Consensus using Practical Byzantine Fault Tolerance

A Project Report

Submitted by:

Raj Dhamsaniya (201501025)

in partial fulfilment for the award of the degree

of

BACHELOR OF TECHNOLOGY

IN

INFORMATION AND COMMUNICATION TECHNOLOGY (ICT)

at

School of Engineering and Applied Science (SEAS)

Ahmedabad, Gujarat

May 2019

DECLARATION

I hereby declare that the project entitled “Consensus using Practical Byzantine Fault Tolerance” submitted for the B. Tech. (ICT) degree is my original work and the project has not formed the basis for the award of any other degree, diploma, fellowship or any other similar titles.

Signature of the Student

Place:

Date:

CERTIFICATE

This is to certify that the project titled “Consensus using Practical Byzantine Fault Tolerance” is the bona fide work carried out by Raj Dhamsaniya, a student of B Tech (ICT) of School of Engineering and Applied Sciences at Ahmedabad University during the academic year 2018-19, in partial fulfillment of the requirements for the award of the degree of Bachelor of Technology (Information and Communication Technology) and that the project has not formed the basis for the award previously of any other degree, diploma, fellowship or any other similar title.

Signature of the Guide

Place:

Date:

ACKNOWLEDGMENT

This project is inspired by the implementation of different technologies like Hyperledger Fabric, Zilliqa, Corda and bitcoin. I am grateful for a number of friends and colleagues in encouraging me to start the work and supporting me through the implementation phase. I would like to thank Prof. Sanjay Chaudhary for introducing the concept of technology about a year ago which lead me to this project implementation. I would like to thank Dr Vikram Sorathiya for providing the knowledge regarding the technology mentioned and resolving the doubts. I also like to thank StackOverflow community for providing support for my implementation problems and thanks to all the creator of this technologies.

ABSTRACT

Blockchain introduced in 2008 along with bitcoin for transfer of tokens from one account to another solving the problem of double spending. Now many blockchain solutions are being implemented by various providers. This solution defers from one another because of different algorithms for cryptography, data distribution, Execution flow, type of blockchain etc. This project covers the execution life cycle part of a transaction or a consensus part and does not focus on cryptography, encryption or TLS for the security purpose. The objective of the project is to create a scalable system to achieve consensus. Scalability is mainly focused on supporting the high number of peers while providing proper transaction throughput. For scalability of the system, microservice architecture is used. In this project, for consensus Practical Byzantine Fault Tolerance (pBFT) algorithm is used.

pBFT algorithm introduced way back in 1999. The previously suggested algorithms assumed a synchronous system whereas pBFT is proposed to work in asynchronous systems. There are many blockchain solutions which implement pBFT or some modified version of pBFT currently. Here Hyperledger Fabric and Zilliqa is referred for the architecture implementation. As the Execute-Order-Validate architecture is used Kafka is used for ordering service. As contract script is pluggable here only amount transfer simple script is used as an example and testing.

TABLE OF CONTENT

TABLE OF FIGURE

1 Introduction

1.1 Problem Definition:

- To state and modify the problems in the implementation of pBFT for consensus mechanism in blockchain technology by various methods.

1.2 Project Overview/Specification:

This project is deployed on 4 PCs assuming they create a network and applying pBFT on them. Because of microservice architecture, Docker is used for containerization. The whole code of the application is written in golang because of its use in distributed systems and other various reasons. Docker-compose is being used to automating the deployment by some amount. gRPC is used for communication between two service or peers because of its reliability and speed. Kafka is a highly scalable streaming service. It provides Crash Fault Tolerance when used with the Zookeeper server. Because of containerization, we can scale the appropriate service as needed. CouchDB is used as database for the system, to store the local state.

- Hardware:
    - 4 intel i7-7700 processor running @3.6 GHz clock frequency
    - 8 GB RAM
    - 1 TB HDD
- Software:
    - Ubuntu 16.04 LTS
    - Docker
    - Docker-compose
    - CouchDB v2.3.1
    - gRPC
    - Kafka v2.2.0
    - Golang v1.11.0
    - Protocol buffers v3

2 Literature Survey

2.1 Existing System

For this project two of the existing system is taken into consideration.

1. Hyperledger
2. Zilliqa

Currently, there are platforms which provide thousands of transaction per second but when we look at the transaction throughput of currently available blockchain technology it comes to few in comparison to the current requirement. First consensus protocol which introduced in blockchain technology was Proof of Work (PoW) in bitcoin [1]. As of today, there are many algorithms available and many of them implements the order-execute-validate architecture. As mentioned in [2], when transaction arrives it first ordered, then executed on each peer and then validate or finalize by active machine replication for this type of architecture. It creates a problem when the transaction throughput is high as this type of system is limited by sequential execution. Every transaction needs to execute on every peer for validation of a block which can be a bottleneck.

Hyperledger implements the Execute-Order-Validate architecture and modular approach for consensus protocol while running in a permissioned environment. Because of the architecture, it is easy to scale the fabric blockchain for higher throughput for a limited number of organizations. In this architecture, First, any invoked transaction is executed as per its endorsement policy. Result of that transaction in putstate, getstate and deletestate is propagated to Ordering service node which orders the transaction as they reach to the node. After sorting the transaction it creates a block based on block length or time duration and sends it to gossip service which propagates that block to every peer. In Validate state every transaction result is checked with current state and if all is fine then the effect of putstate is done to the database. Validation of this transaction result happens sequentially. Every transaction is masked if it's successful or not and block added in the chin. This gives high throughput as transaction execution happens concurrently. However, if the no of organization increases then it's not a viable solution. Because for pBFT, as the number of peers increases, the number of messages is exponentially increased. For a government like entity where there are many organizations implementing various contract script and

having various assets. Fabric has endorsement policy which determines the peers on which contract will be executed which means that it makes the execution predictable for a set of transactions. Scalability of the fabric is based on its architecture. Because of different microservice, we can scale that system well, and that's why its architecture is implemented in this project.

Zilliqa is another blockchain technology which implements the pBFT algorithm for consensus. It is a public blockchain. Zilliqa creates the shards of peers and executes transaction concurrently [3]. Because of this, the overall throughput of the system is increased as a number of peers increase. It uses the Order-Execute-validate architecture but because of the network sharding the throughput is higher than previously implemented pBFT protocol systems with traditional architecture. Because of its public nature, it has many peers. For supporting a higher number of peers it needs to resolve the increasing number of message issue, which they implemented by using public key cryptography [3]. Public key cryptography was expensive when the pBFT was first introduced but it is cheap now and it can be implemented. For this project cryptography is not included and implementing of this is out of scope for the project.

2.2 Proposed System

The proposed solution is a hybrid of two technology which mentioned earlier. Microservice architecture is effective and is popular because of its scaling properties. Because of same I have decided to go with it. The proposed system implements Execute-Order-Validate architecture. Contract Script is pluggable and can have multiple use cases. The whole system is implemented using golang. The contract code which is being written is also required to be in golang.

The proposed system leverages the modular approach of Fabric as well as it leverages the approach of sharding from Zilliqa up to some extent. This system Executes transaction with network sharing approach and further steps of ordering and verification are the same as Fabric. It is possible that the number of transactions that are ineffective due to the double-spending problem at verification stage (i.e. transaction whose getstate is not same as the current state of the database) will increase but it is less possible that in given time out value, there are transactions whose getstate will match and the later one will fail. Less probability of that occurring is because of many peers are connected, many transactions will be invoked. However, it will occur and

when it occurs, the transaction will be marked as invalid and it will not change the database (world view).

For the world view database, I have used CouchDB which provides versioning of the records and stores the data as a Key-Value pair. For the ordering service, I have used Kafka which is a pub-sub scheme from apache. Kafka can write up to 700 MB/Sec with disk as storage hardware because of its sequential write functionality [4]. Reading from Kafka happens in O(1) time and for reading only offset is need to remember by a consumer or in this case ordering service. Kafka also provides the crash fault tolerance and it is used with the zookeeper server. Ordering service is not required to run on all the peers but having a minimum of 3 replicas ensures the higher read-write rate so it is replicated on 3 systems for the deployment part. All the service will run inside a docker container which enables to control the fault tolerance and replication for scaling by various methods. Jenkins is used to automating the deployment of containers. For keeping the image size low multi-stage build is used to create the docker images.

Ordering service will create the block and send that block to gossip service which broadcast to every peer. Every record in the database will have a transaction ID associated with it which shows the transaction responsible for that state in the database. Because of this, it would be easy to restore the database in case of an anomaly in the data. Data will only change if the transaction is valid. If it's required that the organization can't share data then the whole database would be kept on the specific trusted node but can be called when a transaction related to that data is occurred. So at that point database is sharded and a peer or node is act as a database for a peer which are executing the transaction. It secures the database as well as make it less deterministic for execution environment for any invoked transaction. As this solution focus on a large network where the knowledge of the organization is known, a membership service provider as in fabric would be sufficient. However, if the network is open then PoW for being a member makes sense as in Zilliqa.

2.3 Feasibility Study

The Given solution regarding the database sharding can be applied with some modification to existing Fabric network while for parallel execution we need to change in the execution pattern. We can modify to only execute on selected endorsed peers to execute in a shard. We also should implement public key cryptography for

communication. This solution does not have an inbuilt cryptocurrency and because of the pluggable smart contract, we can implement different use case with it from tracking of a food product to money distribution trail of the budget of a nation. It can work as a base of some bigger solution where this type of consensus required. Because of less dependency among service, we can change a service as per the requirement because one fit all solution does not exist. Because of the ledger, all the data that is being generated can be used in machine learning or Artificial Intelligence algorithms.

3 System Analysis & Design

3.1 Implementation Process

System design is a multi-stage process. It started with finding the problem with current systems. One particular problem is scaling of the blockchain, and so the project focus was on how technology should be implemented for it to be scalable. Project implementation can be seen by commits in the repository on Github. The implementation of the project occurred in the following step.

- A server and client is implemented.
- gRPC implemented.
- Implementation of execution replication(single function). – multithreading in golang
- Containerization of system.
- Multi-stage builds for containerization.
- Contract support Implemented.
- CouchDB as Database used.
- Query states (GetState, PutState, DelState) defined and used for the database change.
- Kafka as ordering service tested using ‘sarama’ library.
- Kafka integrated into the project.
- Gossip service implemented.
- Validation package added to the peer for transaction validation.
- Deployment and testing.

3.2 Component selection

Every step in the implemented process is gone through a number of choices from language selection to database selection. Here is the detail on how the selection is made, what benefits it provides to the system and why it is important to the system.

- **Golang:** There are multiple reasons to go with golang. First and foremost would be golang is great for concurrency management. Nowadays CPUs are having multicore and number of cores are increasing and effective use of those is necessary for any scalable systems. Golang has goroutine which takes only 2KB memory from the heap and this creating more goroutine is easy and cheap.

Golang is compiled language meaning code can be converted into binary and that turns out to perform as low-level language. Also, it is backed by Google and is in widely used in distributed systems.[5]

- **gRPC:** APIs implementation is very high in today's implementation. APIs are easy to implement and support for the APIs are easily available. As the microservices are loosely coupled it makes sense to implement APIs but APIs use call-and-response architecture necessary for HTTP + JSON while gRPC is built on top of HTTP/2 which allows client side and server side streaming. Switching from JSON to protocol buffer is actually good as protocol buffer defines and restricts the message types. gRPC is better when the payload is larger compared to REST and provides fully integrated pluggable authentication [6].
- **Docker:** Docker is a containerization technology. As microservice architecture is required containerization of these services can be effective from these reasons. It isolates an execution environment from others so that application can be independent of another environment. It increases the reliability of application because one app or service of a huge app can't be affected by outside changes from other app or service. It reduces the infrastructure lock-in problem as Docker is independent of base OS and will run the same on different OS.

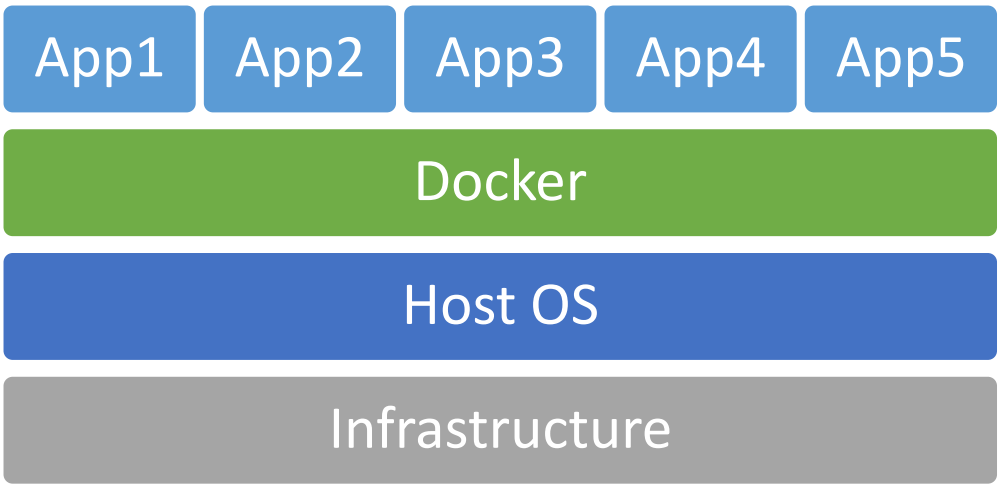

Figure 1: Containerized Application [7]

Here is shown how containerized applications appear in the stack. Virtual machines will virtualize the hardware while docker is abstraction at the app layer. VMs will have the full copy of an OS while container only required the app dependency and code only because the container can run on the same

machine and share the OS kernel. Because of this containers are also much faster than OS to boot. Docker is launched in 2013 and is one of the leaders in containerized technology. There are lots of support available for it and that's why I choose to use Docker. Dockerfiles define the code and the environment for Docker containers. Dockerfile creates a Docker image which when run on docker engine becomes containers. Docker prebuild images can be used to migrate the application and it is essential to keep the image size down. Instruction in the dockerfile adds a layer into the image and one need to clean any component which is not needed moving forward. For this, I have used the docker multi-stage build [7]. When building image we need the go execution environment but when we have created a binary of that go file then the only binary is sufficient to execute the programme on base os containers like Debian. It significantly decreases the size of the image.

- **Docker-compose:** When an application will have multiple dockerfile then it is hard to build and deploy all the files. It is also hard to get context outside of the folder in which dockerfile lies [7]. So for the deployment of multiple services, docker-compose can be used and it's easy to implement compared to other norms.
- **CouchDB:** Database choice is an important choice in any application. In this system, contract script is written by the user and it is up to them which field of data is there. The classic relational database may not be appropriate for this option. We need to have a database which can save unstructured data. A key-value pair database is good for such a scenario. CouchDB stores the data as a key-value pair in the document. CouchDB provides HTTP based REST API which makes communication with database easier. CouchDB is a NoSql Database. CouchDB provides replication whose function is to synchronize two or more database [8]. It also provides the incremental version control which is useful in the validation of the transaction. Because of features like these the CouchDB used as the database.
- **Kafka:** Apache Kafka is a distributed streaming platform. A streaming platform has three key capability [4].
    - Publish and subscribe to streams of records, similar to a message queue or enterprise messaging system.

- Store streams of records in a fault-tolerant durable way.
- Process streams of records as they occur.

Kafka runs on one or more servers that can span multiple datacentres. Kafka cluster stores a stream of records in categories called topics. Each record consist of a key, a value and a timestamp. The topic is a category to which record is published. Topics in Kafka are multi-subscriber that means, a topic can have zero or more consumer that is subscribed to that topic [9]. For each topic, Kafka maintains a partitioned log that looks like shown below.

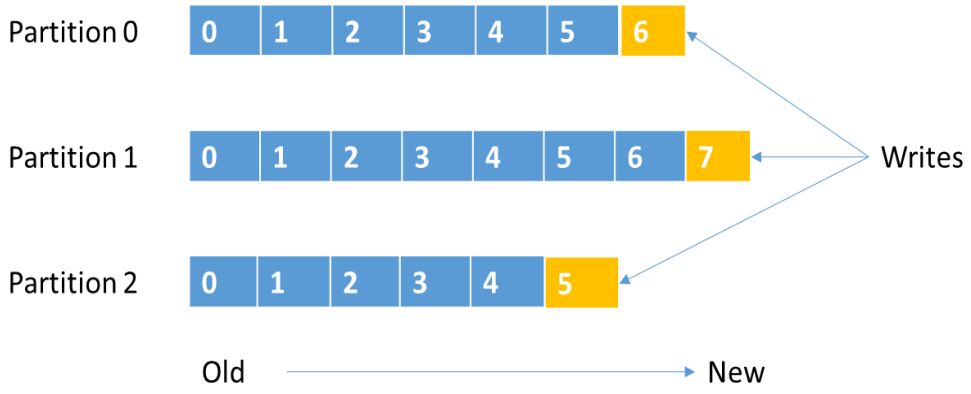

Figure 2: Anatomy of Topic

Each partition is an ordered, immutable sequence of records that is continually appended to a structured commit log. As record or transaction keeps coming to every partition will append the record in an ordered fashion. Every record in partition assigned a sequential id called Offset by which we can identify the record on the partition. Kafka's performance is effectively constant with respect to data size so storing data for a long time is not a problem. This is possible

because consumer only needs to know the offset which it read now.

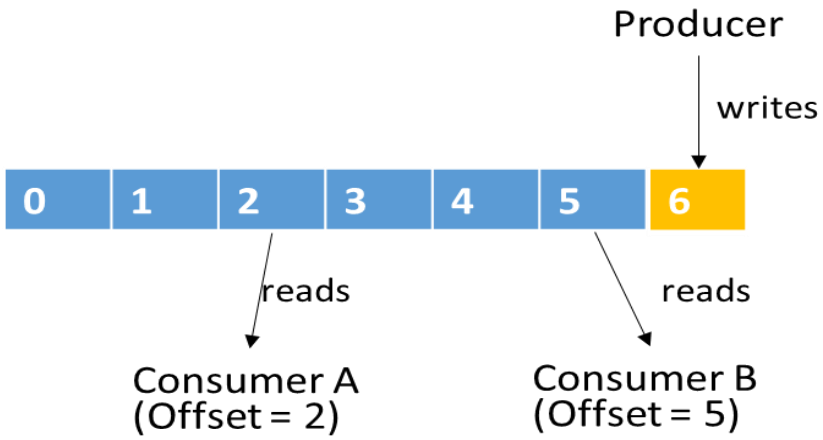

Figure 3: Partition View

As shown in Figure 3, every consumer has its own offset. Generally, the consumer consumes the record linearly by advancing its offset but it is possible that it can consume in any manner required as long as the offset is available. This provides the advantage in comparison to the messaging system. For messaging system queue management is necessary and for publisher subscriber scheme it is necessary that all nodes are alive but with this mechanism, we can use the best of both the system. For use the system in golang I have used "Sarama" library which is MIT licenced go client library. Running the Kafka in VM creates the problem in the local network as switch takes only a mac address and it would be native OS but if we implement it on PC running on the native OS then Kafka implementation is quite easy.

Above criteria show why each and every component is selected and now it is described how it's implemented in the next section.

3.3 Design and Implementation

The design choice of the system is based on the dilemma that the blockchain system is a big distributed system. What is necessary for a system should not be based on the OS of the node of the distributed system. Every peer of the system required to have some service without it can't function. Some of the services is such that for a system as a whole it is required and so not every peer need to have it.

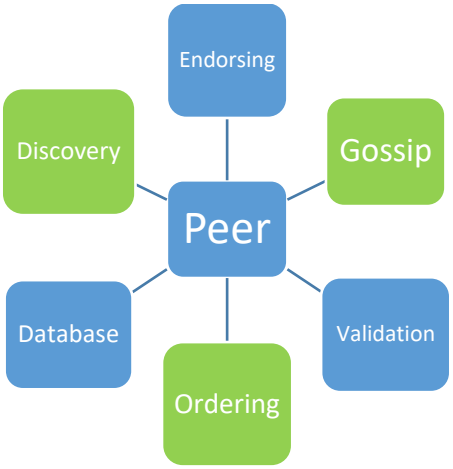

Figure 4: Peer Services

Every Service defined in the blue box is required to have for a peer. Without it functioning on the full order for a peer is not possible. It may possible that in the system some pees are not endorser but in this system, it is required for a peer to have execution environment for transaction suggested. Every service talks to its related service using gRPC. Because of service talks to each other endpoints are fixed or in other terms, the RPCs are fixed by protocol buffer. Here the design of the system as a whole is given.

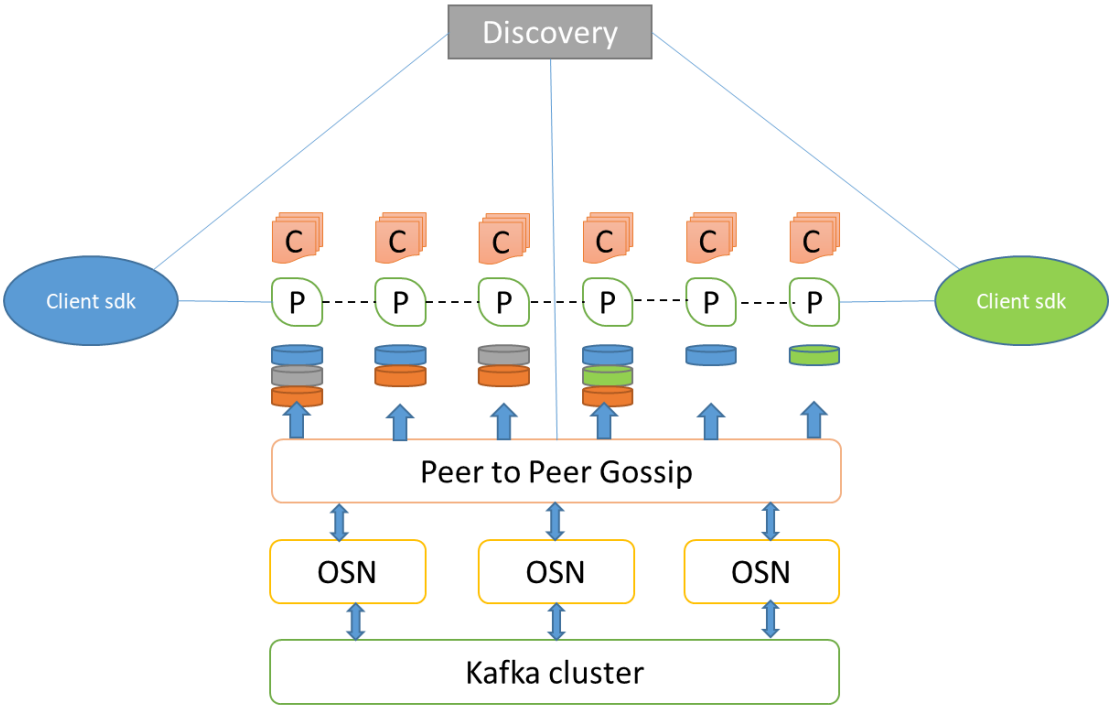

Figure 5: System Architecture

Note that every peer knows other peers address as to where to call for the database when a transaction invoked and it uses the other database. Different database to different

peers states that not every peer needs to have rights for the database access to endorse a transaction. Here is the port is given on which every service will run.

- Discovery: 50050
- PeerManager(Validation): 50051
- Gossip: 50052
- Contract(Endorsing): 50053
- Ordering: 50054

Use and importance of every file in the repository are describe here. Name of the repository is the consensus.

Consensus

- **Core:** This is a directory containing multiple directories for a different purpose. Every call for the peer is accepted here. Call for validation and endorsement first approach here.
    - **Contract:** Directory containing multiple files required for every contract to work in the system.
        - **Contract.go:** Defines the interface for every contract. Responsible for connecting with database to query for the invoked transaction.
        - **Handler.go:** Responsible for creating service for invoking and initialization of transaction. It accepts the request from the peer to endorse a transaction.
    - **Peer:** Directory containing a file to handle all the request related to the contract.
        - **contractHandler.go**: It is responsible for all the request to the peer. It also submits the endorsed transaction with transaction response to ordering service.
    - **Validate**: Directory for validation of block received from gossip service.
        - **validate.go:** it first validates the transaction by checking with an endorsement policy for all the transaction concurrently. After

that, it checks each transaction with its GetState and if it's a match it changes the database according to PutState and DelState queries. Every transaction will be masked as true for a successful transaction and false for an unsuccessful transaction. Masked block then added to the ledger. Double spending problem is taken care at this stage.

- **Discovery:** This directory is responsible for discovery or registry service. Dockerfile inside the directory is responsible for the build and launch the registry service.
    - **discovery_server.go:** Every peer when joining the network will register itself with the discovery server. When gossip service is needed the address of all the peers it will call this service.
- **Example_cc:** This directory stores the example contract script. Docker file will launch contract in a container to execute.
    - **example_new.go:** This implements the required interface for an example meaning two methods, Init and Invoke. The example does not concern itself with connection and gRPC as it is taken care by contract.go and handler.go file.
- **Gossip:** Gossip directory contains the file for gossip service. It also contains the dockerfile for build and deployment of the service.
    - **Gossip.go:** It is responsible for accepting a block from ordering service node and broadcasting it to all the peers.
- **Helloworld:** It is the client which invokes the transaction.
- **Orderer:** The directory contains a directory and file. It is responsible for ordering service and Kafka client, producer and consumer. Dockerfile in it will build and creates a docker image.
    - **Kafka:** Contains a file for handling Kafka for ordering.
        - **kafkaHandler.go:** This file will handle all the request to interact with Kafka. Consumer runs continuously in a goroutine for continuous synchronization for coming transaction. The producer will produce a message when instruction is given in another thread.

    - **serviceNode.go:** This file is responsible for fulfilling the client request of a specific block and call the functions from kafkaHandler.go file to initiate the Kafka service.
- **Protoc:** This directory defines the protocol buffer files for gRPC to use. It contains the proto file for all the services.
    - **Contractcode:** Defines the RPC and messages required for the contract.
    - **Core:** Defines the RPC and messages for a peer.
    - **Discovery:** Defines RPC and messages for registry or discovery service.
    - **Gossip:** Defines RPC and message for gossip service.
    - **Orderer:** Defines RPC and messages for ordering service.
- **Scripts:** Script to execute on Ubuntu 16.04 LTS for installation and testing of the system. The system is OS independent but the script is only for a development environment which is Ubuntu.

3.3.1 Docker Image Creation

In this architecture, it is important to create an efficient image of required service from dockerfile. The steps to create the docker image is mentioned below.

- Choosing the base image. In most of the cases its golang:v1.11
- Download the required library, as most service implements gRPC and uses protobuf one need to install it on the image.
- Then one needs to install the protoc-gen-go which is a generator of go file from protoc file.
- Next step is to create a work directory in the image.
- After creating the work directory one need to copy the required file from the current directory to docker image.
- Then the generation of go file from protoc file is necessary as the dependency will change with change in path structure.
- After generating the pb.go files the executable binary will be created.
- After creating an executable binary, it will be used in the multi-stage build.
- Next is to choose the base image as Linux distro which in this case is chosen Debian.

- Create a work directory.
- Copy executable binary from previous images to the new Debian base image.
- Expose appropriate port for communication.
- Create an entry point which generally is the execution of executable binary.

Note that If we do not use multi-stage built then the output size of creating the image will be greater than 1.2 GB which is not appropriate for a production environment. After implementing the multi-stage built the size of the image is limited to 300 MB at most for the project. When creating go files from proto files it will create error as the path of file and path of imports will mismatch so every output from protoc-gen-go will go in '$GOPATH/src' in the image.

3.3.2 Docker-compose

When there are multiple docekefiles in the repository and we need to build and deploy each we can use docker-compose. Another requirement for docker-compose is for context, meaning when we need to copy only some portion of the whole codebase into a Docker image and it may not possible that Docker file and required code share the same directory. As an example in this project repository, every dockerfile is out of context for the concern of protoc. Because of these reasons docker-compose needed. Now for the use of docker-compos, we need to create a "docker-compose.yaml" file. Structure and building of yaml file is given.

- First, a version is defined for the file. In this case, it's 3.3
- In the next step, services are defined.
- For every service, if we need to create a docker image then context and path are provided.
- However, for prebuild image from docker-hub, we can just specify the image and while building the docker-compose it will skip that particular service. For this type of images pulling is required.
- Every service needs to specify its exposed ports for port binding.

3.3.3 Service Endpoints

Every service defined has endpoints or RPCs by which it communicates with another service. Endpoints matter because a change in one service should not affect on how service behave from outside. This way we can expect ordering service to order

with or without the use of Kafka. With this type of architecture, one can change or optimize a service implementation without worrying the effect on the whole system. Here service endpoints are described.

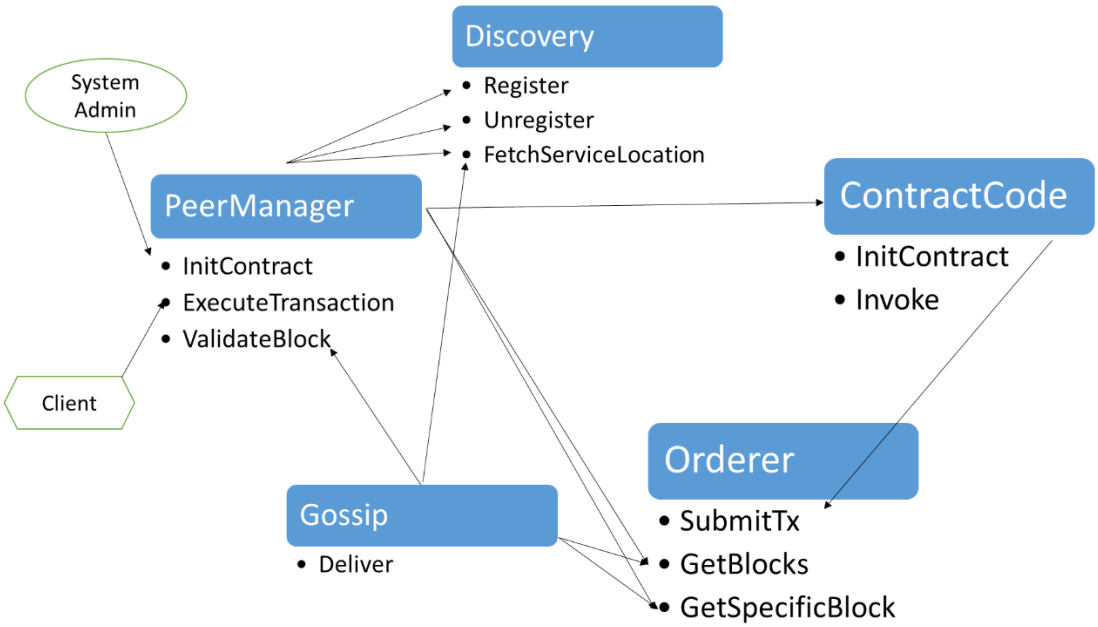

Figure 6: Service Endpoint Communication

This figure shows that every service knows only the endpoint or outer layer of another service. Now if we want to create other partition in Kafka and store the created blocks in that partition we can do in without error from other service and that provides a larger opportunity for the future. Further, every service endpoint is elaborated.

- **Discovery:**
  - **Register:** Peer calls this service when it joins the network. It takes input as 'Registration' which includes name, ipv4 and port of the peer manager service.
  - **Unregister:** Peer calls when it needs to leave the message or frequent peer call is not responsive this will remove the details of a peer as takes 'Registration' as an input.
  - **FetchServiceLocation:** Gossip service calls this method when it needs all the available peer address in the network. This method returns an array of 'Registration'. ContractCode may call this service as it needs the information regarding which database has required info.
- **ContractCode:**

- **InitContract:** This process is invoked by PeerManager. It will return a stream of 'TransactionResponse'. The stream is because as transaction executes it will generate different types of queries and it will be transmitted to where the transaction is invoked. There are three types of query GetState, PutState and DelState. Every TransactionResponse contains a query type and payload which is associated with the query. i.e. If the query is GetState then Key and version of record is the payload.
- **Invoke:** Similar to InitContract when a client invokes a transaction it will be invoked. Also, Invoke returns the stream of 'TransactionResponse' similar to InitContract for the same purpose.

- **PeerManager:**
  - **InitContract:** This can be called by contract administrator only as to initialize the transaction. This procedure takes no input and returns 'ExecuteResponse' to the client.
  - **ExecuteTransaction:** This can be called by any client using this application. It takes 'ExecuteTx' as input which is made of a string storing name of the transaction to call and another array of a string containing arguments of the transaction. This transaction first calls the Invoke from contractcode collets the 'TransactionResponse' for endorsement. After receiving sufficient endorsement it will send it to ordering service by calling 'SubmitTx' service of it.
  - **ValidateBlock:** This service is called by gossip service for validation of block. It accepts block message from gossip protocol buffer file as input containing an array of endorsed signature, timestamp and payload.
- **Orderer:**
  - **SubmitTx:** It will be called by PeerManager for adding the transaction to Kafka and ordering it. It talks to Kafka producer API for adding transaction to Kafka partition.
  - **GetBlocks:** If a peer has missing block after an offset then it calls this procedure for blocks using offset. The procedure returns the stream of blocks after the provided offset.
  - **GetSpecificBlock:** It returns a block whose offset is provided while calling the procedure.

- **Gossip:**
    - **Deliver:** Orderer calls this procedure when it needs to deliver a newly created block to deliver to all the peers. This procedure will call for fetchServiceLocation to get the address of all the available peers in the network.

3.3.4 Transaction Finality flow

The flow of how a transaction will reach its finality is shown in this section. As this is Execute-Order-Validate Architecture Execution is the first step. The process is shown in figure 7.

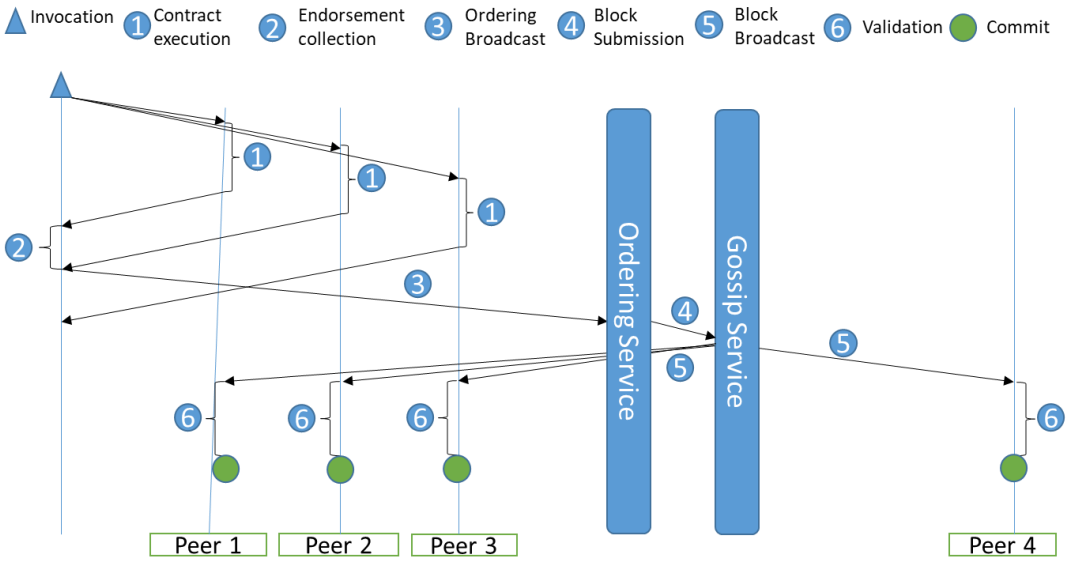

Figure 7: Transaction Flow

After the transaction invocation first process is contract execution. When transaction invoked it is not necessary that every peer execute a transaction at the same time synchronously. When the required endorsements are received, further endorsed transaction result will be ignored. The different thing from Fabric here is if the peer doesn't have access to the database then it will call another peer which has one and accumulate GetState queries. We can ensure that proper transactions can only call the GetState for such a database. For this type of transaction, PutState will be sent to all the peers but the only peer with such database access can view and modify its state. This is to ensure that the execution of the transaction is not destined in a fixed environment.

After the endorsement collection transaction with its result sent to ordering service. As mentioned earlier ordering service uses Kafka for the ordering of

transactions. After ordering it will send the block to gossip service. We can cut the block by many methods. Let's say we take block size 1, then for every transaction, there will be block. It is okay if transaction throughput is less but when there are thousands of transaction it won't be effective to sign each transaction/block and broadcast it. It is also expensive to verify the signature from ordering service for each transaction on high throughput. So we will create a block of the transaction and sign it. With this Sign and verification, overhead will go down. First, if I choose to cut block by a number of transaction then it is possible that if for some time new transaction hasn't been invoked and so our old transaction will not be finalized. For this problem, we need to cut block by some time interval. So if transaction won't occur in time then block would be cut ignoring the transaction in a block. So, one can cut the block by two methods 1) By number of transaction, 2) By Time. We can cut by size also but calculating it would be more tedious and these are sufficient so it's not chosen. Now the problem is if we want to cut block using time then we have to synchronise all the ordering service nodes which in distribute system is quite difficult. This happens because every node receives transaction at a different time and its possible that for different node timeout would be different and thus different block creation [10].

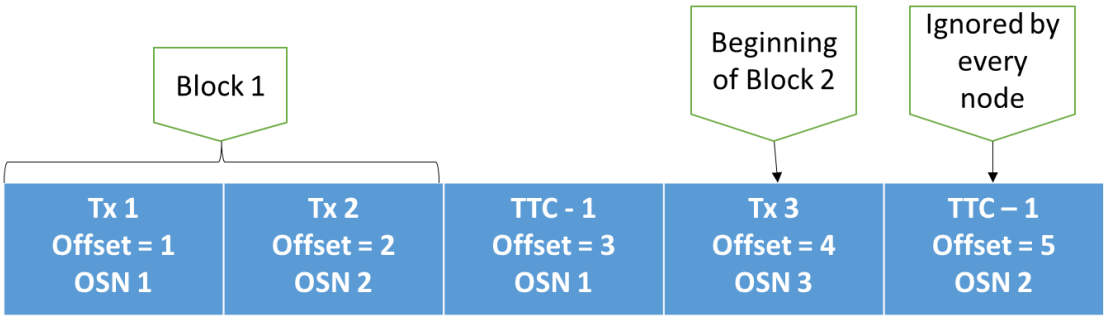

Figure 8: Effect of Time-to-cut-x block

As shown in figure 6, when a node reaches its time-out it will send a time-to-cut-X message for a block. After receiving that message every transaction on Kafka after that block will be accumulated in the next block and the previous transaction only will create the block. There is a possibility that there are many time-to-cut-X message possible on Kafka but message after first would be ignored. The next time to cut block that would be consider should have a number greater than X. Other aspect to consider is that when a transaction is added into the partition then the consumer only consider it as block creation. Meaning when the transaction reaches node just add it to the partition

without consideration of any aspect. Consumer routine only will maintain the creation of block and also TTC-X message.

After the block is created it will be delivered to gossip service. Then gossip service broadcast that block to every peer. After receiving the block every peer validates the transaction of the block and makes appropriate changes if it fulfils the criteria. Every successful and unsuccessful transaction masked accordingly and stored at every peer. After that, a transaction reaches its finality.

3.3.5 Contract Creation Guideline

This system can have a pluggable smart contract. Given that more time is provided for making this possible not all codes can be applied here. There are some guidelines for creating contracts. The guideline is as follow.

- Every contract will import contract package from the codebase.
- It needs to implement two methods 1) Init, 2) Invoke
- Init method will be called while initialization of Contract Script.
- Invoke method will be called whenever a transaction is being invoked.
- For any number of transaction, it is necessary that only Invoke method is a door by which any transaction can be called.
- Only String can be sent as an argument for any transaction invocation.
- The string can be masked in any form developer can use but the developer needs to consider that while creating APIs for the clients.
- For getting the data it needs to call the GetState and for modification, it needs to use PutState and for deletion of data, it needs to use DelState methods accordingly.
- After the execution of transaction, it will return the code which is 200 for successful execution or other than that for unsuccessful execution.
- Creating a secure contract script is as important as creating a secure network.
- The more info on the required methods to create a contract is given.
    - **GetState:** GetState(id string, txDetail *TransactionDetail) (docsQuery []map[string]interface{}, err error)

      As Select query from the database. It is used as a method of contractDetail object and takes id as string and txDetail for input. It returns the array of a map of string to interface object.

- **PutState:** PutState(newState []map[string]interface{}, txDetail *TransactionDetail)

  As Update query for the database. Creating new items will be done by this query only. It is used as a method of contractDetail object. It takes the array of the map of string to interface object array and txDetail as input.

- **DelState:** DelState(id string, txDetail *TransactionDetail) (err error)

  As Delete query of the database. Deletion of the item will be done by this query only. It is used as a method of contractDetail object. It takes the id as string and txDetail as input.

- **Init:** Init(detail *ContractDetail, txDetail *TransactionDetail) (code int)

  As shown above it takes two inputs *ContractDetail and *TransactionDetail. Detail stores the information regarding contract database and its id. txDetail stores the details regarding the stream by which query state is sent as transaction response.

- **Invoke:** Invoke(detail *ContractDetail, tx string, args []string, txDetail *TransactionDetail) (code int)

  Invoke takes 4 inputs two same as for Init and two other one being a string containing the name of the transaction and other being an array of a string containing the arguments for the transaction.

4 Results

At this stage, the system is deployed and is tested by example contract script. Contract script is pluggable and thus at the end of this project, a system is live for use. It is the developer's responsibility to build various application on this platform. However, Security and cryptography are not implemented as it can't work live on a public network. The system should be tested on the high number of peers with a high number of unrelated transaction and high number of related transactions as it would provide minimum and maximum transaction throughput that will commit. So at the end of the project, we get the system which can execute a pluggable contract script given it is created according to the norms of it.

5 CONCLUSION & RECOMMENDATION

This is not a closed end project. Because of it, there are many possibilities to extend this project. The aim of the project is to modify the consensus system for a large number of peers. This project satisfies it but lack of security measure and cryptography can't hold in real-world application. Because of same, this project can be referred to modification in Hyperledger Fabric or a base for the bigger project like Fabric. This Project is made available with MIT licence on GitHub for the sack of these opportunities like this. Here are some future work is given for the project.

- Add Transport level security.
- Add Authorization for peer joining the network.
- Introduce cryptography to ensure immutable ledger.
- Build and test this concept in Hyperledger Fabric.
- See the various implementation of such technology as a developer perspective.

6 REFERANCES

1. Nakamoto, S., 2008. Bitcoin: A peer-to-peer electronic cash system.
2. Androulaki, E., Barger, A., Bortnikov, V., Cachin, C., Christidis, K., De Caro, A., Enyeart, D., Ferris, C., Laventman, G., Manevich, Y. and Muralidharan, S., 2018, April. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference* (p. 30). ACM.
3. ZILLIQA Team, 2017. The ZILLIQA Technical Whitepaper. https://docs.zilliqa.com/whitepaper.pdf
4. Apache Software Foundation. “Kafka Documentation”, April 2019. https://kafka.apache.org/documentation/
5. Keval Patel. “Why should you learn Go?”., Medium, 08 January 2017, https://medium.com/@kevalpatel2106/why-should-you-learn-go-f607681fad65
6. Dale Hopkins, “Why we have decided to move our APIs to gRPC”, gRPC Blog, 29 August 2016, https://grpc.io/blog/vendastagrpc
7. Docker Inc., “Docker Documentation”,retrived on April 2019. https://docs.docker.com/
8. Anderson, J.C., Lehnardt, J. and Slater, N., 2010. *CouchDB: The Definitive Guide: Time to Relax*. " O'Reilly Media, Inc.".
9. Kreps, J., Narkhede, N. and Rao, J., 2011, June. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB* (pp. 1-7).
10. Kostas Christidis, “A Kafka-based Ordering Service for Fabric”, Retrived on April 2019, https://docs.google.com/document/d/19JihmW-8blTzN99lAubOfseLUZqdrB6sBR0HsRgCAnY/edit