Adding Forking Resilience to Hyperledger Fabric Kafka Based Ordering, even in the presence of Byzantine Faults

Bangarugiri Sateesh*†, Pallav Kumar Baruah†, Jason Yellick‡
†Sri Sathya Sai Institute of Higher Learning, Prasanthi Nilayam, India
‡IBM, Raleigh-Durham, North Carolina Area, USA
satti.14u@gmail.com, pkbaruah@sssihl.edu.in

***Abstract*—Hyperledger Fabric is a permissioned ledger platform delivering confidentiality, privacy and scalability to enterprise blockchains. Fabric uses Kafka ordering service as a default ordering service in the production environment. Apache Kafka[1] is a crash fault tolerant(CFT) implementation that uses a leader-follower node configuration. Kafka uses the Zookeeper ensemble for management purposes. Kafka orderer is not Byzantine Fault Tolerant(BFT)[2] so it is susceptible to attack from malicious actors which may halt the consensus.**

We propose a novel model to prevent the Byzantine attacker with the control of a Kafka cluster from being able to cause a blockchain fork. We cannot make a Kafka based orderer tolerant of all byzantine faults, namely, we cannot ensure liveness. However, we can ensure non-forking or correctness in the presence of byzantine faults. This solution adds some degree of BFT resilience.

***Index Terms*—Blockchain, Hyperledger Fabric, BFT, CFT, Apache Kafka, Zookeeper, Blockchain Fork, Kafka Orderer, Blockchain Consensus**

I. Introduction

Hyperledger Fabric[5]: 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 permissionless 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[4]: 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.

Apache Kafka[1][14]: It is a distributed publish-subscribe messaging system that was developed at LinkedIn and later on became a part of the Apache project. It is fast, scalable, agile and distributed by design. In Apache Kafka, a Topic means a stream of messages belonging to a particular category. Producer means any application that can publish messages to a topic. Consumer means any application that can subscribe to topics and consume messages. Kafka cluster is a set of servers, each of which is called a broker. Kafka Architecture is shown in Figure 1.

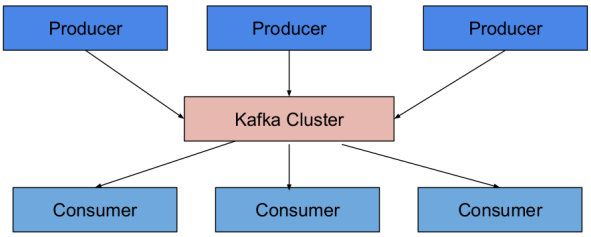

Fig. 1. Apache Kafka Architecture

Ordering Service[5]: Blockchains[3], such as Ethereum and Bitcoin, which are not permissioned which means any node can participate in the consensus process, wherein transactions are ordered and bundled into blocks. So, these systems rely on probabilistic consensus algorithms which eventually guarantee

ledger consistency to a high degree of probability, but which are still vulnerable to divergent ledgers(also known as ledger fork) where different participants have a different view of the accepted order of transactions. Hyperledger Fabric works in a different manner. It features a kind of node called orderer which does this transaction ordering, which along with other nodes forms an ordering service. Because Fabrics design relies on deterministic consensus algorithms, any block a peer validates as generated by the ordering service is guaranteed to be final and correct. Ledgers cannot fork the way they do in other distributed blockchains.

Kafka Ordering Service[5]: The crash fault tolerant ordering service is supported by Hyperledger Fabric by adapting Kafka distributed streaming platform for use as a cluster of ordering nodes. Kafka uses the leader and follower node configuration in which transactions are replicated from leadernode to the follower nodes. In the case the leader goes down, one of the follower nodes becomes leader and ordering can continue, ensuring fault tolerance.

Blockchain Fork[6][7]: A fork occurs on the Bitcoin blockchain when the exact block at some blockheight[8] '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[9]. 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.

Fork is not a blockchain specific term; In software engineering[10], a project fork happens when developers take a copy of source code from one software package and start independent development on it, creating a distinct and separate piece of software. At its most basic, a fork is what happens when a blockchain diverges into two potential paths forward either with regard to a networks transaction history or a new rule in deciding what makes a transaction valid. But forks also can be willingly introduced to the network. This occurs when developers seek to change the rules the software uses to decide whether a transaction is valid or not. Forking in each and every blockchain is different, based on the design architecture.

Fork can happen in following situations[11]:

- Anytime two miners find a block at nearly the same time.
- Developers seek to change the rules the software uses to decide whether a transaction is valid or not.

II. Background

Arbitrary node failures may occur 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[12], 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. In a distributed computing systems, Byzantine failures are defined as arbitrary deviations of a process from its assumed behavior based on the algorithm it is supposed to be running and the inputs it receives. The term takes its name from an allegory, the Byzantine General Problem[13], developed to describe a situation in which, in order to avoid catastrophic failure of the system, the system's actors must agree on a concerted strategy, but some of these actors are unreliable.

Byzantine Faults[18]: Node/Component behaves arbitrarily in which case it acts in one of the following ways.

- It might perform incorrect computation.
- It might give conflicting information to different parts of the system.
- It might collude with other failed nodes.

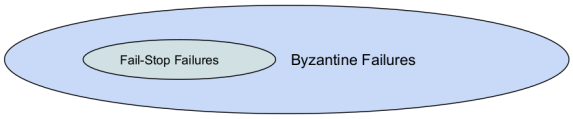

Fig. 2. Failure Modes in Distributed System [15]

Node fails arbitrarily due to one of the following reasons:

- Software bug present in code.
- Hardware failure occurs.
- Hack attack on the system.

The Hyperledger Fabric architecture [16] 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.

Its important to note that BFT, however its 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.

III. Proposed Model

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 PBFT) 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 ledger. 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.

In Figure 3, It is shown the forking problem where the OSN0 and OSN1 have two different blocks C and C'(C != C'). Suppose block C contains two transactions, and block C'

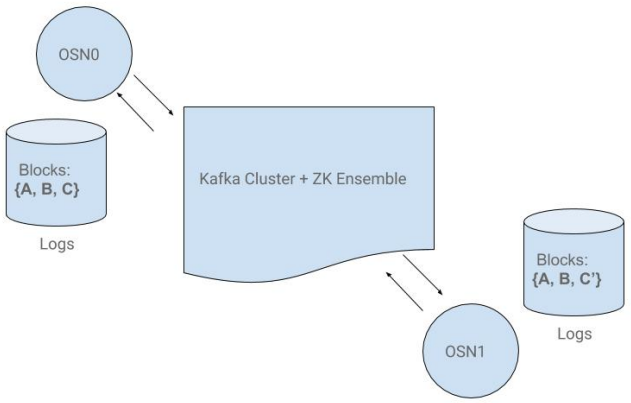

Fig. 3. Forking in Hyperledger Fabric

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 C gets the money and account B gets nothing. This is commonly known as a "Double Spend[17]", 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.

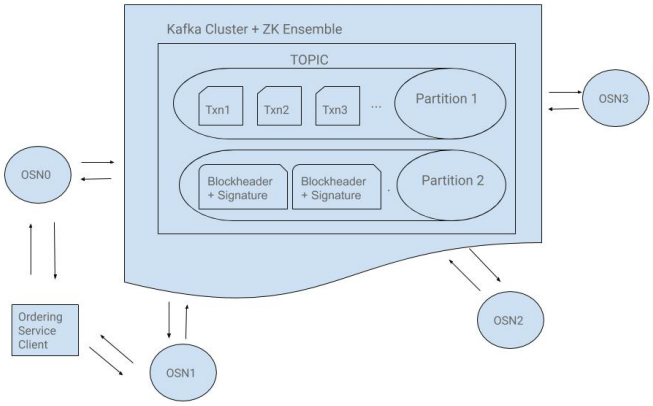

Fig. 4. Modified Kafka Orderer with the second partition in a Kafka Topic

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.Block contents are shown in the Fig 5.

'DataHash' is a hash over the transactions in the Data section of the current block and 'PrevHash' is the hash of the previous block header.

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 4.
- Because every orderer does this, eventually, the orderer will consume enough block headers + signatures, and may combine those signatures before invoking Write-Block 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

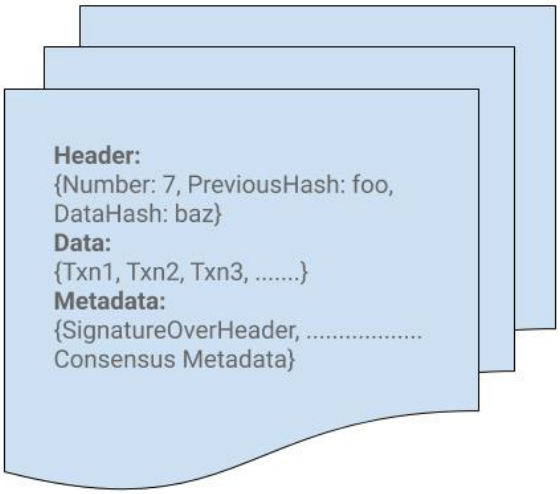

Fig. 5. Block Contents in Hyperledger Fabric

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.

IV. Conclusion

Kafka is not a BFT. So, a byzantine attacker with the appropriate privileges may cause Kafka to return arbitrary streams to the OSNs. This approach still prevents the OSNs from forking, though it obviously does nothing for liveness.
In a permissioned blockchain, actors have significant disincentive to behave in a byzantine manner. This is because the nodes are permissioned, malicious behavior may generally be attributed to the guilty party. So, in these systems, full byzantine fault tolerance is not necessarily required. Full BFT systems, like PBFT have significant performance overhead, and are difficult to administer and debug. By using a CFT system, and layering on only BFT safety (but not liveness), we can retain some of the performance characteristics and simplicity in administration of the CFT system, while eliminating the ability of malicious actors to exploit a window of system incorrectness before they are caught.
Are any actors behave maliciously in permissioned blockchain? Is it possible that to happen? Yes, It is of course possible to happen, but, it makes certain attacks, like an attack on liveliness less practical. If you have four companies trading goods on a blockchain, if one company does something malicious which stops trading (ie, causing a failure in liveliness), what benefit has that company achieved? It has hurt their own ability to conduct business, and, the other companies will likely pursue litigation against them. It does them very little good.
Contrast this if that company were able to trick the other companies into paying them multiple times, or in some way to be able to cause duplicate or fraudulent transactions to be processed. Then that company might be able to flee with their ill gotten gains. This is the difference between liveness and safety.
So missing liveness simply means it's possible for a bad actor to stop processing. While missing safety means that it's possible for a bad actor to cause incorrect processing.

Acknowledgement

We dedicate this work to Bhagawan Sri Sathya Sai Baba, the Founder Chancellor of Sri Sathya Sai Institute of Higher Learning.We thank the Department of Mathematics And Computer Science for providing the required facilities to work on this project.

References

1 Jay Kreps, Neha Narkheda, Jun Rao, Kafka: a Distributed Messaging System for Log Processing, NetDB workshop 2011
2 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.
3 Satoshi Nakamoto, Bitcoin: A Peer-to-Peer Electronic Cash System, www.bitcoin.org, 2008.
4 https://www.skcript.com/svr/consensus-hyperledger-fabric/consensus-in-hyperledger-fabric
5 Hyperledger Fabric Docs. https://hyperledger-fabric.readthedocs.io
6 Fork(blockchain): https://en.wikipedia.org
7 Does Blockchain Fork Happen in Fabric. https://blockchain-backyard.github.io/community-questions/posts/does-blockchain-fork-happen-in-fabric/
8 https://lightrains.com/blogs/what-is-meant-by-forking-blockchain
9 https://en.bitcoin.it/wiki/Genesis-block
10 https://en.wikipedia.org/wiki/Fork-(software-development)
11 https://lightrains.com/blogs/what-is-meant-by-forking-blockchain
12 Fault Tolerant Computing in Industrial Automation Hubert Kirrmann ABB Research Center CH-5405 Baden, Switzerland:
13 Leslie B. Lamport; Shostak, R.; Pease, M. (1982)."The Byzantine Generals Problem", ACM Transactions on Programming Languages and Systems.
14 Apache Kafka. https://www.edureka.co/blog/apache-kafka-next-generation-distributed-messaging-system
15 Byzantine Failures: https://www.geeksforgeeks.org/practical-byzantine-fault-tolerancepbft/
16 https://medium.com/kokster/understanding-hyperledger-fabric-byzantine-fault-tolerance-cf106146ef43
17 https://www.coindesk.com/the-double-spend-what-bitcoins-white-paper-solved-forever
18 https://www.cs.princeton.edu/courses/archive/fall16/cos418/docs/L9-bft.pdf
19 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 30:130:15.ACM, 2018.
20 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)