Blockchain Consensus and Smart HBasechainDB
Project Interim Report

Bangarugiri Sateesh(18555) - II MTech(CS)
Supervisor: Dr. Pallav Kumar Baruah

September 2019

1 Introduction

Hyperledger Fabric: It is an open-source enterprise-grade permissioned distributed ledger technology(DLT) platform, designed for use in enterprise contexts, that delivers some key differentiating capabilities over other popular distributed ledger or blockchain platforms. The Fabric platform is also permissioned, meaning that, unlike with a public permission less network, the participants are known to each other, rather than anonymous and therefore fully untrusted.
In distributed ledger system, Consensus is the process of reaching an agreement on next set of transactions to be added to the ledger.In Hyperledger Fabric, consensus involves three distinct steps: i) Transaction Endorsement, ii) Ordering and iii) Validation and Commitment.
Consensus: A consensus algorithm is a process in computer science used to achieve agreement on a single data value among distributed processes or systems. Consensus algorithms are designed to achieve reliability in a network involving multiple unreliable nodes. Solving that issue – known as the consensus problem – is important in distributed computing and multi-agent systems.
The consensus in Hyperledger Fabric network is a process where the nodes in the network provide a guaranteed ordering of the transaction and validating those block of transactions that need to be committed to the ledger. Consensus must ensure the following in the network:

- Confirms the correctness of all transactions in a proposed block, according to endorsement and consensus policies.
- Agrees on order and correctness and hence on results of execution (implies agreement on global state).
- Interfaces and depends on smart-contract layer to verify correctness of an ordered set of transactions in a block.

Consensus must satisfy two properties to guarantee agreement among nodes: safety and liveness.
Safety means that each node is guaranteed the same sequence of inputs and results in the same output on each node. When the nodes receive an identical series of transactions, the same state changes will occur on each node. The algorithm must behave identical to a single node system that executes each transaction atomically one at a time.
Liveness means that each non-faulty node will eventually receive every submitted transaction, assuming that communication does not fail.
HBasechainDB is a super peer-to-peer network operating using a federation of nodes. All the nodes in the federation have equal privileges which gives HBasechainDB its decentralization. Such a super peer-to-peer network was inspired by the Internet Domain Name System. Any client can submit or retrieve transactions or blocks, but only the federation nodes can modify the blockchain. The federation nodes need not trust each other and yet operate to provide a tamper-proof data store.
Hbase is a wide column store database. It is a distributed, scalable, reliable, and versioned storage system capable of providing random read/write access in real-time. It provides a fault-tolerant way of storing large quantities of sparse data. HBase features compression, in-memory operation and Bloom filters on a per-column basis. We use the characteristics of HBase extensively to derive performance in HBasechainDB.

2 Problem Statement(s)

- To build a forking resilient Kafka ordering service by modifying existing one and ensuring safety guarantees.
- To make existing HBasechainDB support Smart Contract by modifying its architecture and check for flexibility.

3 Objectives

- To study the Bitcoin blockchain technology for zeroing on problems to work upon.
- To explore newer blockchain implementations – Ethereum and Hyperledger Fabric.
- To write smart contracts in the Ethereum framework and acquire an understanding of its use cases.
- To study approaches taken to addressing Byzantine Faults in blockchain.
- To explore consensus algorithms like Paxos, Raft, Kafka Orderer, Honey Badger.
- To study the HBasechainDB1.0 and HBasechainDB2.0 for understanding its architecture.
- To implement modified Kafka Orderer for a forking resilience in Hyperledger Fabric in GO language.
- To accommodate the changes in architecture of HBasechainDB2.0 for supporting Smart Contract and check for flexibility.

4 Work Done

4.1 June

- Studied Blockchain basics including Bitcoin technology.
- Explored Ethereum Blockchain system and its usecases.
- Finished Certification courses on Blockchain essentials and Building applications using Ethereum and IBM Hyperledger Fabric.

4.2 July

- Learnt about how to write a Smartcontract in Ethereum.
- Learnt Hyperledger Fabric blockchain system and its transaction flow.
- Setup Hyperledger Fabric network and ran some example Chaincodes on it.
- Explored Cosmos Network for interoperability of Blockchains.
- Studied Edge Computing for integrating with Blockchain and its usecases.

4.3 August

- Studied Consensus algorithms(Paxos, Raft, Kafka Orderer, Honey Badger).
- Studied Crash Failures and Byzantine Failures in Distributed Systems.
- Explored Byzantine Fault Tolerance concepts and its effects in Blockchain systems.
- Identified risk areas in Kafka Orderer for mitigation.

4.4 September

- Identified Fork situation in Kafka Orderer in Hyperledger Fabric and modified its architecture for avoiding forking problem.
- Studied HBasechainDB architecture and its transaction flow.
- Written HiPC(High Performance Computing) Paper on **"Forking Resilience in Kafka Orderer"** for IEEE Student Research Symposium.

5 Literature Survey

5.1 Hyperledger Fabric[1][2]

Fabric is a distributed operating system for permissioned block-chains that executes distributed applications written in general-purpose programming languages (e.g., Go, Java, Node.js). It securely tracks its execution history in an append-only replicated ledger data structure and has no cryptocurrency built in.

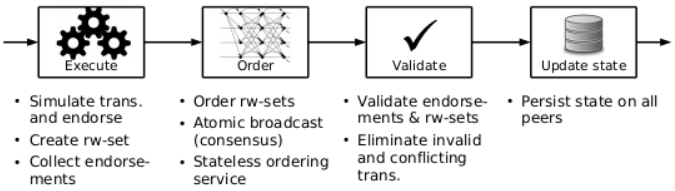

Figure 1: **Execute-order-validate architecture of Fabric(rw-set means a readset and writeset)**

Fabric introduces the execute-order-validate blockchain architecture (illustrated in Fig. 1) and does not follow the standard order-execute design, In a nutshell, a distributed application for Fabric consists of two parts:

- A smart contract, called chaincode, which is program code that implements the application logic and runs during the *execution phase*. The *chaincode* is the central part of a distributed application in Fabric and may be written by an un-trusted developer. Special chaincodes exist for managing the blockchain system and maintaining parameters, collectively called system chaincodes.
- An *endorsement policy* that is evaluated in the validation phase. Endorsement policies cannot be chosen or modified by untrusted application developers. An endorsement policy acts as a static library for transaction validation in Fabric, which can merely be parameterized by the chaincode. Only designated administrators may have a permission to modify endorsement policies through system management functions. A typical endorsement policy lets the chaincode specify the endorsers for a transaction in the form of a set of peers that are necessary for endorsement; it uses a monotone logical expression on sets, such as "three out of five" or "$(A \wedge B) \vee C$".

A client sends transactions to the peers specified by the endorsement policy. Each transaction is then executed by specific peers and its output is recorded; this step is also called endorsement. After execution, transactions enter the ordering phase, which uses a pluggable consensus protocol to produce a totally ordered sequence of endorsed transactions grouped in blocks. These are broadcast to all peers, with the (optional) help of gossip. Unlike standard active replication [48], which totally orders transaction inputs, Fabric orders transaction outputs combined with state dependencies, as computed during the execution phase. Each peer then validates the state changes from endorsed transactions with respect to the endorsement policy and the consistency of the execution in the validation phase. All peers validate the transactions in the same order and validation is deterministic.

5.2 HBasechainDB [3]

HBasechainDB is a super *peer-to-peer network* operating using a federation of nodes. All the nodes in the federation have equal privileges which gives HBasechainDB its decentralization. Such a super peer-to-peer

network was inspired by the Internet Domain Name System. Any client can submit or retrieve transactions or blocks, but only the *federation nodes* can modify the blockchain. The federation can grow or shrink during the course of operation of HBasechainDB. Let us say there are n federation nodes N1, N2, ..., Nn. When a client submits a transaction t, it is assigned to one of the federation nodes, say Nk. The node Nk is now responsible for entering this transac- tion into the blockchain. Nk first checks for the validity of the transaction. Validity of a transaction includes having a correct transaction hash, correct signatures, existence of the inputs to the transaction, if any, and the inputs not having been already spent. Once Nk has validated a set of transactions, it bundles them together in a block, and adds it to the blockchain. Any block can only contain a specified maximum number of transactions. Let us say t was added in the block B.
When the block B is added to the blockchain its validity is undecided. Since the federation is allowed to grow or shrink during the operation of HBasechainDB, blocks also include a list of voters based on the current federation. All the nodes in the voter list for a block vote upon B for its validity. For voting upon a block, a node validates all the transactions in the block. A block is voted valid only if all the transactions are found to be valid, else it is voted invalid. If a block gathers a majority of valid or invalid votes, its validity changes from undecided to valid or invalid respectively. Only the transactions in a valid block are considered to have been recorded in the blockchain. The ones in the invalid blocks are ignored altogether. However, the chain retains both valid and invalid blocks. A block being invalid does not imply that all the transactions in the block are invalid. Therefore, the transactions from an invalid block are re-assigned to federation nodes to give the transactions further chance of inclusion in the blockchain. The reassignment is done randomly. This way, if a particular rogue node was trying to add an invalid transaction to the blockchain, this transaction will likely be assigned to a different node the second time and dropped from consideration. Thus, if block B acquires a majority of valid votes, then transaction t would have been irreversibly added to the blockchain. On the other hand, if B were invalid, then t would be reassigned to another node and so on until it is included in the chain or removed from the system.
When a block is entered into *hbasechain table*, the blocks are stored in HBase in the lexicographical order of their ids. The chain is actually formed during vote time. When a node votes on a block, it also specifies the previous block that it had voted upon. Thus, instead of waiting for all the federation nodes to validate the current block before proceeding to the creation of a new block, blocks are created independent of validation. This is the technique of blockchain pipe-lining described earlier. Over time,the blockchain accumulates a mix of valid and invalid blocks. The invalid blocks are not deleted from the chain to keep the chain immutable. What we also note here is that while it would seem that different nodes could have a different view of the chain depending upon the order in which they view the incoming blocks, it is not seen in practice in HBasechainDB due to the strong consistency of HBase and the fact that the blocks to be voted upon are ordered based on their timestamp. Thus, each node sees the same order of blocks, and we have the same chain view for different nodes.
To tamper with any block in the blockchain, an adversary will have to modify the block, leading to a change in its hash. This changed hash would not match the vote information for the block in the votes table, and also in subsequent votes that refer to this block as the previous block. Thus an adversary would have to modify the vote information all the way up to the present. However, we require that all the votes being appended by nodes are signed. Thus, unless an adversary can forge a node's signature, which is cryptographically hard, he cannot modify the node's votes. In fact, he has to forge multiple signatures to affect any change in the blockchain preventing any chances of tampering. This way HBasechainDB provides a tamper-proof blockchain over HBase.

6 Experiments

- Installed **Bitcoin core** application for understanding its test network.
- Installed **Ganache** which is an ethereum client which one can use for Ethereum development.
- Installed **MetaMask** which allows you to run Ethereum dApps right in your browser without running a full Ethereum node.
- Written a few basic **Smartcontracts** like ballotVoting,Crowdfunding etc. for Ethereum in Solidity language with help of Ethereum Wallet, MetaMask and Ganache.
- Successfully setup **Hyperledger Fabric** network and ran few chaincodes on it.

- Successfully ran **Solidity** code on Hyperledger Fabric network.
- Installed **Cosmos Network** for interoperability of Blockchains.
- Setup **EdgeX Foundry** open-source Edge Computing Platform and tried for integrating it with Blockchain.
- Created **Docker container** replicas for simulating distributed environment.

7 Forking Resilience in Kafka Ordering Service

Blockchain Fork: A fork occurs on the Bitcoin blockchain when the exact block at some blockheight 'h' is different at different nodes. Typically occurs when two or more miners find blocks at nearly the same time. It can also happen as part of an attack. The block height of a particular block is defined as the number of blocks preceding it in the blockchain. A blockchain contains a series of blocks, hence the name blockchain. These blocks are essentially data units which are used to store transactional information of the network.
The very first block on a blockchain is called the genesis block. It has a block height of zero, as no blocks precede it in the blockchain. The total height of the blockchain is taken to be the height of the most recent block, or the highest block, in the chain.
Arbitrary node failures may occur in distributed environment in which case node responds with incorrect results or deliberately misleading results or a different result to different parts of the system. Blockchains use a model that consider every byzantine failure, which is known as the **Byzantine Failure Model**. This is the hardest known failure model to utilize in a distributed system. The fail-stop failures are node crashes which can be detected by other nodes.
The Hyperledger Fabric architecture lets users choose an orderer service that implements a consensus algorithm that fits their application. One desirable property is **Byzantine fault tolerance (BFT)**, which says the orderer can do its job even in the presence of malicious actors. It's important to note that BFT, however it's implemented, only applies to the ordering of transactions in Hyperledger Fabric. Its job is to ensure that every peer has the same list of transactions on its ledger.
In Fabric, the Ordering Service is the sub-system that is tasked with putting together a block of transactions. Then the ordering service transmits the block to all nodes. The ordering service in Hyperledger Fabric is a pluggable module, meaning you can use any existing consensus protocol. The ones currently being used **(Solo, Kafka (CFT) and Raft)** achieve consensus deterministically.
In Hyperledger Fabric, the Kafka ordering service consists of a cluster of **Ordering Service Nodes(OSNs)** used for transaction ordering deterministically. Transactions are received by any of the OSN nodes which will be posted to a Kafka partition. The transactions are consumed by all OSNs including the one which posted it into the Kafka cluster. Each OSN consumes all transactions from the Kafka partition and stored into their local ledgers. A block is cut as soon as one of three conditions is met: (1) the block contains the specified maximal number of transactions; (2) the block has reached a maximal size (in bytes); or (3) an amount of time has elapsed since the first transaction of a new block was received. So every OSN creates the block from the transactions stored in the OSNs local logs.
The above mentioned process is currently happening in the Kafka ordering service. We know that Kafka orderer is not a Byzantine Fault Tolerant(BFT) so byzantine attacks can occur. Consider a scenario where transaction stream sent to the OSN was tampered by a byzantine attacker on Kafka. One more scenario where block signature stream sent to OSN was tampered by a byzantine attacker on Kafka. This kind of scenarios will be addressed and safety guarantees will be assured by our model. Finally, this model makes the kafka ordering fork resilient. Forking means blocks mismatch at Ordering service nodes.
Forking situation in Hyperledger Fabric as shown in Fig 1.

Suppose block C contains two transactions, and block C' contains those same two, but in opposite order and account A has a balance of $10.

- In transaction1, it says "transfer $10 from A to B"
- In transaction2, it says "transfer $10 from A to C"

If in block C, the order is transaction1, transaction2, and in block C' the order is transaction2, transaction1, then in block C, account B ends up with the money and account C gets nothing, while in block C', account

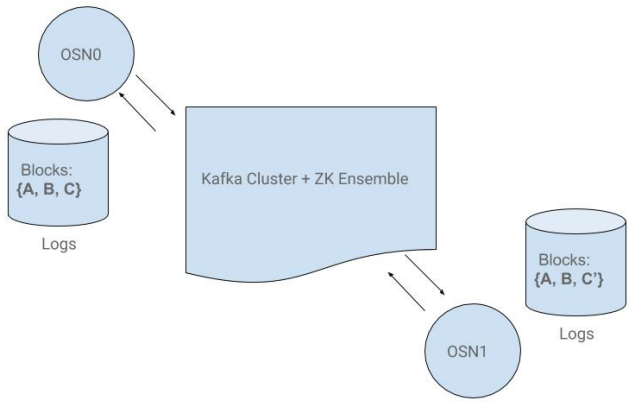

Figure 2: Forking in Hyperledger Fabric

C gets the money and account B gets nothing. This is commonly known as a "Double Spend", where transaction1 spends the money, transaction2 spends it again. Only one can be valid, and it will be the first one which is processed.
So, if the blockchain is forked, it's possible to convince two parties that they have each been paid $10, when there is only $10 initially in the account.

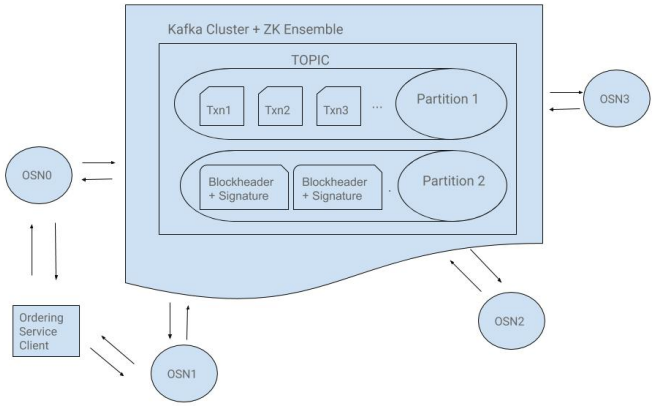

Figure 3: Modified Kafka Orderer with the second partition in a Kafka Topic

To avoid forking situation in the Kafka orderer, we perform the following steps:

- Create a second partition in the Kafka topic for each channel apart from the first partition from where the OSNs consume the transactions.
- Have each OSN produce its signature over the block header, but before committing it, produce the signature and block header into the second partition.
- Have each OSN wait on this other partition for a 'sufficient' number of matching signatures.
- Each OSN commit their block into their local logs after getting a sufficient number of matching signatures from the second partition and all these signatures must be added into block metadata.

Today, orderers/OSNs do not share blocks with each other, they assume that the blocks are the same because they were generated deterministically(all OSNs take the transactions from the same partition). The modification we propose, is to have a second round through Kafka where they share block header hashes and

signatures, and require a certain number of them before committing.

The overall process of the proposed model as follows:

- Each Orderer/OSN generate a block and its header by invoking 'CreateBlock' function and sign it.
- It then pushes the blockheader+signature to the second partition as shown in Fig 2.
- Because every orderer does this, eventually, the orderer will consume enough block headers + signatures, and may combine those signatures before invoking 'WriteBlock' function.
- Each OSN would check for matching signatures and if majority matches then it will commit the block.

With today's Kafka Fabric Ordering Service, each block only has one signature. But, the field in the block metadata for signatures is arbitrary in length. This was always part of the design to support BFT protocols in the future.
So, instead of only having one signature, the OSN would collect 'a majority' of signatures, before committing the block. Because an orderer only ever signs once, it's impossible to get two sets of 'majority' signatures for two different blocks. Hence, the fork cannot occur. So, the stated goal of 'preventing a fork' is achieved and the method also provides the safety guarantees.

8 Smart HBasechainDB

Smart contract is a facility which allows the blockchain application developer to define his own transaction validation protocol. This brings a great *flexibility* to the blockchain users. The current implementation of HBasechainDB does not support smart. HBasechainDB can be rearchitectured to support smart contract which will get a whole lot of flexibility to the users.

9 Future Work

- Installing HBase on 8-node Hadoop Cluster and HBasechainDB.
- To implement modified Kafka Orderer for a forking resilience in Hyperledger Fabric in GO language and check for performance.
- To accommodate the changes in architecture of HBasechainDB2.0 for supporting Smart Contract and check for flexibility.

References

[1] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolic, Sharon Weed Cocco, and Jason Yellick. **"Hyperledger fabric: a distributed operating system for permissioned blockchains"**. In EuroSys, pages 30:1–30:15. ACM, 2018.

[2] Christian Gorenflo, Stephen Lee, Lukasz Golab, and S. Keshav. **"FastFabric: Scaling Hyperledger Fabric to 20,000 Transactions per Second"**. IEEE International Conference on Blockchain and Cryptocurrency (ICBC), pages 455–463, 2019.

[3] Sahoo, M.S.; Baruah, P.K. **HBasechainDB—A Scalable Blockchain Framework on Hadoop Ecosystem**. In Supercomputing Frontiers; Springer International Publishing: Cham, Switzerland, 2018; pp. 18–29.

[4] A. Baliga, N. Solanki, S. Verekar, A. Pednekar, P. Kamat, and S. Chatterjee, "**Performance Characterization of Hyperledger Fabric"**, in Crypto Valley Conference on Blockchain Technology (CVCBT), 2018.

[5] D. Ongaro and J. Ousterhout, **"In search of an understandable consensus algorithm"**, in Proc. USENIX Conf. USENIX Annu. Tech. Conf. (USENIX ATC), Berkeley, CA, USA, 2014, pp. 305–320. [Online].

[6] Lamport, L. **"Paxos made simple"**. ACM SIGACT News 32, 4 (Dec. 2001), 18–25.

[7] Avi Asayag, Gad Cohen, Ido Grayevsky, Maya Leshkowitz,Ori Rottenstreich, Ronen Tamari and David Yakira. **"Helix: A Scalable and Fair Consensus Algorithm"**, Orbs Research (orbs.com)

[8] Leslie B. Lamport; Shostak, R.; Pease, M. (1982)."**The Byzantine Generals Problem"**, ACM Transactions on Programming Languages and Systems.

[9] Allen Clement, Edmund Wong, Lorenzo Alvisi, Mike Dahlin, **"Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults"**, NSDI 2009: 6th USENIX Symposium on Networked Systems Design and Implementation.

[10] Satoshi Nakamoto, **"Bitcoin: A Peer-to-Peer Electronic Cash System"**, www.bitcoin.org, 2008.

Bangarugiri Sateesh
(Student)

Dr. Pallav Kumar Baruah
(Supervisor)

Dr. R. Raghunatha Sarma
(Associate Head, DMACS)