International Journal of Advanced Science and Technology
Vol. 29, No. 7s, (2020), pp. 3549-3554

Comparison of Performance Between Kafka and Raft as Ordering Service Nodes Implementation in Hyperledger Fabric

*H. Yusuf[1,2] and I Surjandari[1,3]

[1]*Industrial Engineering Department, Universitas Indonesia, Depok 16424, Indonesia*
[2]*harman.yusuf@ui.ac.id*, [3]*isti@ie.ui.ac.id*

Abstract

One of well-known Hyperledger platform named Hyperledger Fabric, which was first developed by IBM and Digital Asset, is a blockchain framework with execute-order-validate architecture. With this architecture, transactions on Hyperledger Fabric become deterministic. One of the nodes that plays an important role in the process is ordering service node. It plays a role in maintaining channel configuration and create a block of transaction. In its implementation, Hyperledger Fabric consists of three ordering service node types, namely Solo, Kafka, and Raft. Solo does not have a consensus algorithm and is only used during the development of blockchain networks while Kafka and Raft have (crash fault tolerant). This paper simulates Blockchain Network with Kafka and Raft as implementation of the ordering service node. From the research obtained, it is known that Raft is more superior than Kafka in terms of success rate and throughput rate when conducting invoke transactions due to its simpler framework (Raft has the Raft consenter which was created directly from ordering service nodes) than Kafka (which requires Kafka brokers and Zookeeper Ensembled). However, in the process of querying the transaction, Kafka is superior to Raft since the throughput rate is bigger than Raft.

Keywords: *Blockchain, Hyperledger Fabric, Multiple Channels, Ordering Service, Raft, Kafka*

1. Introduction

Blockchain is a new distributed ledger technology. The first application of the Blockchain was started from bitcoin that was created by Satoshi Nakamoto [1]. As time goes by, several companies began to develop a new Blockchain network (more precisely Permissioned Blockchain) for the enterprise case because many researchers and entrepreneurs realize that the ability of the Blockchain both in privacy and transparency can be explored even deeper.

One of the big companies that started developing the Blockchain platform was Linux Foundation which created Hyperledger (an umbrella project for the Blockchain platform). Hyperledger is a large project that has been followed by more than 200 large companies [2]. One well-known Hyperledger platform is Hyperledger Fabric which initially contributed by IBM and Digital Asset [3]. This platform has transaction capabilities of up to 3500 transactions per second with a minimum latency of around sub seconds [4].

In a bigger picture, Hyperledger Fabric has three processes in transaction flow, which are order, execute, and validate. At the order stage, the client will provide a transaction along with a proposal to each endorsing peer. After endorsing peers accept it, the endorsing peers will provide the transaction along with its endorsed signature (endorsed transaction) to the client. Then, at the order stage, endorsed transactions from the client are given to the existing ordering service to be collected and arrange into a specific block.

[*] Corresponding Author

Finally, at the validate stage, the ordering service will provide a block to all peers in the channel [5].

In its implementation, Hyperledger Fabric ordering service node consists of three types, namely solo, Kafka, and Raft. Solo does not have a consensus algorithm and is only used during the development of Blockchain networks while Kafka and Raft have (crash fault tolerant) [6]. This paper will examine the capabilities of Kafka and Raft as implementation of the ordering service node. Although Kafka and Raft use the same consensus algorithm, the architectural forms of Kafka and Raft are different so that the two ordering service node implementations must have different performance from success rate to throughput rate.

2. Methodology

2.1. Ordering Service Nodes

Ordering service node is one of the nodes that plays an important role in maintaining channel configuration and carrying out the transaction process. In channel configuration, the ordering service nodes controls the basic access of the channel (through the consortium of the configuration file). The ordering service nodes also limits which nodes are able to read and write data to them in accordance with previously made consortium [6].

Ordering services also have several functions in each transaction flow process. In the Order phase, the ordering service node will receive endorsed transactions from clients (transactions that have been endorsed by endorsed peers). After the process of receiving a transaction reach batchSize (the limit of the number of transactions that can be received per batch) or batchTimeout (the time limit for receiving transactions per batch) [7], ordering service nodes will arrange transactions in batches into a strict order and develop the batch of transactions into a block. Because transactions that are in block are in the form of strict ordering, all successful transactions that are validated will not be discarded (will not become ledger forks). Last, in the validate phase, the order will distribute blocks to all peers connected in the channel (according to channel configuration) [6].

In its implementation, Ordering Service is divided into three types, namely Solo, Kafka, and Raft.

2.2. Solo

Solo is an implementation of ordering service only to test the Blockchain created. Solo works without using a consensus mechanism and only consists of one ordering node [6].

2.3. Kafka

Kafka is an ordering service implementation based on the Crash Fault Tolerant (CFT) mechanism, where in this mechanism, the process will continue even when the some of the existing nodes experiences N failures while N / 2 + 1 nodes can still run [7]. This implementation uses "leader and follower" in the configuration node and managed using Zookeeper Ensembled. However, the process of finding offset numbers is straight from the ordering service node (because the Ordering Service Node has been configured to be able to maintain local logs) and not through Kafka partitions as usual. Because of this process, it is unlikely that duplications of the block will occur, but the process is slower than directly from Kafka [6][8][9].

2.4. Raft

Raft is actually similar to Kafka because the ordering service implementation also uses CFT. To run the implementation of the "leader and follower" Raft uses the Raft consenter which is actually ordering service nodes itself. Raft is also a system used by Hyperledger Fabric as a connecting bridge to design a consensus of Practical Byzantine Fault Tolerant because there are similarities in integrating the consensus with Hyperledger Fabric [6][10].

3. Comparison Between Raft and Kafka in Hyperledger Fabric

3.1. General Comparison

Although using the same consensual mechanism, the application of the two ordering service node implementations has a quite clear difference.

First, Kafka and Zookeeper were initially set up not for large networks, so that even though there are many organizations and channels on the Blockchain network, the process is like one organization (not too decentralized). Different from Raft which uses ordering service nodes directly as a replication state machine (Raft Consenter) so that each organization can have its own ordering services node and make the Blockchain network more decentralized [6].

Second, Kafka must need docker images for the CFT process to work because Kafka was built by Apache. This is quite complicated and its use must also be further studied. Raft on the other hand, developed natively in the Hyperledger Fabric itself so that its use is easier [6].

3.2. Comparison in Terms of Configuration

Overall, Raft is easier to set up than Kafka because the Raft is built directly from the ordering services node while Kafka must make the Kafka brokers and Zookeeper Ensembled so that the CFT process can run. However, Raft is more complicated than Kafka when making channel configuration only because Raft has to set up a client and server transport layer security certificate compared to Kafka which only needs to determine the number of Kafka brokers and Zookeepers.

Then, when forming the docker container, Raft only needs to form an ordered service node that has been configured according to the previous network configuration. Different from Kafka which must form separate Kafka and Zookeeper containers and must determine the number of Kafka and Zookeeper to be formed into their containers in the compose docker file. Because of that, the process of forming CFT through Kafka becomes trickier.

4. Performance Analysis

4.1. Simulation Configuration

In this paper, Kafka and Raft ordering service implementation will be tested to see which performance is better. The Blockchain network consists of three different channels, each channel has two organizations, and each organization has two peer nodes. What distinguishes Kafka from the configuration made is the Raft consists of five ordering service nodes according to default, while Kafka consists of three ordering service nodes which each ordering service nodes is set for one particular channel. The chaincode used for this process is simple chaincode. Because this paper focused on the ability of the Raft and Kafka, all the timeout of the chaincode is changed to 100 seconds. The performance test for Kafka and Raft on the Hyperledger Fabric is carried out using a Hyperledger caliper that is specifically designed to test the ability of the Blockchain network transactions. The configuration diagram of blockchain network can be seen at figure 1 and 2.

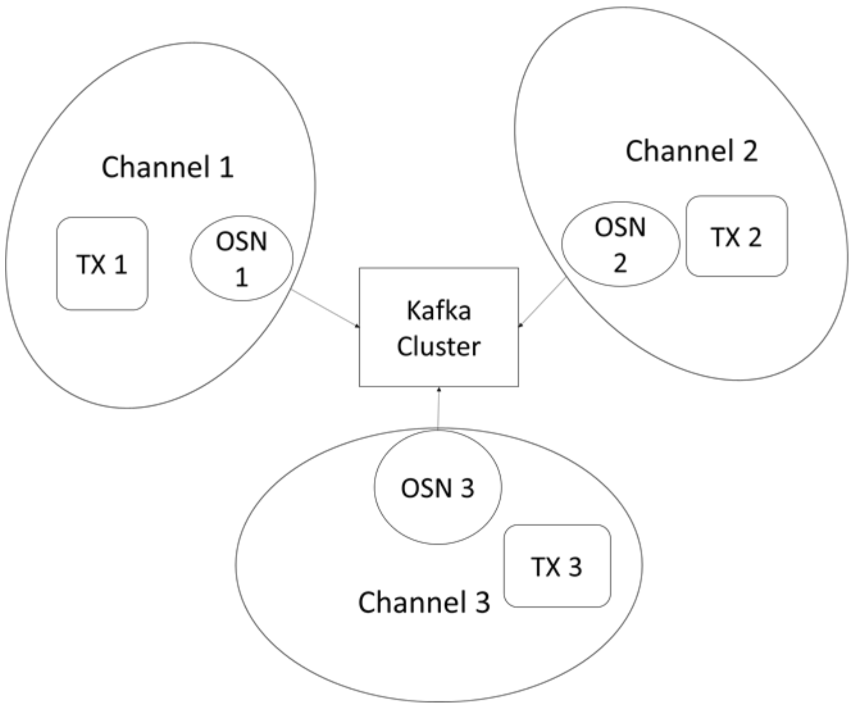

Figure 1. Simulation Configuration Framework with Kafka

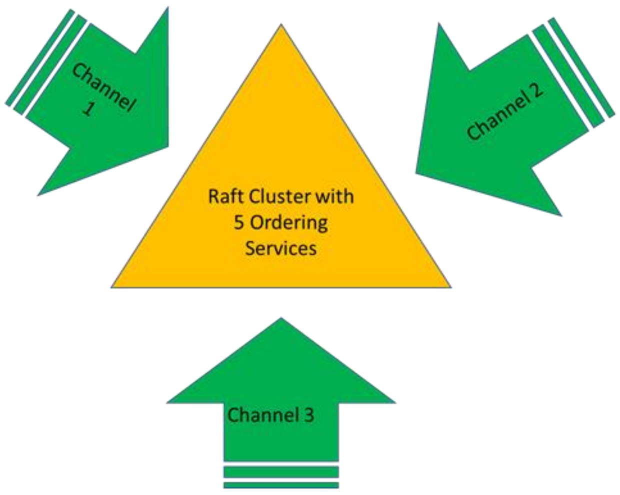

Figure 2. Simulation Configuration Framework with Raft

4.1. Simulation Result

In the simulation results of the create an account process, Raft has bigger possibility of success in making transactions compared to Kafka. This can be seen from table 1 that the number of rounds that have failed in a transaction and the minimum total number of successful transactions. Of the number of rounds that failed, Raft only failed two rounds of 12 rounds while Kafka had four failed rounds of 12 rounds. Furthermore, from the minimum total success (which means there was a failure in the round), Raft managed to do 954 transactions out of 1000 transactions while Kafka succeeded in doing 548 transactions out of 1000 transactions.

In addition, Raft also has a higher throughput compared to Kafka in this process. It can be seen in table 1 that both maximum throughput, minimum throughput and average throughput whether there are transaction failures in the process or not which make Raft faster than Kafka.

Table 1. Simulation Result of Create an Account Process

Type	Maximum Success	Minimum Success	Total Failure (round)	Maximum Throughput (TPS)	Minimum Throughput without Failed (TPS)	Minimum Throughput with Failed (TPS)	Minimum Throughput without Failed (TPS)	Minimum Throughput with Failed (TPS)
Raft	1000	954	2	27.3	21.6	7.3	24.05	21.33333
Kafka	1000	548	4	22.9	19.8	4.2	16.71	17.48333

Then, in a query an account on table 2, Raft and Kafka have the same success probabilities (100%). However, for the speed of the throughput process in the query (checking transactions that are in the block), Kafka is faster than the Raft, both maximum throughput, minimum throughput and average throughput whether there are transaction failures in the process or not.

Table 2. Simulation Result of Query an Account Process

Type	Maximum Success	Minimum Success	Total Failure (round)	Maximum Throughput (TPS)	Minimum Throughput without Failed (TPS)	Minimum Throughput with Failed (TPS)	Minimum Throughput without Failed (TPS)	Minimum Throughput with Failed (TPS)
Raft	1000	1000	0	77.7	63.2	-	70.55	-
Kafka	1000	1000	0	84.7	68.3	-	74.25	-

Finally, in the simulation of transfer (sending money to another account), Raft and Kafka have equal success possibilities. But for the throughput process, the simulation shows that Raft is faster than Kafka. This can be seen in table 3 that both maximum throughput, minimum throughput and average throughput whether there are transaction failures in the process or not, Raft is faster than Kafka.

Table 3. Simulation Result of Transfer Money Between Accounts Process

Type	Maximum Success	Minimum Success	Total Failure (round)	Maximum Throughput (TPS)	Minimum Throughput without Failed (TPS)	Minimum Throughput with Failed (TPS)	Minimum Throughput without Failed (TPS)	Minimum Throughput with Failed (TPS)
Raft	1000	1000	0	18.1	16.9	-	17.5	-
Kafka	1000	1000	0	17.1	15.3	-	16.13333	-

From the results obtained, the invoke process (create and transfer) on the Raft is superior to Kafka due to several posibilities: (1) Raft has the Raft Consenter which created directly from ordering service nodes compared to Kafka which must make Kafka brokers and Zookeeper Ensembled so that the invoke process is carried out faster. (2) batchTimeout and batchSize are not suitable for Kafka, so there is a failure in the block making process.

5. Conclusion and Future Research

From the research obtained, it is known that Raft is more superior than Kafka in terms of success and speed when conducting invoke transactions due to its simpler framework (Raft has the Raft Consenter which created directly from ordering service nodes) than Kafka (which requires Kafka brokers and Zookeeper Ensembled). However, in the process of querying the transaction, Kafka is superior to Raft since the throughput rate is bigger than Raft.

In further research, CFT in the form of a Raft should be tested for its ability with a new consensus of Hyperledger if the Hyperledger Fabric makes a new consensus such as Practical Byzantine Fault Tolerant or Zero-Knowledge-Proof.

References

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system", Consulted, vol. 1 no. 2012, **(2008)**, pp. 1-9.

[2] Hyperledger, "Members", Hyperledger Project. (n.d.). Retrieved 11 2019, from Hyperledger: https://www.hyperledger.org/members.

[3] C. Ferris, "Hyperledger Fabric: Are we there yet?", IBM Blockchain Blockchain Pulse, **(2017)**, Retrieved 11 2019, from IBM Blockchain Blog: https://www.ibm.com/blogs/blockchain/2017/02/hyperledger-fabric-yet/.

[4] H. Sukhwani, "Performance Modeling & Analysis of Hyperledger Fabric (Permissioned Blockchain Network)", (Doctoral Dissertation, Durham: Duke University), **(2018).**

[5] Hyperledger, "Transaction Flow", Hyperledger Project. (n.d.). Retrieved 11 2019, from ReadtheDocs: https://hyperledger-fabric.readthedocs.io/en/latest/txflow.html.

[6] Hyperledger, "The Ordering Service", Hyperledger Project. (n.d.). Retrieved 11 2019, from ReadtheDocs: https://hyperledger-fabric.readthedocs.io/en/latest/orderer/ordering_service.html.

[7] R. Alagappan, A. Ganesan, J. Liu, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Fault-Tolerance, Fast and Slow: Exploiting Failure Asynchrony in Distributed Systems", Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18), California, USA, **(2018)**, November 8-10.

[8] K. Christidis, "A Kafka-based Ordering Service for Fabric", Hyperledger Project. **(2016)**, Retrieved 11 2019, from GoogleDocs: https://docs.google.com/document/d/19JihmW-8blTzN99lAubOfseLUZqdrB6sBR0HsRgCAnY/edit.

[9] H. Yusuf, I. Surjandari, and A. M. M. Rus, "Multiple Channel with Crash Fault Tolerant Consensus Blockchain Network: A Case Study of Vegetables Supplier Supply Chain", Proceeding of the 2019 16th International Conference on Service Systems and Service Management (ICSSSM), Shenzhen, China, **(2019)**, July 13-15

[10] K. Christidis, "Fabric Proposal: A Raft-Based Ordering Service", Hyperledger Project. **(2018)**, Retrieved 11 2019, from GoogleDocs: https://docs.google.com/document/d/138Brlx2BiYJm5bzFk_B0csuEUKYdXXr7Za9V7C76dwo/edit#.