)
}

```

```

func main() {
    err := shim.Start(new(SmartContract))
    if err != nil {
        fmt.Printf("Error starting chaincode: %s", err)
    }
}

```

3.8. API

Get Registered user

In the `getRegisteredUser` function we take the name of the client and the name of organisation in which it want to register itself. The initial line of code is used to fetch the necessary variables from other files. We then check whether the user is already registered or not. Then we find the admin fo the given organisation. Then the certificate authority register the user and generate the necessary documents.

The X509 certificate of the user in the wallet and the response is sent back. This api is called using the postman making a post request on <http://localhost:4000/users>

```
app.post('/users', async function (req, res) {
  var username = req.body.username;
  var orgName = req.body.orgName;
  logger.debug('End point : /users');
  logger.debug('User name : ' + username);
  logger.debug('Org name : ' + orgName);
  if (!username) {
    res.json(getErrorMessage('\username\'));
    return;
  }
  if (!orgName) {
    res.json(getErrorMessage('\orgName\'));
    return;
  }

  var token = jwt.sign({
    exp: Math.floor(Date.now() / 1000) + parseInt(constants.jwt_expiretime),
    username: username,
    orgName: orgName
  }, app.get('secret'));

  let response = await getRegisteredUser(username, orgName, true);

  logger.debug('-- returned from registering the username %s for organization %s', username, orgName);
  if (response && typeof response !== 'string') {
    logger.debug('Successfully registered the username %s for organization %s', username, orgName);
    response.token = token;
    res.json(response);
  } else {
    logger.debug('Failed to register the username %s for organization %s with::%s', username, orgName, response);
    res.json({ success: false, message: response });
  }
});
```

```
const getRegisteredUser = async (username, userOrg, isJson) => {
  let ccp = await getCCP(userOrg)

  const caURL = await getCaUrl(userOrg, ccp)
  const ca = new FabricCAServices(caURL);

  const walletPath = await getWalletPath(userOrg)
  const wallet = await Wallets.newFileSystemWallet(walletPath);
  console.log(`Wallet path: ${walletPath}`);

  const userIdentity = await wallet.get(username);
  if (userIdentity) {
    console.log(`An identity for the user ${username} already exists in the wallet`);
    var response = {
      success: true,
      message: username + ' enrolled Successfully',
    };
    return response
  }

  let adminIdentity = await wallet.get('admin');
  if (!adminIdentity) {
    console.log('An identity for the admin user "admin" does not exist in the wallet');
    await enrollAdmin(userOrg, ccp);
    adminIdentity = await wallet.get('admin');
    console.log("Admin Enrolled Successfully")
  }

  const provider = wallet.getProviderRegistry().getProvider(adminIdentity.type);
  const adminUser = await provider.getUserContext(adminIdentity, 'admin');
  let secret;
```

```

let secret;
try {
  secret = await ca.register({ affiliation: await getAffiliation(userOrg), enrollmentID: username, role: 'client' }, adminUser)
} catch (error) {
  return error.message
}

const enrollment = await ca.enroll({ enrollmentID: username, enrollmentSecret: secret });

let x509Identity;
if (userOrg == "Org1") {
  x509Identity = {
    credentials: {
      certificate: enrollment.certificate,
      privateKey: enrollment.key.toBytes(),
    },
    mspId: 'Org1MSP',
    type: 'X.509',
  };
} else if (userOrg == "Org2") {
  x509Identity = {
    credentials: {
      certificate: enrollment.certificate,
      privateKey: enrollment.key.toBytes(),
    },
    mspId: 'Org2MSP',
    type: 'X.509',
  };
}

```

```

await wallet.put(username, x509Identity);
console.log(`Successfully registered and enrolled admin user ${username} and imported it into the wallet`);

var response = {
  success: true,
  message: username + ' enrolled Successfully',
};
return response
}

```

Invoke Transaction for SetDataInfo.

Invoke transaction for SetDataInfo is another api call which is used to invoke the smart contract/chaincode function. It takes various parameters like channelname, codename, fcn(chaincode function to call), arguments, etc. It generated the necessary documents to register the user. Then it check whether user exists or not. If user exists, then it generate the connectOptions and other options to connect with chaincode. It then call the Submit transaction and finally return the response back.


```

app.post('/channels/:channelName/chaincodes/:chaincodeName', async function(req, res){
  try{
    var chaincodeName= req.params.chaincodeName;
    var channelName= req.params.channelName;
    var username= req.body.username
    var orgname= req.body.orgname
    var fcn= req.body.fcn
    var args= req.body.args
    // var peers= req.body.peers;

    if (!chaincodeName) {
      res.json(getErrorMessage('\chaincodeName\'));
      return;
    }
    if (!channelName) {
      res.json(getErrorMessage('\channelName\'));
      return;
    }
    if (!fcn) {
      res.json(getErrorMessage('\fcn\'));
      return;
    }
    if (!args) {
      res.json(getErrorMessage('\args\'));
      return;
    }

    let message = await invoke.invokeTransaction(channelName, chaincodeName, fcn, args, username, orgname);
    console.log("message is " + message)

    const response_payload = {
      result: message,
      error: null,
      errorData: null
    }
  }

```

```

    console.log(response_payload)
    res.send(response_payload);
  }
  catch(err){
    const response_payload = {
      result: null,
      error: err.name,
      errorData: err.message
    }
    res.send(response_payload)
  }
})

```

```

const invokeTransaction = async (channelName, chaincodeName, fcn, args, username, org_name, transientData) => {
  try{
    const ccp = await helper.getCCP(org_name)
    const walletPath = await helper.getWalletPath(org_name)
    const wallet = await Wallets.newFileSystemWallet(walletPath);
    console.log(`Wallet path: ${walletPath}`);

    let identity = await wallet.get(username);
    if (!identity) {
      console.log(`An identity for the user ${username} does not exist in the wallet, so registering user`);
      await helper.getRegisteredUser(username, org_name, true)
      identity = await wallet.get(username);
      console.log('Run the registerUser.js application before retrying');
      return;
    }

    const connectOptions = {
      wallet, identity: username, discovery: { enabled: true, asLocalhost: true },
      eventHandlerOptions: {
        commitTimeout: 100,
        strategy: DefaultEventHandlerStrategies.NETWORK_SCOPE_ALLFORTX
      }
    }

    const gateway = new Gateway();
    await gateway.connect(ccp, connectOptions);

    const network = await gateway.getNetwork(channelName);
    const contract = network.getContract(chaincodeName);

```

```

    let result
    let message;

    result = await contract.submitTransaction(fcn, args[0], args[1], args[2], args[3], args[4]);
    message = `Successfully added the images asset with key ${args[0]}`

    result = JSON.parse(result.toString());

    let response = {
      message: message,
      result
    }

    return response;
  } catch(err){
    console.log(`Getting error: ${err}`)
  }
}

```

Invoke Transaction for GetAllData

Invoke Transaction for GetAllData is api call which is used to retrieve the information and simultaneously train model on the data. The api call take the parameters especiall channelName and call the qssc function. The qssc function first validate the user , then validate whether user is the member of that channel and that call the evaluate transaction to retrieve the channel data. Once the data is retrieved, it send the data to the federated learning model. Since the user can only retrieve the data of channel for which it is user, it ensures the privacy of data in the training model.

```

app.get('/qsc/channel/:channelName/chaincodes/:chaincodeName', async function (req, res) {
  try {
    logger.debug('===== QUERY BY CHAINCODE =====');

    var channelName = req.params.channelName;
    var chaincodeName = req.params.chaincodeName;
    console.log(`chaincode name is :${chaincodeName}`);
    // let args = req.body.args;
    let fcn = req.body.fcn;
    let username = req.body.username;
    let orgname = req.body.orgname;
    // let peer = req.query.peer;

    logger.debug('channelName : ' + channelName);
    logger.debug('chaincodeName : ' + chaincodeName);
    logger.debug('fcn : ' + fcn);
    logger.debug('username : ' + username);
    logger.debug('orgname : ' + orgname);
    // logger.debug('args : ' + args);

    if (!chaincodeName) {
      res.json(getErrorMessage('\chaincodeName\'));
      return;
    }
    if (!channelName) {
      res.json(getErrorMessage('\channelName\'));
      return;
    }
    if (!fcn) {
      res.json(getErrorMessage('\fcn\'));
      return;
    }
    // if (!args) {
    //   res.json(getErrorMessage('\args\'));
    //   return;
    // }
  }

```

```

    // }
    // console.log('args=====', args);
    // args = args.replace(/'/g, '');
    // args = JSON.parse(args);
    // logger.debug(args);

    let response_payload = await qsc(channelName, chaincodeName, fcn, username, orgname);

    // const response_payload = {
    //   result: message,
    //   error: null,
    //   errorData: null
    // }
    console.log("Printing response_payload " + response_payload);
    res.send(response_payload);
  } catch (error) {
    const response_payload = {
      result: null,
      error: error.name,
      errorData: error.message
    }
    res.send(response_payload)
  }
});

```

```

export const qsc = async (channelName, chaincodeName, fcn, username, org_name) => {
  try {
    logger.info("In qsc");

    const ccp = await getCCP(org_name);
    const walletPath = await getWalletPath(org_name);
    const wallet = await Wallets.newFileSystemWallet(walletPath);
    logger.info(`Wallet path: ${walletPath}`);

    let identity = await wallet.get(username);
    if (!identity) {
      logger.warn(`An identity for the user ${username} does not exist in the wallet, so registering user`);
      await getRegisteredUser(username, org_name, true);
      identity = await wallet.get(username);
      logger.info('Run the registerUser.js application before retrying');
      return;
    }

    const gateway = new Gateway();
    await gateway.connect(ccp, {
      wallet,
      identity: username,
      discovery: { enabled: true, asLocalhost: true }
    });

    const network = await gateway.getNetwork(channelName);
    const contract = network.getContract(chaincodeName);

    let result;
    result = await contract.evaluateTransaction(fcn);
    logger.info("Got the result: " + result);

    // Example of using axios to make an HTTP POST request
    let url = "http://127.0.0.1:5000/start_simulation";
    const response = await axios.post(url, result, {

```

```

      headers: {
        'Content-Type': 'application/json'
      },
    });
    const res = response.data; // Assuming the response is JSON
    logger.info("Got the response: " + JSON.stringify(res));

    await gateway.disconnect();

    return result.toString('utf8'); // Convert result to string if needed
  } catch (error) {
    logger.error(`Failed to evaluate transaction: ${error}`);
    return error.message;
  }
};

```

API to train model on data

This api is written in python. In the /startsimulation api, we get data from different channel in the request. Then we prepare divide this data in proper order. Then the we train the model on different data in the datasets and aggregate their results using various algorithms like Federated Averaging, Federated Adaptive Moment Estimation etc. We also compare the accuracy by training the model on the whole data.

Note- The model is created to train on the MNIST data.

```

# pylint: disable=too-many-locals
# pylint: disable-all

"""
high level support for doing this and that.
"""

> import logging...

app = Flask(__name__)
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')

history = None
save_path = None

@hydra.main(config_path="conf", config_name="base", version_base=None)
@app.route('/start_simulation', methods=['POST'])
def start_simulation():
    print("Currently in the start_simulation congrats")
    global history, save_path

    if request.method == 'POST':
        data = request.get_json()
        print("Hello world")
        print(data)
        train=[]
        trainloaders, validationloaders, testloader = preparedatasets(10, 10)
        t=[]
        v=[]
        for i in data:
            match = re.search(r'client(\d+)', i)

```

```

            if match:
                b=int(match.group(1))
                t.append(trainloaders[b])
                v.append(validationloaders[b])

trainloaders=t
validationloaders=v
print("Generating client")
client_fn = generate_client_fn(trainloaders, validationloaders, 10)
print("Client generated successfully")
print("Generating Strategy")
strategy = fl.server.strategy.FedAvg(
    fraction_fit=0.0,
    min_fit_clients=2,
    fraction_evaluate=0.0,
    min_evaluate_clients=2,
    min_available_clients=10,
    on_fit_config_fn=get_on_fit_config({
        "lr":0.01,
        "momentum":0.9,
        "local_epochs":1
    }),
    evaluate_fn=get_evaluate_fn(10, testloader),
)
print("Strategy generated successfully")
print("Starting simulation")
history = fl.simulation.start_simulation(
    client_fn=client_fn,
    num_clients=10,
    config=fl.server.ServerConfig(num_rounds=5),
    strategy=strategy,

```

```

        client_resources={'num_cpus':2}
    )

    print("Simulation started successfully")
    return jsonify({"succ": "Successfullt done the operation"})
else:
    return jsonify({"error": "Invalid request method"})

@app.route('/get_history', methods=['GET'])
def get_history():
    global history

    if history:
        return jsonify(history)
    else:
        return jsonify({"message": "History not available"})

if __name__ == "__main__":
    Port= 5000
    print("Backend is running ${Port}" )
    app.run(debug=True,port= Port)

```

3.9. Working

Step-1 : Open the Windows Subsystem Linux(WSL) and then go to the proper directory using cd command.

Step-2: Go to artifacts/channel(in project) and change the information regarding organisations, peers, orderers, certificate authority as per the network need.

Step-3 In WSL, go to channel directory and run command “./create-artifacts.sh”. This will create crypto-config folder which contain the artifacts of the channel.

Step-4 Open the docker-desktop on the system. Go to docker-compose.yaml in project and make proper changes in the file according to your network. Then go to artifacts directory in the WSL. Run command “docker-compose up –d”. This will create the containers for peers, ordererd, certificate authority etc in the docker-desktop. To stop this container, run “docker-compose down” in WSL in artifacts directory.

Step-5 Once the docker-desktop started the container successfully, the next task is to create channel. For this go to create-channel.sh file and make changes in the file according to the network. Then in WSL, go to proper directory and run command “./createchannel.sh”

Step-6 After the channel has been created, next task is to deploy the chaincode on the channel peers. The chaincode can be found in the artifacts/src/github.com/fabcar2/got/fabcar.go. Make proper changes in the deploy chain code file according to network need and run command `./deploychaincode.sh` in WSL.

Step-7 Go to `api-2.0/config` in (in project) . There you will find `connection-org1.json` and other similar name file. We need to add the proper certificates in them. We can find those certificates in `artifacts/channel/crypto-config`. In `ordererOrganizations`, we have the certificate for orderer. In `peerOrganizations`, we have the certificate for peers.

Step-8 After this we need to register the organisation. First start the app server by going to app directory in `api-2.0` and run the command `"nodemon app.js"` in the terminal. After this, go to postman and make a post request on this url `"http://localhost:4000/users"`. In the body give the username and orgName as parameter. After clicking on send, we get a token as response. Take the username, orgName and token and set them in the `main.py` file.

Step-9 Go to the `main.py` file in the `api-2.0`. Run the `prepare datasets` function. This will divide the mnist data equally among all the clients in different organizations and channel. Make sure that the `app.js` function in the `api-2.0` is running. (Note- Since I am not able to create a large data for the network, I have taken Mnist data and distribute them among the clients using this function.)

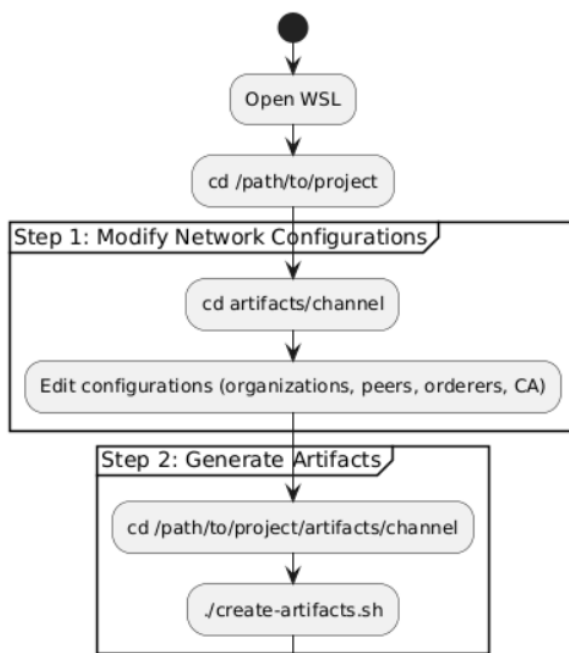
Step-10 The `prepare datasets` function call the `"http://localhost:4000/channels/:channelname/chaincodes/:chaincodeName"`, which further call the `invoke` function in the `api-2.0/app/invoke`. This function call the `StoresDataSetInfo` in smart contract.

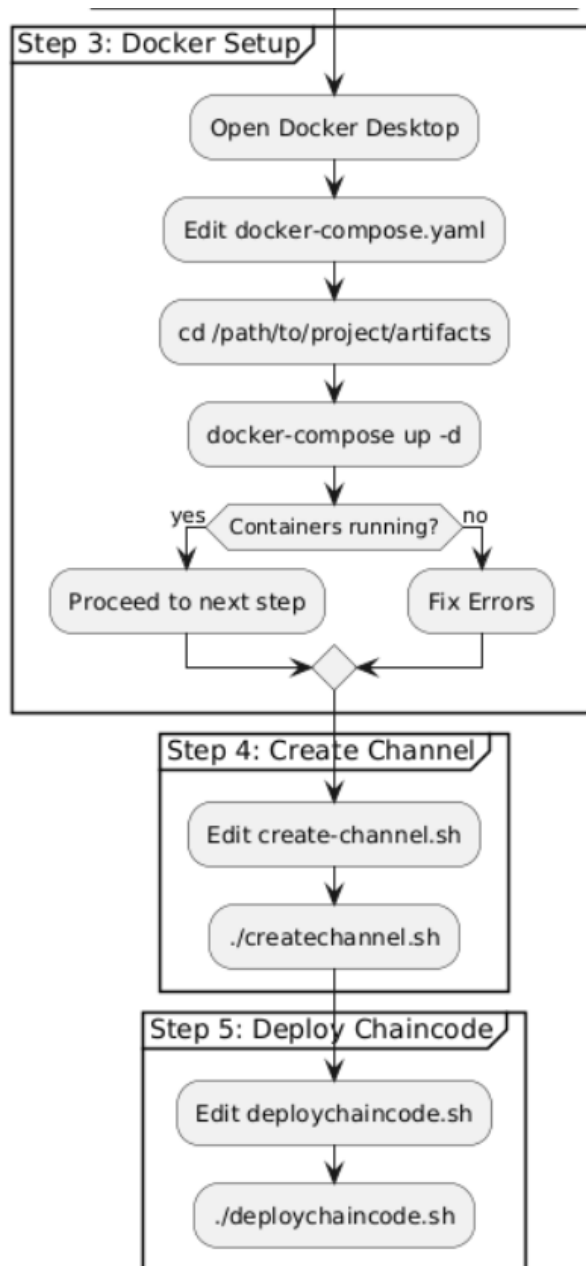
Step-11 Once the data for different channel has been stored in the ledger, we can start the federating training. For this go to `federatedlearning/backend.py` file and in the terminal run the command `"python backend.py"`. This will start the federated learning server used for training the data.

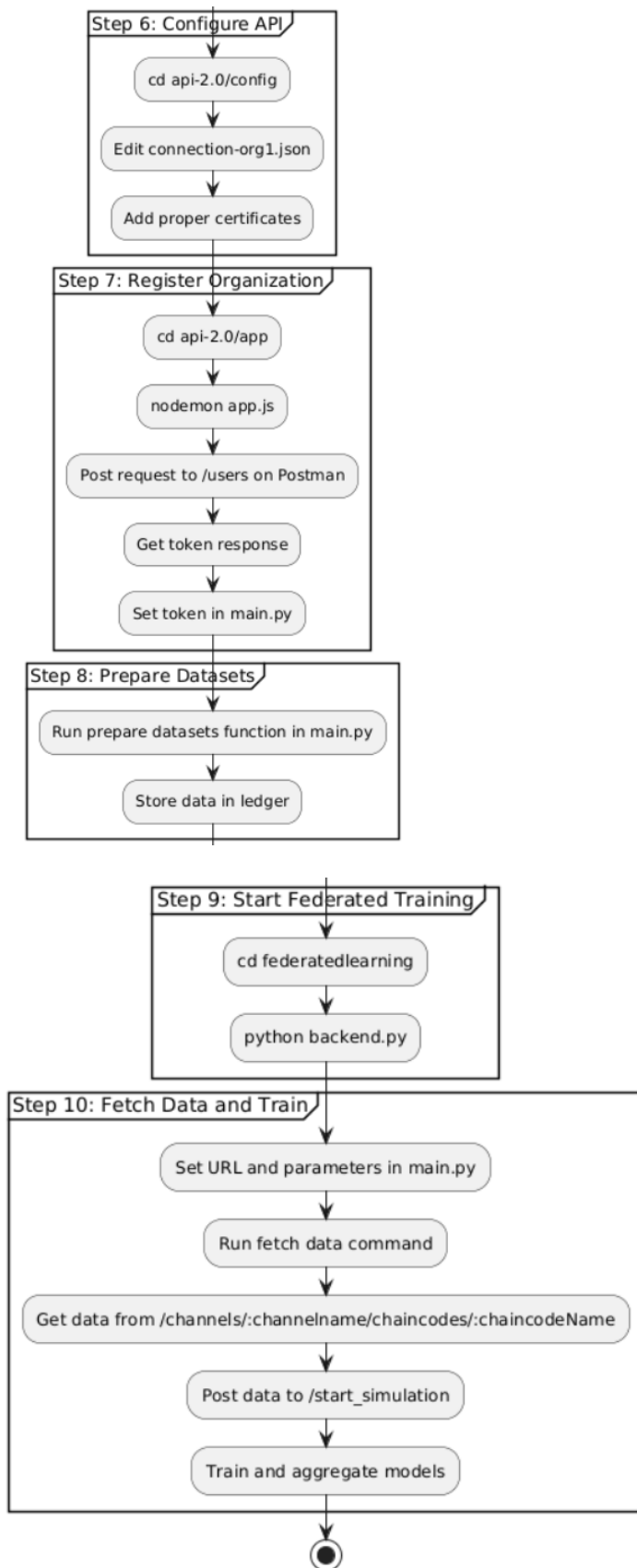
Step- 12 Go to the `main.py` file in `api-2.0` and set the proper url, name of the client, organization, client token, channel name and chaincode name.(Note- The client should be the member of the channel). Then run the `fetch data` command.

Step-13 The fetch data command make a get request to “http://localhost:4000/channels/:channelname/chaincodes:chaincodeName” which further call the qssc function in the api-2.0/app/qssc. This function call the GetDataInfo function in the smart contract. The GetDataInfo function return the data of a particular channel. Once the qssc function has get the data of different channel , it makes a post request to another api “http://localhost:5000/start_simulation”. This is the api where the python server is running. qssc function send the data for different channel to this api.

Step-14 In the api, the model is train for data of the different channel. The model for different data is then aggregated using various algorithms like Federated Averaging, Federated Adaptive Moment Estimation etc . Then we also combine the data of different channel and train them simultaneously. The result of the aggregated model and the model trained for all data is compared for accuracy and the lost.







3.10. Network Configuration

- In this project , I have created a single channel. In this channels there is a consortium of 2 organization (Org1 and Org2) each having 2 peer node (peer0org1, peer1org1, peer0org2, peer1org2). There are 3 orderer in the channel.
- For storing the data, I have used couchdb. There are 4 couchdb instances in the channel, one for each peer node.
- For the data, we have used MNIST data that contains 50,000 samples for training and 10,000 samples for validation. This data is used in independent and identical data form.
- In the project, I have created 4 clients, 2 for each organization. The MNIST data is distributed equally among this client in this project.
- For aggregating the results of different model, I have used three different federated learning algorithms and compare their accuracy. Different algorithms are Federated Averaging, Federated Adaptive Moment Estimation and Federated Proximal.
- In federated learning model I have used $lr=0.01$, $momentum=0.9$ and $local\ epochs=1$.

4.Result

4.1. Federated Averaging

Accuracy-

```
INFO : {'accuracy': [(0, 0.1043),
2024-07-07 15:46:23,735 - INFO - {'accuracy': [(0, 0.1043),
INFO : (1, 0.9468),
2024-07-07 15:46:23,736 - INFO - (1, 0.9468),
INFO : (2, 0.9678),
2024-07-07 15:46:23,745 - INFO - (2, 0.9678),
INFO : (3, 0.9765),
2024-07-07 15:46:23,760 - INFO - (3, 0.9765),
INFO : (4, 0.9771),
2024-07-07 15:46:23,763 - INFO - (4, 0.9771),
INFO : (5, 0.9777)]]}
2024-07-07 15:46:23,785 - INFO - (5, 0.9777)]]}
INFO :
2024-07-07 15:46:23,806 - INFO -
```

Loss-

```
2024-07-07 15:46:23,660 - INFO - History (loss, centralized):
INFO : round 0: 182.01992917060852
2024-07-07 15:46:23,675 - INFO - round 0: 182.01992917060852
INFO : round 1: 14.963489128276706
2024-07-07 15:46:23,685 - INFO - round 1: 14.963489128276706
INFO : round 2: 8.144381206482649
2024-07-07 15:46:23,689 - INFO - round 2: 8.144381206482649
INFO : round 3: 5.738522799452767
2024-07-07 15:46:23,691 - INFO - round 3: 5.738522799452767
INFO : round 4: 5.77473653270863
2024-07-07 15:46:23,700 - INFO - round 4: 5.77473653270863
INFO : round 5: 6.044387566274963
2024-07-07 15:46:23,711 - INFO - round 5: 6.044387566274963
```

4.2. Federated Proximal

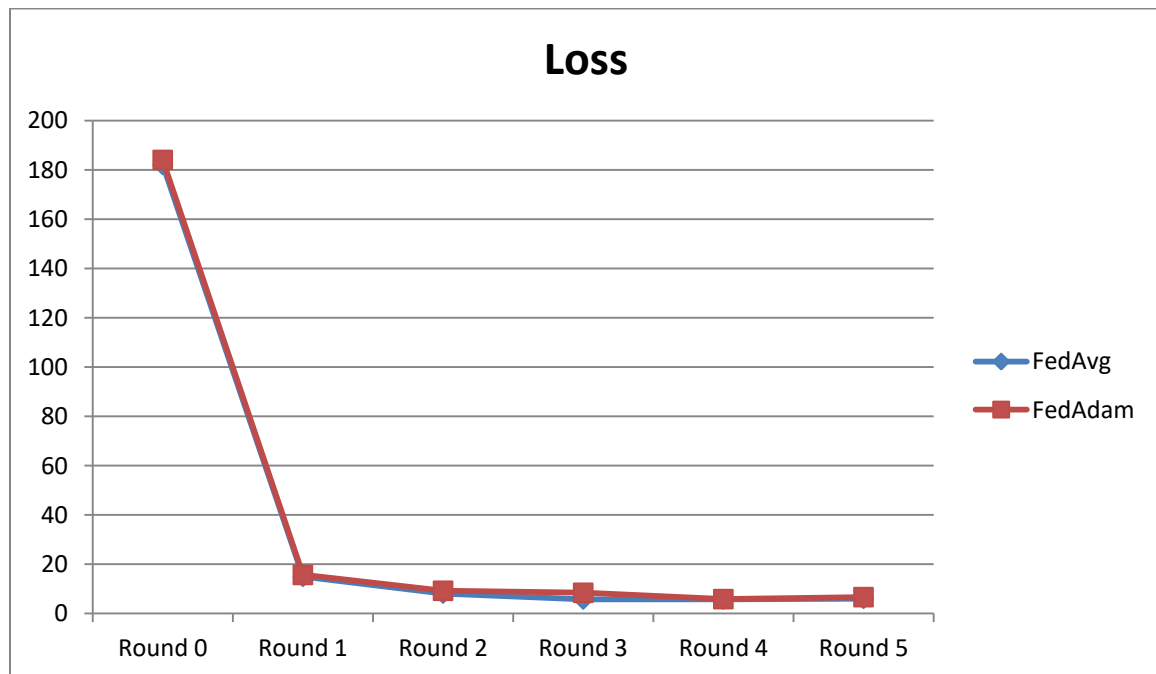
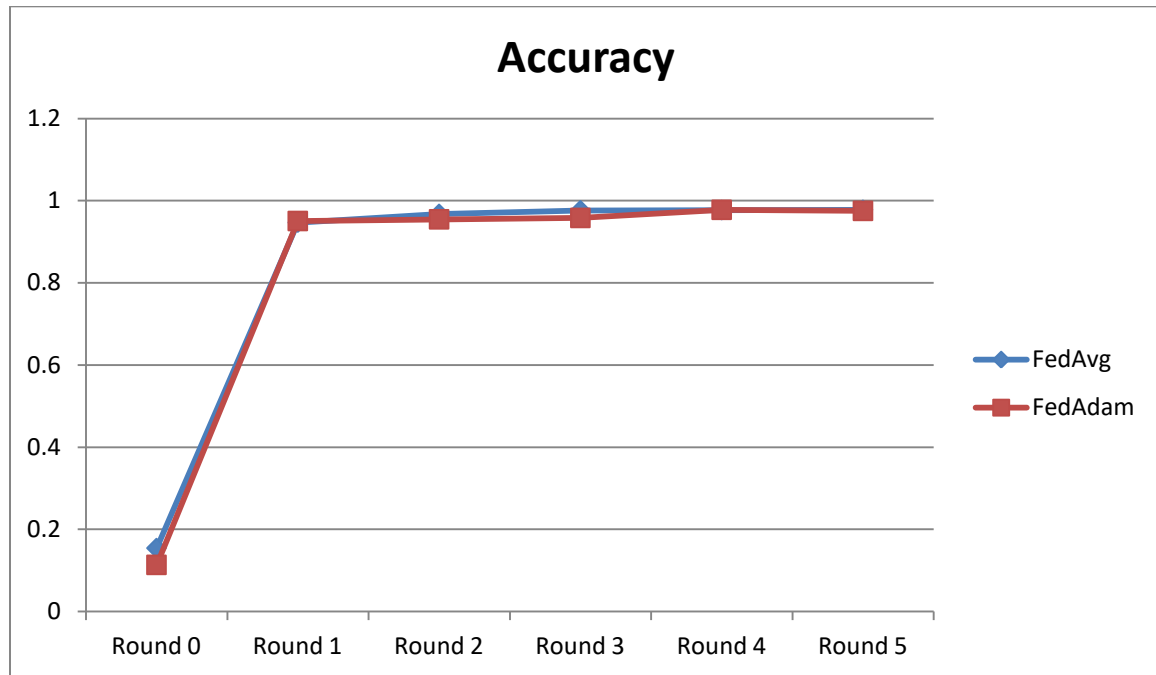
Accuracy-

```
INFO :          {'accuracy': [(0, 0.1135),
2024-07-07 16:07:37,061 - INFO -          {'accuracy': [(0, 0.1135),
INFO :          (1, 0.9505),
2024-07-07 16:07:37,064 - INFO -          (1, 0.9505),
INFO :          (2, 0.9646),
2024-07-07 16:07:37,065 - INFO -          (2, 0.9646),
INFO :          (3, 0.968),
2024-07-07 16:07:37,066 - INFO -          (3, 0.968),
INFO :          (4, 0.9778),
2024-07-07 16:07:37,068 - INFO -          (4, 0.9778),
INFO :          (5, 0.9752)]]}
2024-07-07 16:07:37,068 - INFO -          (5, 0.9752)]]}
INFO :
```

Loss-

```
2024-07-07 16:07:37,029 - INFO -          History (loss, centralized):
INFO :          round 0: 182.0577883720398
2024-07-07 16:07:37,030 - INFO -          round 0: 182.0577883720398
INFO :          round 1: 14.706685556098819
2024-07-07 16:07:37,032 - INFO -          round 1: 14.706685556098819
INFO :          round 2: 9.280929777771235
2024-07-07 16:07:37,033 - INFO -          round 2: 9.280929777771235
INFO :          round 3: 8.46462329174392
2024-07-07 16:07:37,035 - INFO -          round 3: 8.46462329174392
INFO :          round 4: 5.899968016339699
2024-07-07 16:07:37,036 - INFO -          round 4: 5.899968016339699
INFO :          round 5: 6.692959965002956
2024-07-07 16:07:37,039 - INFO -          round 5: 6.692959965002956
```

4.3. Comparison



5. Conclusion

Working on a federated learning project using Hyperledger Fabric was an enlightening experience that combined the principles of decentralized data processing with blockchain's security and transparency. Federated learning allows multiple parties to collaboratively train machine learning models on their local data without sharing the actual data, addressing privacy concerns. Hyperledger Fabric, a permissioned blockchain framework, provided the necessary infrastructure to manage this collaborative process securely and efficiently.

Through this project, I learned how to set up and configure a Hyperledger Fabric network, which included defining and managing various network components like peer nodes, ordering services, and smart contracts (chaincode). This involved creating and configuring multiple organizations within the network, each with its own peers and certificate authorities, ensuring that only authorized participants could join and interact with the blockchain.

One of the key aspects I focused on was the deployment and execution of chaincode, which encapsulates the business logic and governs how transactions are processed on the blockchain. I learned how to write, deploy, and invoke chaincode functions to facilitate secure and verifiable updates to the blockchain ledger. This was crucial for recording the contributions of each participant in the federated learning process, ensuring transparency and accountability.

The integration of federated learning algorithms with the blockchain network was particularly challenging yet rewarding. I explored various federated learning frameworks and adapted them to work within the constraints of a decentralized environment. This involved designing protocols for securely aggregating model updates from different participants and implementing them using chaincode. By doing so, I ensured that the learning process remained efficient while maintaining data privacy and security.

Furthermore, I delved into the use of private data collections in Hyperledger Fabric to handle sensitive information that should not be exposed to all participants. This feature allowed me to enforce fine-grained access controls and ensure that only authorized entities could access specific pieces of data, thus enhancing the overall privacy and confidentiality of the federated learning process. Overall, this project provided a comprehensive understanding of how to leverage blockchain technology to enhance the security, transparency, and trustworthiness

6. Future Prospect

The future prospects of integrating federated learning with Hyperledger Fabric are highly promising, offering significant advancements in data privacy, security, and collaborative analytics across various industries. As data privacy concerns continue to escalate, this combination provides a robust solution by ensuring sensitive data remains localized while enabling secure and transparent interactions through blockchain. Healthcare, finance, and smart cities are just a few sectors that can benefit from this technology, where secure collaboration on sensitive data is critical. Moreover, the transparency and immutability of blockchain enhance trust among participants, making it easier to comply with regulatory requirements. The ability to securely aggregate and analyze data from multiple sources without compromising privacy opens up new opportunities for innovation and insights, potentially transforming how organizations approach data-driven decision-making. Overall, this integration not only addresses current data privacy challenges but also paves the way for more secure and efficient collaborative machine learning frameworks in the future.

7. References

- Flower, www.youtube.com/@flowerlabs
- IBM, www.ibm.com
- Hyperledger, www.hyperledger.org
- blockchain-systems/ScaleSFL, <https://github.com/blockchain-systems/ScaleSFL>
- Flower Framework, www.flower.ai
- Wikipedia, www.wikipedia.com
- PavanAdhav, www.youtube.com/@PavanAdhav
- <https://github.com/adhavpavan/BasicNetwork-2.0>
- AdityaJoshi, www.youtube.com/@AdityaJoshi
- Geeksforgeeks, www.geeksforgeeks.com
- Wikipedia, www.wikipedia.com