Performance Evaluation of Ordering Services and Endorsement Policies in Hyperledger Fabric

Christopher Harris
University of Northern Colorado
Greeley, USA
Christopher.harris@unco.edu

Abstract—**Hyperledger Fabric is an open-source project designed to deploy permissioned blockchains. Performance characteristics of Hyperledger Fabric v2.2, such as the impact of ordering services, bottleneck, and scalability) are difficult to understand due to the performance challenges of distributed systems. In this paper, we evaluate the performance of each phase in Hyperledger Fabric's new execute-order-validate architecture. We also assessed the performance of ordering services (Solo, Kafka, and Raft) on the OR and the AND endorsement policies. We found the execution phase was more scalable using the OR endorsement policy than the AND endorsement policy. While all three ordering services (Solo, Raft, and Kafka) performed relatively well, we discuss why Raft may be the best choice for most organizations. Last, we focus on the performance capabilities of theF Raft ordering service using different transaction settings. This evaluation helps to understand some of the trade-offs in Hyperledger Fabric v2.x.**

I. Introduction

Blockchains are immutable digital ledger systems implemented in a distributed fashion (i.e., without a central repository) and typically without a central authority. Blockchain networks enable trusted parties to send and receive ransactions in a peer-to-peer manner verifiably without a need for trusted intermediaries. Therefore, a blockchain allows parties to settle transactions more quickly, resulting in faster movement of goods and services [1].

A blockchain is an open distributed ledger hosted by numerous decentralized devices called *nodes*. Blockchain transactions between two or more parties (irrespective of whether the parties are trusted or untrusted) can be recorded in a verifiable and immutable manner, mitigating the risk of fraud. When transactions occur, additional blocks are created to record this information, the ledger is updated with these blocks, and the updated ledger is then synchronized across each node in a blockchain system.

The Hyperledger project is an open-source collaborative effort hosted by the Linux Foundation to advance blockchain technologies for business enterprises. Hyperledger Fabric is currently deployed in over 400 proof-of-concept and production distributed ledger systems across different industries [2]. Public blockchain networks such as Bitcoin or Ethereum allow anyone to join the network; however, Hyperledger is a permissioned blockchain network in which participants know and can identify each other but do not necessarily trust each other. Therefore, organizations can benefit from a distributed ledger technology (DLT) without the requirement of a fungible currency (e.g., cryptocurrency) [3].

As the Hyperledger project evolves and matures, it is imperative to model the complex interactions between peers performing different yet coordinated functions. Such models provide a quantitative framework from which different configurations can be produced and trade-off decisions considered. A detailed understanding of each phase of Hyperledger Fabric is vital because system bottlenecks, when they occur, can be reduced. Additionally, a performance comparison of the ordering services available in Hyperledger is essential because it helps explain the distinctive performance characteristics of these ordering services (Solo, Kafka, and Raft). In our research, we conducted experiments using Hyperledger Fabric v2.2 to provide performance analysis on the ordering, verification, and endorsement phases. Our study contributes the following:

- An evaluation of the performance characteristics following the Hyperledger Fabric execute-order-validate architecture was first released with Hyperledger Fabric v.1.4.
- An evaluation using decentralized governance for smart contracts - a considerable change from earlier 1.x versions of Fabric where one organization could influence chaincode parameters and endorsement policies for the entire consortium on a channel. With Fabric v2.2, multiple organizations must agree to the chaincode parameters as shown in the endorsement policy. There is now a much more deliberate process to upgrade chaincode that requires a threshold amount of organizations within a consortium to agree before the chaincode can become active on the channel.
- An evaluation of the performance of the settings for the different ordering services (Solo, Kafka, and Raft).
- An examination of the endorsing peers and ordering service node scalability under the OR and the AND endorsement policies.

The remainder of this paper is organized as follows. Section II introduces the essential components of Hyperledger Fabric and Hyperledger Fabric's ordering services. Section III presents our experimental setup on scalability; Section IV presents our results and provides an analysis. Last, Section V concludes our paper.

II. Essential Components of Hyperledger Fabric

A Hyperledger Fabric network comprises (1) "Peer nodes," which execute chaincode, access ledger data, endorse transactions, and interface with applications; (2) "Orderer nodes," designed to ensure blockchain consistency and deliver the endorsed transactions to the peers of the network; and (3) “Membership Service Providers” (MSPs), which are typically implemented as a Certificate Authority, managing X.509 certificates and usedFupd to authenticate member identity and roles.

Fabric ledgers cannot fork as is possible with other distributed blockchains. Fabric-based applications that seek to update the ledger are involved in a three-phase process that ensures all peers in a blockchain network keep their ledgers consistent with each other. One mechanism featured by Fabric is a type of node called an orderer (or ordering node) that performs transaction ordering. Working with other contributing nodes, these form an ordering service. Fabric's design uses deterministic (instead of probabilistic) consensus algorithms, any block that is validated is then guaranteed to be final and correct. Once a transaction has been written to a block, its position within the ledger is immutably assured.

A. Transaction Overview

Fabric-based applications that attempt to update the ledger are involved in a three-phase process. Thist ensures that all blockchain network peers keep ledgers that are consistent with each other. Initially, a client application sends a transaction proposal to a subset of peers that will invoke chaincode (a smart contract in Fabric) to produce a proposed ledger update and subsequently endorse the results. Endorsement is a mechanism in Fabric to check the validity of a transaction (see [4] for an overview and evaluation of various Fabric endorsement policies). Since transactions need to be ordered across multiple nodes, the endorsing peers do not apply the proposed update to their copy of the ledger; instead, the endorsing peers return a proposal response to the client application. The endorsed transaction proposals will ultimately be ordered into blocks in phase two and distributed to all peers for final validation and commitment in phase three.

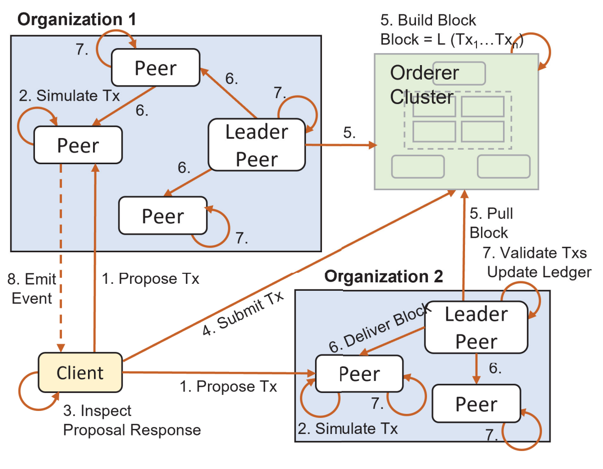

Fig. 1. An illustration of the eight steps used in our proposed Fabric model for ordering and executing chaincode-based transactions.

Fig. 1 illustrates how a participant (i.e., client) invokes a transaction request through the client application; Fig. 2 demonstrates how these roles interrelate with each other depicted by a swim lane diagram.

1) The client application broadcasts the transaction invocation request to the endorser peer.
2) The endorser peer checks the information, such as Certificate Authority details, necessary to validate the transaction. Next, it executes the chaincode and the endorsement responses are returned to the client. As part of the endorsement response, the endorser peer provides a transaction approval or rejection.
3) Next, the client evaluates the response from the endorser peer, a step usually accomplished by a software application.
4) If approved, the client now sends the approved transaction to the orderer peer to be properly ordered and be included in a block.
5) The orderer node creates a transaction (which consists of key-value pairs) in a block and then forwards the block to the anchor nodes of other member organizations of the Fabric network. Consequently, the leader peers deliver that block to the other peers within the organization.
6) Anchor nodes then broadcast the block to the other peers inside their organization, providing consistency within the organization.
7) These individual peers then update their local ledger with this latest block. Thus, synchronization occurs across the entire network. All nodes in the system now contain the updated transaction information.
8) The event status is subsequently emitted back to the client indicating success or failure. Thus, it provides a message to the calling application indicating the success of the upload to the block.

B. Ensuring Transaction Order

The ordering service is an essential component in Fabric; it is one of this framework’s strengths and is ideal for situations in which the ordering of transactions must be ensured.. As mentioned in the previous section, special nodes called orderers receive transactions from different application clients simultaneously. These orderers work together to collectively form the ordering service with the role of arranging batches of submitted transactions into a precise sequence and packaging them into blocks, which become part of the blockchain [5].

Solo, Raft, and Kafka are the most prominent ordering services in Fabric. Solo involves only a single ordering node, Kafka is a more challenging ordering service to implement than Raft, but does not provide namy advantages over a Raft-based implementation. Kafka is not designed to be run across large networks and is more suitable as an ordering service among a tight group of hosts. This limits a Kafka cluster to a single organization and consequently making it a poor choice for most ordering models that operate across organizations [6].

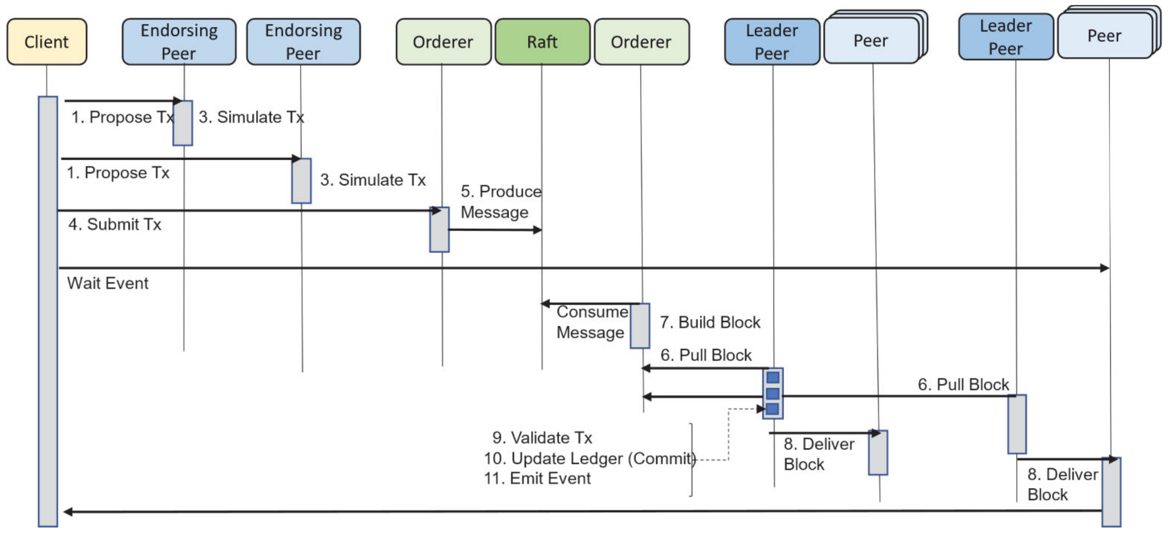

Fig. 2. A swim lane diagram illustrating the order each of the components of our proposed Fabric model interact in order to write a transaction to the block

Orderer peers maintain the central communication channel for Fabric. As a result, the orderer peer ensures a consistent ledger state across the entire network. The orderer peer is responsible for creating the block and delivering the block to all peers within the same network. In addition to promoting finality, separating the endorsement of chaincode execution from ordering gives Fabric advantages in scalability and performance, eliminating crucial bottlenecks which may occur when the same nodes perform both execution and ordering.

The choice of consensus algorithm is critical in the Fabric system. For example, suppose a Fabric system consisted of a single orderer node, and that orderer node fails. Consequently, blocks that were not yet committed would be lost prior, and the integrity of the Fabric system would be lost. Thus, consensus algorithms replicate the blocks to cope with failures. The nodes used to perform data replication are different for each consesnus algorithm. For Kafka, broker nodes perform the block replication, whereas in Raft, these are done by the orderer nodes. The consensus algorithms replicate blocks across all the orderer nodes (broker nodes for Kafka). If the number of replicated blocks in the Fabric system is denoted as n and the minimum number of data replications is denoted as m, the parameter m indicates the *fault-tolerable range*, or the maximum number of crashed nodes that Fabric can continue to operate, is denoted as n-m. If the number of faulty nodes exceeds the fault-tolerable range, the consensus algorithm stops the data replication of blocks or transactions, and Fabric ceases to operate.

The consensus algorithms determine m differently. Solo consists of a single ordering node and is not meant for applications where lost data can be tolerated, and therefore m is not applicable. When using Kafka, m is a configurable parameter, whereas with Raft, m is not configurable.

Ordering notes also enforce basic access control for channels, limiting which nodes can read and write data and which nodes can make these requests. In addition to ordering transactions, orderers also maintain the list of organizations that are permitted to create channels. This list is known as the *consortium* and is kept in the configuration of the orderer system channel. Due to their central role in access control, configuration transactions are processed by the orderer to ensure that the requestor holds the proper administrative rights. If so, the orderer validates the update request against the existing configuration, produces a new configuration transaction, and packages the transaction into a block that is relayed to all peers within the channel.

The peers then process these transactions to verify that the modifications approved by the orderer satisfy the defined channel policies. For example it can be used to keep track of the provenance of goods in a supply chain. Each of these properties can ensure information integrity in the block.

III. BENCHMARKING SCALABILITY

Regrettably, the performance of both permissionless and permissioned/federated blockchain systems lags well behind that of a typical database. The primary bottleneck is the consensus protocol used to ensure node consistency [7] and we investigate this further in this section.

Introduced in Fabric,version v1.1, a new transaction model called the *execute-order-validate model* is used. Mimicking the optimistic concurrency control mechanisms in advanced database systems, this execute-order-validate model consists of three phases. In the first phase, transactions are speculatively executed (or simulated). The global state of the ledger is not affected by this simulation. In the second phase, the transactions are ordered and grouped into blocks. In the third phase, validation or commit, the transcations are checked for conflicts between the order and the execution results. Last, transactions without conflicts are committed to the ledger. By allowing parallel transaction execution, Fabric can achieve a higher transaction throughput than systems that execute transactions sequentially.

When comparing the three types of ordering services, namely Solo, Kafka, and Raft, we note that Solo does not have a consensus algorithm and is only used during the development of blockchain systems, whereas Kafka and Raft are crash fault tolerant (CFT). We evaluate the throughput and latency of transactions using Fabric v2.2 in a local cluster of 32 nodes running with the Raft ordering service. Our work makes use of an internally designed tool. This differs from Blockbench [7], a benchmarking tool that works with earlier versions of Fabric which contained the less robust order-execute transaction model. Many benchmarking studies, such as [8], did not consider the effect of scaling the cluster on performance. Similarly, other studies (e.g., [9], [10]) fix the size of the ordering cluster and use fewer than ten nodes. In contrast, we examine the impact of scaling ordering clusters on the overall performance, using up to 32 nodes in a cluster, which is likely to occur in a majority of more recent Fabric implementations.

In [11], Surjandari et. al. found Raft was superior to Kafka in terms of its transaction success and throughput rates when invoking transactions due to its less complex framework. Raft employs the Raft consenter which was created directly from ordering service nodes whereas Kafka requires Kafka brokers and an elaborate Zookeeper Ensemble. Indeed, most other performance benchmarking studies (e.g. [12]) examine Kafka, whereas the focus of our study is to benchmark Raft.

Although Hyperledger Caliper [13] is the official benchmarking tool for Fabric, it offers little documentation and support on how to benchmark real distributed Fabric ordering services such as Kafka or Raft. Most of the documentation and scripts are considering the Solo orderer and a single client limiting its external validity. To accommodate benchmarking at scale, we use a set of scripts to invoke a Raft-based Fabric network across several cluster nodes, launch additional benchmarking tools, and benchmark performance across distributed clients.

We apply different *endorsement policies* – sets of transaction validation rules that define necessary and sufficient conditions for a valid transaction endorsement [14]. A validation rule usually contains two parts: target endorsing peers and the Boolean operators. The first part dictates the required endorsements from a subset of peers. The second part supports Boolean conditional logic, including AND, OR, and OUTOF. For example, a typical endorsement policy could specify a validation rule requiring at least k endorsements from target p endorsing peers. We focus on the AND and the OR endorsement policies in our study and leave OUTOF operations for a future study.

A. Benchmarking Setup

A Fabric network topology is defined by the number of endorsing peers *N*, the number of clients, *C,* maintaining a mutual transaction send rate*, T*, number of Fabric orderers, *O*, number of Raft consenters, *R*. All the peers belong to different organizations and serve as endorsing peers, *P*. As with [27], the block size is set to 100 transactions/block, the timeout is set to 3 seconds and the number of clients, *C*, is fixed at 1.

We use Smallbank [15], a widely used OLTP database benchmark workload tester. Simulating a typical transfer scenario and a large class of transactional workloads allow us to test Fabric at scale. These experiments were run on a 32-node commodity cluster. Each server node has an Intel Xeon E5-1603 with a processing speed of 2.80GHz, 16 GB of RAM, a Gigabit Ethernet card, and a 1 TB hard drive. Our nodes are running Ubuntu 20.04.3 (Focal Fossa).

As was done in [16], we proposed several designs and principles to avoid potential bottleneck issues. First, we separated endorsing nodes and ordering service nodes, and ran these nodes on different machines. Second, we invoked multiple transactions per client since setting up a client requires time-consuming MSP configurations. In the worst-case scenario, if one client is set up per transaction, it would require significant computing overhead. Third, we invoked transactions asynchronously, starting new transactions without waiting for responses from other transactions, which replicates a real-world scenario. Last, to double-check load generation and reception information, we implemented a transaction logging system.

B. Metrics

We focused on transaction throughput and transaction latency using the definition used in Hyperledger Fabric's white paper.

Throughput - the rate at which transactions are committed to the ledger in transactions per second (tps).

Latency - the committing timestamp of a transaction minus the submission timestamp of a transaction, where the committing timestamp means the timestamp when a transaction is committed to the ledger. Following this approach, we can calculate the transaction latency for each transaction and the average latency.

IV. Results and Analysis

We examined transactions to each of the three ordering services separately. We applied the AND and the OR endorsement policies and calculated the overall throughput and overall latency. Our goal is to best replicate the conditions in [17], which examined these types of transactions using Fabric v1.4.3.

A. Overall Throughput.

Fig. 3 and Fig 4 illustrate the overall transaction throughput under different ordering services and endorsement policies. We used an OR endorsement policy requiring 17 (a majority) of the 32 participating nodes, or all 17 nodes that were pre-specified for the AND endorsement policy.

There are two findings related to the throughput and the chosen endorsement policy.

- First using the OR endorsement policy and a transaction size of 1 byte (Fig 3), the maximum throughputs are nearly 300 tps. There was no significant difference in the maximum performance achieved by each of the ordering services (Solo, Raft and Kafka).
- The AND endorsement policy (Fig. 4) achieved a maximum throughput of 200 tps, a significant drop from the OR endorsement policy for the same number of required nodes.

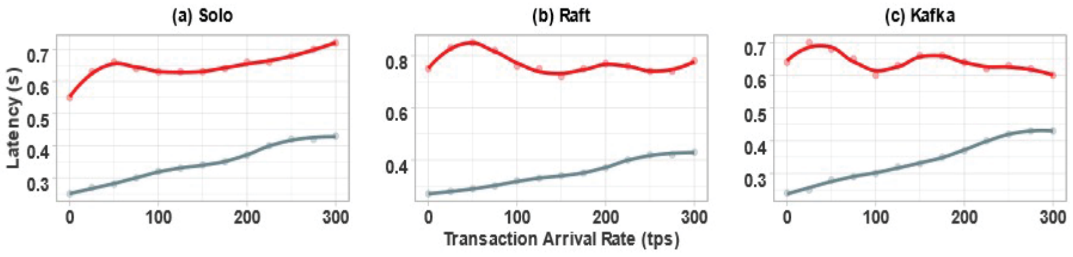

Fig 3. Comparison of the transaction latency of the “order and validate” phases (red) with the “execute” phase (black) for the three different ordering services using the OR endorsement policy

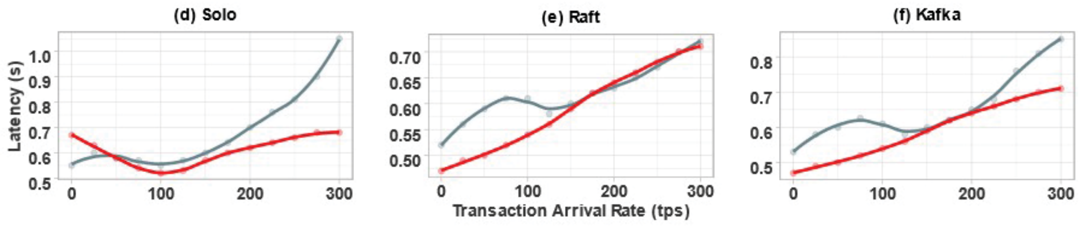

Fig 4. Comparison of the transaction latency of the “order and validate” phases (red) with the “endorse” phase (black) for the three different ordering services using the AND endorsement policy

B. Overall Latency.

Fig 5 and Fig 6 illstrate an overall transaction latency under different ordering services and endorsement policies. One observation is that when the system reaches peak performance, the transaction latency will increase rapidly. In particular, the transaction latency with the endorsement policy AND (Fig. 6) increases more significantly with the OR endorsement policy (Fig 5). Since the maximum throughput of AND is considerably lower than OR, the performance bottleneck comes earlier than OR. In our experiment, we used Nodejs and set the maximum transaction latency for the ordering service at 3 seconds – if our client failed to receive a response from the ordering service nodes within 3 seconds, the client would reject the response. In a worst-case scenario – in which transactions have an ordering latency of 3 seconds or greater – the client rejects those transactions. Therefore, the overall transaction latency will increase rapidly when the transaction arrival rate surpasses the peak system performance, since many subsequent transactions will be rejected.

C. Evaluation of Performance using Raft

Next, we focus on the most implemented ordering service, Raft, and examine the effects of increasing the number of orderers and peers. We chose Raft since it was the easiest of the CFT policies to implement, is widely used, and Solo and Kafka have been deprecated beginning with Fabric 2.0).

Overall, as the number of Fabric orderers increases, system throughput performance significantly degrades due to elevated communication overhead – in Fig 7 (a)), we increase the number of orderers from O = 4 to O = 12. In our subsequent experiments, we fix the number of Fabric orderers to O = 4 and vary the number of Raft consenters, R, to assess the

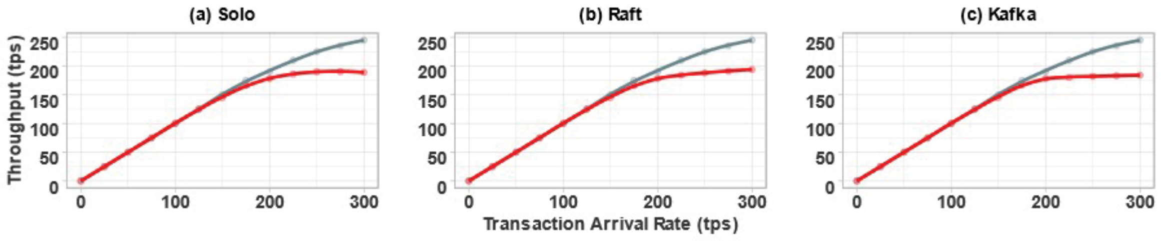

Fig 5. Overall transaction throughput using the “OR” endorsement policy (red) with the “AND” endorsement policy (black) for the three different ordering services

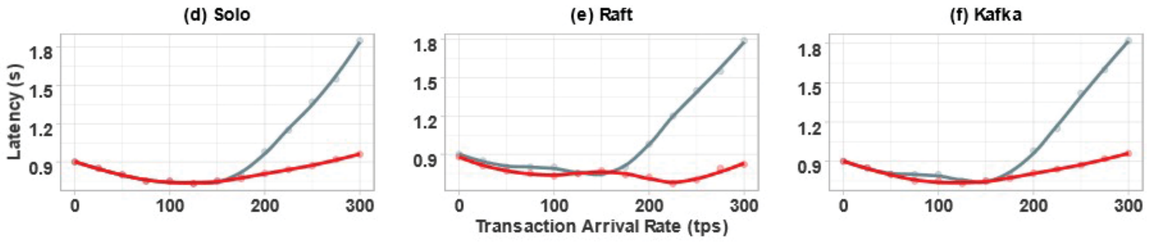

Fig 6. Overall latency using the “OR” endorsement policy (red) with the “AND” endorsement policy (black) for the three different ordering services.

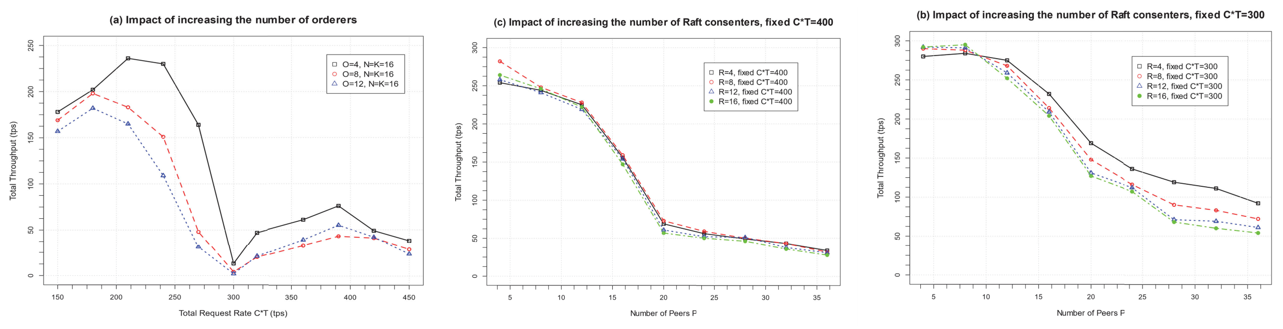

Fig. 7. Changes in performance as (a) we increase the number of orderers from O = 4 to 12; we increase the number of Raft consenters, R, from 4 to 16 and we increase the number of peers, P, from 4 to 32 for (b) C*T = 300 and (c) C*T = 400.

impact on scaling Fabric.

At the networking level, we find that the client, peers, Fabric orders, and Raft consenters exhibit the highest traffic both when sending and receiving. The increased traffic of the Fabric orderer is explained by its double role as a receiver of requests from the client and as a dispatcher of requests to Raft.

Next, we fix the request rate and investigate the impact of increasing the number of peers. We set the total request rate to $C * T = 300$ and $C * T = 400$, representing the points before and after saturation. We vary the number of Raft consenters from R= 4 to R = 16 in multiples of 4. We increase the number of peers to P = 32. The average throughput and latency of each experimental setting are shown in Fig 7(b) and Fig 7(c), respectively.

D. Raft Performance Results and Analysis

Increasing the number of peers has a strong negative impact on the system's throughput and it limits scalability. We examine the system logs and observe that scaling the number of peers produces significant overhead in both the endorsement and ordering phases of the transaction. As the number of endorsing peers grows, each client must wait for a more extensive set of endorsements from a larger group of peers to prepare the endorser transaction.

From the logs, we observe that the clients return numerous timeout errors while collecting the endorsements from peers and discard those transaction proposals accordingly. Consequently, clients send endorser transactions to the orderers at a decreased rate due to dropped transactions and the time overhead for collecting endorsements. Unlike our findings when scaling the peers, we find that scaling the Raft consenters does not impact the throughput pattern or scalability of the system, which is similar to what [12] found performing a similar benchmark with the Kafka brokers.

We note the following observations:

- The Raft ordering service is not a limitation on our system, thus providing the ability to conduct transactions across different agencies or offices (peers)
- Given our node configurations, when the number of peer nodes engaging in transactions approaches 32, we notice a strong performance decrease; we would need to set the timeout to greater than 3 seconds, particularly when network latency can be anticipated.
- Fabric v.2.2 can attain a better throughput than previous versions of Fabric, especially with the newer decentralized governance for smart contracts. However, it is still unable to achieve OLTP levels of performance; however, using a blockchain allows us to ensure tamper-proof transactions.

Of the three phases, we found the validation phase was the bottleneck due to the larger time required to validate blocks (and transactions) – while v2.2 improves this slightly from earlier versions of Fabric, it remains an area for improvement.

V. Conclusion

We presented a performance evaluation and analysis of latency and throughput on version 2.2 of Hyperledger Fabric. We examined the performance characterization of each phase of the transaction life cycle and have made a comparison of different ordering services. The execute phase showed good performance scalability under the OR endorsement policy but far worse performance under the AND endorsement policy. The validation phase was the system bottleneck, not only because the speed of transaction and block validation is relatively low, but also because there is a high computational burden on the validation node.

We have investigated the performance as the number of nodes, orderers, and Raft consenters increased using Smallbank and a modified version of Hyperledger Caliper as our benchmarking tools. Although increasing the number of orderers degrades the system performance; also, an increase in the number of peers negatively affects system throughput and limits scalability as we approach 32 nodes, while additional stress on the Raft ordering service does not have a comparable decrease in performance. We note while the benefits of our proposed model provide for tamper-proof transactions, it is not meant to, nor does it need to, be transaction intensive.

Blockchain technology is still a nascent technology, with its vast potential understood but to date there have only been limited endeavors in solving real-world problems. In future work, we plan to investigate different ways transactions can be run concurrently and its effects, as well as explore some of the more advanced settings in Fabric v2.x.

References

[1] D. Tapscott and A. Tapscott, Blockchain Revolution : How the Technology behind Bitcoin is Changing Money, Business, and the World. `New York: Portfolio Penguin, 2016.

[2] E. Androulaki et al., "Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains," in EuroSys,2018, pp.30:1–30:15.,

[3] M. Vukoli'c, "Rethinking Permissioned Blockchains," in ACM Workshop on Blockchain, Cryptocurrencies, and Contracts (BCC), 2017, pp. 3–7.

[4] M. Soelman, V. Andrikopoulos, J. A. Pérez, V. Theodosiadis, K. Goense, & A. Rutjes, (2020). Hyperledger Fabric: Evaluating Endorsement Policy Strategies in Supply Chains. In 2020 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPS) (pp. 145-152). IEEE.

[5] C. G. Harris (2021). Using Hyperledger Fabric to Reduce Fraud in International Trade. 2021 IEEE International Conference on Artificial Intelligence and Blockchain Technology (AIBT) IEEE.

[6] Androulaki, E., Barger, A., Bortnikov, V., Cachin, C., Christidis, K., De Caro, A., ... & Yellick, J. (2018). Hyperledger fabric: a distributed operating system for permissioned blockchains. In Proc thirteenth EuroSys conference 1-15.

[7] Dinh, T. T. A., Wang, J., Chen, G., Liu, R., Ooi, B. C., & Tan, K. L. (2017). Blockbench: A framework for analyzing private blockchains. In Proceedings of the 2017 ACM Int'l Conference on Management of Data (pp. 1085-1100).

[8] Androulaki, E., Barger, A., Bortnikov, V., Cachin, C., Christidis, K., De Caro, A., ... & Yellick, J. (2018,). Hyperledger fabric: a distributed operating system for permissioned blockchains. In Proc thirteenth EuroSys conference 1-15.

[9] Baliga, A., Solanki, N., Verekar, S., Pednekar, A., Kamat, P., & Chatterjee, S. (2018). Performance characterization of hyperledger Fabric. In 2018 Crypto Valley conference on blockchain technology (CVCBT) (pp. 65-74). IEEE.

[10] Thakkar, P., Nathan, S., & Viswanathan, B. (2018). Performance benchmarking and optimizing hyperledger fabric blockchain platform. In 2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS) (pp. 264-276). IEEE.

[11] Yusuf, H., & Surjandari, I. (2020). Comparison of performance between kafka and Raft as ordering service nodes implementation in hyperledger Fabric. International Journal of Advanced Science and Technology, 29(7s), 3549-3554.

[12] Nguyen, M. Q., Loghin, D., & Dinh, T. T. A. (2021). Understanding the scalability of Hyperledger Fabric. arXiv preprint arXiv:2107.09886.

[13] Hyperledger Caliper. https://www.hyperledger.org/projects/caliper

[14] F. Benhamouda, S. Halevi, and T. Halevi, "Supporting private data on Hyperledger fabric with secure multiparty computation," IBM Journal of Research and Development, vol. 63, no. 2/3, pp. 3–1, 2019.

[15] Cahill, M. J., Röhm, U., & Fekete, A. D. (2009). Serializable isolation for snapshot databases. ACM Transactions on Database Systems (TODS), 34(4), 1-42.

[16] Wang, C., & Chu, X. (2020, November). Performance characterization and bottleneck analysis of hyperledger Fabric. In 2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS) (pp. 1281-1286). IEEE.

[17] Wang, C., & Chu, X. (2020, November). Performance characterization and bottleneck analysis of hyperledger fabric. In 2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS) (pp. 1281-1286). IEEE.