IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 23 June 2022, accepted 10 July 2022, date of publication 14 July 2022, date of current version 21 July 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3190979

RESEARCH ARTICLE

Resource Analysis of Blockchain Consensus Algorithms in Hyperledger Fabric

GYEONGSIK YANG[1], (Member, IEEE), KWANHOON LEE[1], KYUNGWOON LEE[2], YEONHO YOO[1], (Graduate Student Member, IEEE), HYOWON LEE[1], AND CHUCK YOO[1], (Member, IEEE)

[1]Department of Computer Science and Engineering, Korea University, Seoul 02841, South Korea
[2]School of Electronics Engineering, Kyungpook National University, Daegu 41566, South Korea

Corresponding author: Chuck Yoo (chuckyoo@os.korea.ac.kr)

This work was supported in part by the Institute of Information and Communications Technology Planning and Evaluation through the Korea Government (Ministry of Science, ICT), SW Starlab, Next Generation Cloud Infra-Software toward the guarantee of performance and security SLA, under Grant 2015-0-00280; in part by the Basic Science Research Program through the National Research Foundation of Korea funded by the Ministry of Education under Grant NRF-2021R1A6A1A13044830; and in part by the Korea University Grant.

ABSTRACT In the blockchain network, the consensus algorithm is used to tolerate node faults with data consistency and integrity, so it is vital in all blockchain services. Previous studies on the consensus algorithm have the following limitations: 1) no resource consumption analysis was done, 2) performance analysis was not comprehensive in terms of blockchain parameters (e.g., number of orderer nodes, number of fault nodes, batch size, payload size), and 3) practical fault scenarios were not evaluated. In other words, the resource provisioning of consensus algorithms in clouds has not been addressed adequately. As many blockchain services are deployed in the form of blockchain-as-a-service (BaaS), how to provision consensus algorithms becomes a key question to be answered. This study presents a kernel-level analysis for the resource consumption and comprehensive performance evaluations of three major consensus algorithms (i.e., Kafka, Raft, and PBFT). Our experiments reveal that resource consumption differs up to seven times, which demonstrates the importance of proper resource provisioning for BaaS.

INDEX TERMS Blockchain, blockchain-as-a-service, cloud, consensus algorithm, hyperledger fabric, performance analysis, Kafka, Raft, PBFT, Microsoft Azure, AWS.

I. INTRODUCTION

The successful launch of Bitcoin in 2010 [1] brought blockchain into the spotlight. Blockchain is based on a decentralized architecture to ensure highly secure transaction validations, offering anonymity and auditability. The distributed architecture requires a consensus algorithm between nodes [2] so that the participant nodes in a blockchain network[1] maintain data consistency and integrity. The data consistency ensures that the blockchain remains reliable even when the nodes fail. Also, the integrity prevents data manipulation, which makes the validation of nodes trustworthy.

Recently, blockchain-based services operate on cloud systems, such as Microsoft Azure [3] and Amazon web services (AWS) [4] in the form of blockchain-as-a-service (BaaS). An advantage of BaaS enables operators to scale their blockchain services dynamically according to their usage. This advantage has led to a growing number of BaaS services, and BaaS is used in a variety of industries such as health care (e.g., IBM digital health pass) [5], financial services [6], energy trading [7], internet-of-things [8], [9], and supply chain management in manufacturing industries [10]. Clouds provide dynamic scale-in and out on computing resources [11]–[15]; thus, the success factor of BaaS depends on how blockchain services are efficiently provisioned on BaaS.

The associate editor coordinating the review of this manuscript and approving it for publication was Jules Merlin Moualeu.

[1]The term "network" refers to two concepts: a single blockchain and the data sent and received between computers. For clarification, we use "network" to express a single blockchain and "communication" for the data sent and received between computers.

The most widely used BaaS platform is Hyperledger Fabric [16], an open-source project maintained by the Linux Foundation. BaaS allows only pre-authorized nodes to participate in a blockchain system so that it exhibits relatively high performance (in terms of throughput and latency) compared with other public blockchain systems [17]–[19]. Hyperledger Fabric allows its system components to be selected flexibly, which can be tailored for respective services [20], [21]. For example, one of the essential components is the consensus algorithm such as Kafka [22], Raft [23], or practical byzantine fault tolerance (PBFT) [24]. Although the three algorithms are aimed at the data integrity and reliability of Hyperledger Fabric systems (details are provided in Section II-B), there are significant differences in how each algorithm operates to achieve the goal.

Because of the importance of consensus algorithms, various state-of-the-art studies on Hyperledger Fabric [25]–[32] have investigated the performance impact of consensus algorithms. However, these studies have critical limitations. First, existing studies only measured the performance of the consensus algorithms but did not measure their resource consumption. Thus, it is challenging to determine how the computing resources are to be provisioned for BaaS. So, the computing resources have to be over-provisioned to run the consensus algorithms.

Second, each BaaS system has numerous configurable parameters, such as the number of nodes for the consensus algorithm (called the number of nodes), number of transactions for batch processing (batch size), and maximum size of each transaction (payload size). However, existing studies only included one or two configurable parameters in their evaluations. Thus, they fall short of covering the comprehensive configurations of BaaS.

Third, the consensus algorithm enables the Hyperledger Fabric system to sustain its performance even when the nodes fail. Because Hyperledger Fabric is a distributed system of multiple nodes, the consensus algorithm has a "fault-tolerable range," which represents the maximum number of node faults that the system can endure. When a blockchain system consists of tens of hundreds of nodes, it is critical to comprehend the performance and resource consumption of Hyperledger Fabric in the presence of the number of fault nodes. Practical blockchain systems are expected to have a tolerable fault range of six to seven nodes [27], [28], [33]. However, existing studies only evaluated two to three fault nodes. Consequently, a larger fault-tolerable range remains to be investigated to demonstrate the practicality of consensus algorithms in terms of node faults.

To address these limitations, we first present in-depth analysis and comparison of the architectures of three consensus algorithms, Kafka, Raft, and PBFT (Section III). Then, we run comprehensive experiments to evaluate their operations and fault handling (Section IV and Section V). The novel contributions of this study that differ from those of previous studies are summarized as follows.

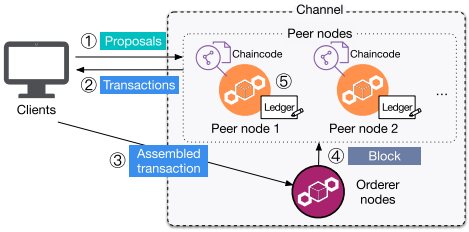


FIGURE 1. Hyperledger Fabric nodes.

- To the best of our knowledge, this is the first study to evaluate the resource consumption of consensus algorithms in BaaS.
- This study evaluates the performance of Hyperledger Fabric of up to eight node faults. Moreover, we analyze how the performance changes as the number of faults increases.
- Experiments are carried out with more varied parameters of the Hyperledger Fabric system (e.g., number of nodes, number of fault nodes, batch size, and payload size) than those in previous studies. The parameter combinations total 355 cases, and we conduct the experiments three times per case.
- We report unique findings from the experiments, which have not been previously reported. So, we provide suggestions on resource provisioning and Hyperledger Fabric system configurations (e.g., batch size and block creation rate, provisioning priority between orderer nodes, and orderer nodes in PBFT algorithm, Section V-F).

In the remainder of this paper, we present the motivating experiments in Section II. In Section III, we briefly explain the operations of Kafka, Raft, and PBFT in Hyperledger Fabric. In Section IV, evaluation methodologies and their rationale are described. Section V shows evaluation results and analyses, and Section VI summarizes the related work and novelties of this work. Section VII has the discussion points, and finally, this study concludes in Section VIII.

II. BACKGROUND AND MOTIVATION

In this section, first, a brief introduction to Hyperledger Fabric is presented to aid in understanding the consensus algorithms. We then compare three consensus algorithms at high-level. In addition, as motivating experiments, we measure the resource consumption of the consensus algorithms.

A. OVERVIEW OF HYPERLEDGER FABRIC

Fig. 1 shows the four nodes of a Hyperledger Fabric system: 1) peer node, 2) chaincode, 3) ledger, and 4) orderer node. The workflow of Hyperledger Fabric is as follows. The client sends proposals (① in Fig. 1) to the peer nodes. The peer nodes convert them into a transaction and send the transaction back to the client (②). Then, the client sends the transaction to the orderer nodes (③), where a block is created (④). Finally, the orderer nodes send the block to the

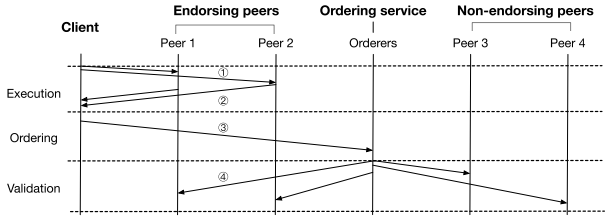


FIGURE 2. Hyperledger Fabric operation flowchart.

peer nodes, which store the block in the ledger. Thereafter, the block is committed to the Hyperledger Fabric system (⑤). To help understand consensus algorithms, we explain the key operations of Hyperledger Fabric: execution, ordering, and validation.

1) EXECUTION

In Hyperledger Fabric, two types of peer nodes exist: endorsing peers and non-endorsing peers. Endorsing peers receive proposals and create transactions, whereas non-endorsing peers do not participate in transaction creation and only participate in transaction validation when a block is committed. In the execution step, a client issues proposals to all endorsing peers (Peer 1 and Peer 2, ① in Fig. 2). Then, each endorsing peer creates a transaction in two steps: 1) chaincode execution converts the proposal into the transaction, and 2) the transaction is endorsed via a digital signature with the private key of the endorsing peer. Endorsing peers return the created transaction to the client. In Fig. 2, the client receives two transactions from Peer 1 and Peer 2 (②).

The client then checks the validity of the signature in each transaction, that is, the existence of a signature and the source of the signature. This checking criterion is called the ''endorsement policy.'' After confirming that the received transactions are all endorsed, the client assembles the transactions into a single transaction that contains all the signatures of the endorsing peers. The assembled transaction is sent to the orderer nodes (③).

2) ORDERING

In the ordering step, the orderer node collects the transactions.[2] When the orderer node has collected a certain number of transactions (batch size), it packages the transactions into a ''block.'' Once the block is created, the orderer node replicates the block using a consensus algorithm so that all orderer nodes have the same block. We explain the consensus algorithms in detail in Section III. Once all orderer nodes have the same block, orderer nodes distribute[3] the block to peer nodes (including both endorsing and non-endorsing) (④). Note that each orderer node has a set of peer nodes to deliver the block, which is pre-determined when the Hyperledger Fabric system is initialized [34].

3) VALIDATION

In the validation step, when all peer nodes receive a block from orderer nodes, transaction validation is performed over the block. Each peer node validates whether the endorsements within the transactions satisfy the endorsement policy. If the endorsement policy is satisfied, then peer nodes commit the block to their ledgers.

B. COMPARISON OF CONSENSUS ALGORITHMS

The consensus algorithm is critical in the Hyperledger Fabric system [35]. Suppose a Hyperledger Fabric system consists of a single orderer node. If the orderer node fails, the blocks are lost before being committed, which means that the integrity of the Hyperledger Fabric system is lost. Thus, consensus algorithms replicate the blocks to cope with failures. The nodes that perform data replication are different for each consensus algorithm. For Kafka, broker nodes perform the block replication, while orderer nodes do for Raft and PBFT (Section III). The consensus algorithms replicate blocks across all the orderer nodes (broker nodes for Kafka).

We denote the number of replicated blocks in the Hyperledger Fabric system as n. The consensus algorithms have the minimum number of data replications, m. The parameter m is related to the fault-tolerable range that is the maximum number of crashed nodes with which Hyperledger Fabric continues to operate. In other words, the fault-tolerable range is n-m. If the number of fault nodes exceeds the fault-tolerable range, the consensus algorithm stops the data replication of blocks or transactions, and Hyperledger Fabric ceases to operate.

The consensus algorithms determine m differently. For Kafka, m is a configurable parameter. If n of the system is 10 and m is 3, the fault-tolerable range is 7. For Raft and PBFT, m is not a configurable parameter and the details of determining m is discussed in Section III while the fault-tolerable range is n-m.

C. MOTIVATING EXAMPLE

The above differences in consensus algorithms can affect their performance characteristics and resource consumption. Thus, to compare the consensus algorithms, we conduct experiments and compare them on a cluster of seven servers (the detailed experiment setting in Section IV-A). The experiments are done with proper m (i.e., three, three, and four for Kafka, Raft and PBFT, respectively) for the same fault-tolerable range of one.

Fig. 3 shows the results of the performance and resource consumption. As for performance (Fig. 3a), we measure the throughput , which is the number of transactions committed per second, and latency, which is the time that each transaction takes to commit. The throughput and latency are measured through Caliper benchmark running in the Hyperledger Fabric system. For the resource consumption

[2]We use the term ''transaction'' to refer to the assembled transaction for the ordering and validation processes.

[3]Note that we denote the block delivery between orderer nodes as ''replication'' and delivery from orderer nodes to peer nodes as ''distribution'' to distinguish the two.

(Fig. 3b), we measure the CPU cycles (%) and communication bandwidth (MB/s) on each orderer node with *pidstat* and *vnstat*, and we depict the average of the orderer nodes.

Specifically, Fig. 3a presents the comparisons of the throughput (left y-axis with bars) and latency (right y-axis with dots). The throughput of Raft is 1.54× higher than that of PBFT. Also, in terms of latency, Raft shows 3× better latency than PBFT. Second, Fig. 3b shows the CPU cycle (left y-axis with bars) and communication bandwidth (right y-axis with dots). Raft consumes 35% and 65% fewer CPU cycles than Kafka and PBFT, respectively. However, the communication bandwidth of PBFT is higher than Kafka and Raft up to 4× and 7×, respectively. In short, although the three consensus algorithms operate with the same fault-tolerable range, the evaluation results indicate that they exhibit significant differences in the performance and resource consumption.

When running BaaS in clouds, therefore, understanding the resource impact of consensus algorithms is key to building reliable and stable services. However, to our knowledge, no previous studies have offered such analysis. To this end, this study conducts a thorough analysis of their architectures and carries out a comprehensive evaluation to characterize the performance and resource consumption for BaaS. We experiment with varying parameters that affect the performance and resource consumption of Hyperledger Fabric systems, such as the number of nodes, the number of fault nodes, batch size, and payload size, which results in 355 evaluation cases (details in Section IV-B). This provides far more extensive than previous studies.

III. CONSENSUS ALGORITHMS

Here, we analyze the architectures of Kafka, Raft and PBFT. We explain the structure, operations, and fault tolerance of each algorithm, which forms the basis for the performance and resource consumption evaluations in Section V.

A. KAFKA

1) STRUCTURE AND OPERATION

Kafka [22] was originally proposed by LinkedIn to store its user data in a distributed system. Kafka comprises three roles: 1) producer, 2) consumer, and 3) broker nodes. The producers create the data,[4] and the consumers read the data from the producer through the broker nodes. The data for producers and consumers is stored in the broker nodes, a set of multiple nodes separated from producers and consumers. The producers and consumers write and read the data independently from the broker nodes.

When Kafka is used in Hyperledger Fabric, the broker nodes are responsible for data replication (transactions). Specifically, the broker nodes and orderer nodes operate as follows. An orderer node first receives a transaction from

[4]We use the term "data" when we explain the general architecture of a consensus algorithm. When we explain the consensus algorithm in Hyperledger Fabric, we use the term "transaction" or "block."

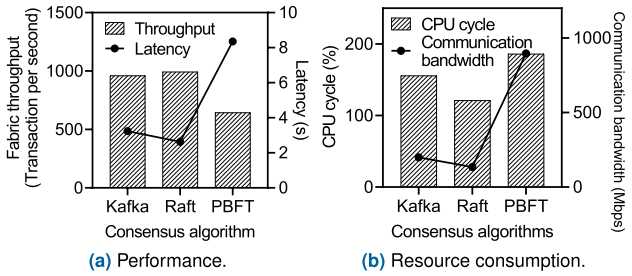


FIGURE 3. Evaluation of consensus algorithms.

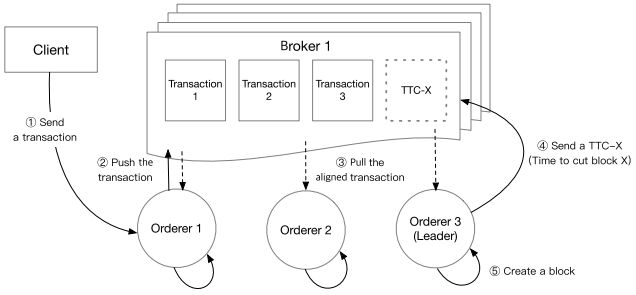


FIGURE 4. Kafka operation in Hyperledger fabric.

a client (① in Fig. 4). The orderer node that receives transactions from the client is pre-determined when the Hyperledger Fabric system is initialized. The orderer node then sends the transaction to a broker node, called the leader broker node. This operation is denoted "push" (②). The leader broker node determines the order of the pushed transaction. After sorting the order of transactions, the leader broker node performs transaction replication—replicating the transaction between the other broker nodes. Each orderer node then reads the replicated transactions from the leader broker node (③, denoted as "pull").

The pull operation is performed as follows. All orderer nodes periodically send messages to the leader broker node to determine whether new transactions exist to pull. This message contains both the index of the last-pulled transaction and the buffer size for storing new transactions. Upon receiving the message, the leader broker node checks whether new transactions have been sent. If this is the case, the leader broker node sends a reply message containing the new transactions as large as the buffer size.

After pulling the transactions, one of the orderer nodes (leader orderer node) decides to create a block when the number of pulled transactions exceeds a certain number (batch size). The selection of the leader orderer node is explained in Section III-A2. The leader orderer node then sends (pushes) a "time to cut block X" (TTC-X) message to the leader broker node (④) that contains the range of transactions to be wrapped in the block.

Once receiving the TTC-X message, the leader broker node replicates the TTC-X between broker nodes. Also, other orderer nodes receive the TTC-X message from the leader broker node through pull operation. Each orderer node then creates the block (⑤) and distributes it to the peer nodes.

2) FAULT TOLERANCE

In Kafka, the parameter m is considered as a configured parameter. When the number of broker nodes is n, each broker node has a replicated transaction, so the Hyperledger Fabric system has a total of n replicated transactions. The fault-tolerable range of Kafka is then n-m. When the number of replicated transactions becomes less than m, Kafka considers the system is unreliable to maintain consistency of transactions; thus, no further transactions are processed. For example, the default value of m in Kafka is three. Then, when n is seven and three broker nodes are operational, Kafka considers the system is reliable. However, if one more broker nodes fail, only two transaction copies are live. Then, the minimum m is not met; so, no more transactions are processed.

a: LEADER ORDERER NODE SELECTION

Because the leader orderer node determines the generation of blocks, Kafka maintains at least one leader orderer node alive in the system. So, when the current leader orderer node fails, a new leader is selected by Apache Zookeeper [36].

Zookeeper continuously monitors the status of the orderer nodes.[5] Each orderer node has an index number that is assigned sequentially when the Hyperledger Fabric system is initialized. If Zookeeper detects a fault on the leader orderer node, it immediately sends messages to the remaining orderer nodes (called follower nodes). Zookeeper then checks the index number of the failed leader orderer node and selects the orderer node of the following index number as the leader. Then Zookeeper sends a message containing the ID and network address of the new leader to follower nodes. With the newly selected leader orderer node, Kafka resumes its operations.

b: LEADER BROKER NODE SELECTION

When a leader broker node halts, 1) the orderer nodes can no longer perform push and pull operations, and 2) the transaction replication between broker nodes stops. To avoid such a situation, when the leader broker node goes down, Zookeeper selects the next leader broker, similar to the leader orderer node selection. As with the index numbers for orderer nodes, each broker node also has an index number that is determined at system startup. Based on the index number of the fault leader broker node, the broker node with the following index number is selected.

B. RAFT

1) STRUCTURE AND OPERATION

Raft [23] is another consensus algorithm used in Hyperledger Fabric. In contrast to Kafka, which requires broker nodes to store transactions, Raft accomplishes consensus with orderer nodes. In Raft, the orderer nodes maintain the leader and follower properties. Only the leader orderer node creates a new block with transactions and then replicates the block to the follower nodes.

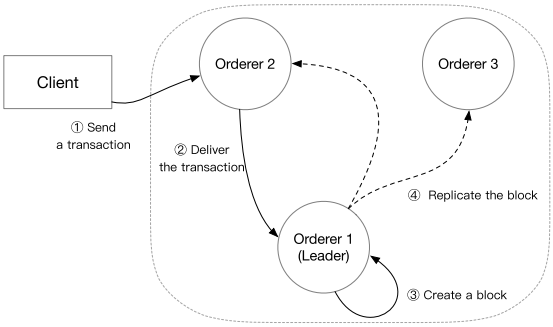


FIGURE 5. Raft operation in Hyperledger fabric.

Fig. 5 presents the detailed operations of Raft as follows. First, an orderer node receives a transaction from the client (①). The orderer node then delivers it to the leader orderer node—"Orderer 1" (②). Orderer 1 accumulates the transactions in the order in which they arrive until the batch size exceeds the limit. After the accumulation, the leader orderer node creates a block with the transactions (③). Orderer 1 then sends (replicates) the block to other orderer nodes (followers) (④). After receiving the block, each follower orderer node stores the received block. Orderer 1 then sends messages to the follower orderer nodes to confirm whether all follower nodes have successfully received the block. If they have not, Orderer 1 retransmits the block. When all orderer nodes have successfully replicated the block, the orderer nodes (including leader and follower) distribute the block to the peer nodes.

2) FAULT TOLERANCE

When the number of replicated data (orderer nodes) is n, the parameter m of Raft is determined as the $\lceil (n+1)/2 \rceil$. Then, the fault-tolerable range of Raft is $\lceil (n-1)/2 \rceil$, which means that Raft withstands faults when more than half of the orderer nodes are normal [23]. For example, when n is five, m is three, so Raft can process the transactions until the failures of two orderer nodes (fault-tolerable range of two). When n is six, m is four, so the fault-tolerable range is still two. In general, the number of orderer nodes is configured to be an odd number.

Leader orderer node selection: When the leader orderer node faults occur, no new transactions can be received, and the generation of blocks halts. Thus, Raft selects a new leader orderer node as follows. Raft maintains at least one leader node through "heartbeat mechanism." The leader node periodically sends "heartbeats" to notify the follower orderer nodes of its existence. If the follower orderer nodes do not receive the heartbeat in a particular time, the follower orderer nodes initiate the leader election. Then, randomly, some of the follower orderer nodes change their status from "follower" to "candidate." The candidates send heartbeats to the other orderer nodes. Each orderer node replies to the candidate whose heartbeat it receives first. The candidate with the

[5]Note that as Zookeeper does not participate in the block consensus operations itself, we do not depict it in Fig. 4.

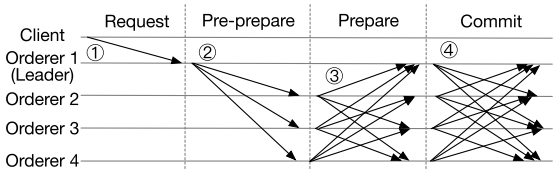


FIGURE 6. PBFT operation in Hyperledger fabric.

highest number of responses from the orderer nodes is elected as a new leader. If there is a tie, the leader election repeats until a single leader is elected. When a new leader orderer node is elected, the leader orderer node starts sending a heartbeat to the follower orderer nodes and generating blocks.

C. PBFT

1) STRUCTURE AND OPERATION

PBFT [24] has the leader and follower property that is similar to Raft. However, the follower orderer nodes in PBFT perform the verification operation of the created blocks in addition to the block distribution. Specifically, Fig. 6 shows PBFT operations in the four stages: 1) request, 2) pre-prepare, 3) prepare, and 4) commit[6] [24]. First, the leader orderer node receives a transaction from the client (① in Fig. 6, request stage). The leader orderer node accumulates the transactions until the number exceeds the batch size, and the leader orderer node creates a new block from the transactions. Then, the leader orderer node performs data replication— delivering the block to the follower orderer nodes. When receiving the block, each follower orderer node performs verification of the received block, checking the transaction order within the block and the digital signature from the leader orderer node (②, pre-prepare stage).

Once each follower orderer node completes the verification, it adds the received block and its own digital signature to the "prepare message." Then each follower orderer node broadcasts the prepare message to the other orderer nodes (③, prepare stage). When a follower orderer node receives the prepare messages as many as a specific threshold (k), it broadcasts a "commit message" indicating that it agrees on committing the block. The threshold is related to fault tolerance, which is explained in the next subsection. Each orderer node finally commits the block (④, commit stage). The block is then handed over to peer nodes.

Note that the operations ② and ③ are performed asynchronously among the orderer nodes. Therefore, from the perspective of each orderer node, 1) the reception of a block from the leader orderer node (pre-prepare stage) and 2) the reception of m commit messages from other nodes (prepare stage) can be performed independently. Consequently, as long as an orderer node gets m commit messages, the orderer node can directly commit the blocks (④)

[6]Note that the terminology "commit" used in the four stages of PBFT does not refer to the final block commit from peer nodes to ledgers. Rather, PBFT uses the term "commit" to indicate the end of the processing on a block.

2) FAULT TOLERANCE

When the number of orderer nodes in Hyperledger Fabric is n, the PBFT algorithm determines the parameter m as $(2n+1)/3$. In other words, PBFT requires at least 2/3 of the orderer nodes to be functional to withstand faults. For example, when n is seven, m of PBFT is five. In other words, among seven orderer nodes, PBFT requires at least five replicated data in orderer nodes, which means that PBFT withstands up to two orderer node failures—its fault-tolerable range becomes two. Note that PBFT requires n to be in the form of multiples of three plus one, for example, of four, seven, or ten, which is for making $(2n+1)/3$ be an integer.

For fault tolerance, PBFT also considers the minimum number of verifications on the blocks. Each orderer node waits for m commit messages, which is $(2n+1)/3$. For example, when n is seven, m becomes five. Thus, each orderer node waits for receiving five prepare messages from five other orderer nodes. Also, the orderer node waits for receiving commit messages from five orderer nodes.

Leader Orderer Node Selection: Similar to Raft, PBFT should avoid the absence of a leader orderer node. PBFT has a mechanism to periodically check the status of the leader orderer node, so-called "view change mechanism." Each orderer node has an index number given at the initialization of the Hyperledger Fabric system. The follower orderer node with the next index number of the leader orderer node monitors the interval between the messages from the leader orderer node. When the interval exceeds a certain time, the follower orderer node assumes that the leader orderer node is faulty. Then the follower orderer node becomes the candidate leader and propagates the "view change" messages to other orderer nodes. Other orderer nodes receiving the "view change" messages send acknowledge (ack) messages to the candidate leader. When the candidate leader receives m acks, it acts as the leader node.

To summarize, Table 1 compares three consensus algorithms—from the perspectives of consensus operations, replication, roles of orderer nodes, fault-tolerable range, and node fault handling. The comparison is the basis for understanding the performance evaluation and analysis results, to be presented in Section V.

IV. EVALUATION SETTINGS

Here, we present the evaluation settings and analysis criteria of this study. As consensus algorithms contain complex operations, we elaborate on how experiments are conducted. The configurable parameters and evaluation metrics are explained as well.

A. EVALUATION SETUP

For the evaluation, we use Hyperledger Fabric v1.4[7] with Kafka, Raft, and PBFT. We run four peer nodes and the

[7]Note that although new versions of Hyperledger Fabric (e.g., v2.1) have been released, the consensus algorithms, structures, and implementations, on which we focus in this study, are identical to the version that we measured.

TABLE 1. Consensus algorithms comparison.

		Kafka	Raft	PBFT
Hyperledger Fabric nodes		Orderer nodes and broker nodes	Orderer nodes	Orderer nodes
Replicated data		Transactions	Blocks	Blocks
Consensus operations		Transaction replication, and block creation	Block creation and block replication	Block replication, block verification, block creation
Role of orderer nodes	Leader	Decision on block creation (sending TTC-X message)	Block creation and block replication	Block creation and sending block in pre-prepare stage
	Follower	Block creation according to the TTC-X message, distribution of the created block to the peer nodes	Distribution of the received block to the peer nodes	Verification on blocks using digital signature, distribution of the received block to the peer nodes
Minimum number of data replication (m)		Independently configured	$m=\lceil(n+1)/2\rceil$	$m=(2n+1)/3$ (n: multiples of three plus one)
Fault-tolerable range		n-m		
Node fault handling		Using Apache Zookeeper	Heartbeat mechanism	View change mechanism

varying number of orderer nodes (Section IV-C2). For Kafka, four broker nodes and three Zookeeper nodes[8] run, which is the minimum configuration for operating Kafka in Hyperledger Fabric. For broker fault experiments, we increase the number of broker nodes up to eight. We use seven physical servers, and each server is equipped with Intel Xeon CPU E5-2650 processor. The nodes are distributed as follows: one server for running peer nodes, one server for broker nodes, one server for zookeeper nodes, and other servers for orderer nodes.

For Raft and PBFT, which do not require broker and Zookeeper nodes, six servers run orderer nodes, and one server runs peer nodes. Each node is packaged using a container with two CPU cores, and these containers run on our cluster of seven physical servers. In addition, one server hosts up to four containers to ensure that the experiments are carried out without any bottlenecks in computing resources. The servers are connected with a 10 Gbps Ethernet switch.

For the workload, we use smallbank chaincode, which has been used in previous studies [26], [37]. Smallbank contains chaincodes that perform account openings, deposits, and money transfers. For all measurements, 20 clients generate transactions and send them to Hyperledger Fabric. The clients run on a 24-core machine, separated from the servers running the Hyperledger Fabric nodes. Each of the 20 clients sends 50 transactions per second; thus, 1000 transactions per second are generated. We ensure that 1000 transactions per second do not impose any bottleneck in the Hyperledger Fabric system, so the performance and resource consumption of Hyperledger Fabric are measured in its "steady-state." In addition to smallbank, we run two other workloads, and their results are presented in Section VII because they exhibit similar tendencies to those of smallbank.

There are two types of endorsement policies: "AND" and "OR." The policy "AND" indicates that the peer nodes in the Hyperledger Fabric system should all sign on each transaction. On the other hand, the policy "OR" indicates that one signature is sufficient. It is known that the "OR" policy incurs fewer bottlenecks on Hyperledger Fabric systems for validation [25]. The focus of this paper is on the evaluation of consensus algorithms, so we select "OR," which reduces the overhead from the validation step.

[8]Zookeeper nodes are used to coordinate multiple broker nodes. Regarding resource consumption, we omit the results of the Zookeeper nodes because they consume less than 1% of the CPU.

B. EVALUATION METRICS

The following evaluation metrics are used to analyze the experiment results.

1) PERFORMANCE

For performance, we use throughput (transactions per second, TPS) and latency (s), which are the key factors and the most widely used indicators of most blockchain systems [38]. Throughput represents the number of successfully committed transactions per second. Latency is the time between the arrival of a transaction from a client at the peer nodes and the transaction being committed to Hyperledger Fabric. Latency contains the entire delay of each transaction (i.e., endorsement latency, broadcast latency, commit latency, and ordering latency [34]). For measurements, we use Caliper [39], which has been used in previous studies on blockchain platforms. TPS and latency are measured at least three times, and unless otherwise noted, averaged values are presented.

2) RESOURCE CONSUMPTION

For the resource consumption, we measure the nodes related to consensus algorithms (i.e., orderer nodes and broker nodes). Specifically, we measure CPU cycle (%) and communication bandwidth (Mbps) of transmission (TX) and reception (RX) on each orderer node and broker node. For the resource consumption measurements, we employ *pidstat* and *vnstat* for the CPU cycle and communication bandwidth, respectively. The resource consumption of orderer nodes and broker nodes represents the average CPU cycle and communication bandwidth measured for one minute.

TABLE 2. Experiment ranges (cases) for configurable parameters.

Experiment set		Parameter			
		Batch size	Node number	Fault node	Payload size
Batch size changes		**10–200**	Minimum[9]	0	10
Node number changes		100	**3–10 orderers**	0	100
Payload size changes		100	Minimum	0	**64–4096**
Fault node changes	Kafka	100	10 brokers	**1–8**	100
	Raft		10 orderers	**1–5**	
	PBFT		10 orderers	**1–4**	

C. CONFIGURABLE PARAMETERS

Hyperledger Fabric has several configurable parameters that affect its performance [40]. We explain the representative configurable parameters in our evaluations. "Batch size" represents the maximum number of transactions that each block can have. "Node number" indicates the number of entire orderer nodes. "Fault node number" refers to the number of fault (crashed) nodes. In addition, "payload size" is the length of transactions. We run four sets of experiments—one set per parameter—and explain how the experiments are carried out below. The total combinations of parameters used in the experiment with is 355. Table 2 summarizes the configurable parameters and the ranges used in the experiments.

1) BATCH SIZE CHANGES

The first set of experiments varies the number of batch sizes from 10 to 200 transactions to investigate the effect of the batch size on the consensus algorithm. When the batch size is set to 1, Kafka, Raft, and PBFT reveal that 10400, 9734, and 12155 transactions are not committed to the ledger (transaction failures). Note that these failures are not caused by faults on the nodes. Because the batch size of 1 means that a block is created per transaction, thereby causing a significant delay in the block generation. Thus, communication between Hyperledger Fabric nodes frequently times out due to the increased delay, which eventually leads to the failure of the Hyperledger Fabric system. Thus, we exclude the batch size of 1 and small batch sizes and set the starting batch size to 10. The first set of experiments is performed without node faults and the minimum number of orderer nodes (i.e., three for Raft and Kafka, respectively, while four for PBFT). Additionally, the payload size of each transaction is set to 10 bytes (Table 2).

2) NODE NUMBER CHANGES

As consensus algorithms depend on orderer nodes, experiments over the varying number of orderer nodes are essential. Thus, the second set of experiments increases the number of orderer nodes from the minimum numbers of nodes (e.g., three, three, and four for Raft, Kafka, and PBFT, respectively) to 10. When multiple orderer nodes are executed, up to three orderer nodes run on the same physical server. This ensures that each orderer node can run without bottlenecks caused by the lack of computing resources, such as CPU or memory on the physical server. For this set of experiments, the batch size is fixed to 100, and the payload size is set to 100 bytes without any fault nodes.

[9]Three orderer nodes, three orderer nodes, and four orderer nodes for Kafka, Raft, and PBFT, respectively.

3) PAYLOAD SIZE CHANGES

The third set of experiments is on the payload size. We vary the payload size from 64 to 4096 bytes. The experiments assume no fault node. The number of orderer nodes is fixed to the minimum number of orderer nodes, while the batch size is set to 100.

4) FAULT NODE CHANGES

The fourth set of experiments measures the performance and resource consumption of the Hyperledger Fabric system while the number of fault nodes increases. Remember that Kafka performs data replication through broker nodes, while Raft and PBFT do so via orderer nodes. This set of experiments, then, uses 10 broker nodes and three orderer nodes for Kafka, while 10 orderer nodes are used for Raft and PBFT. Note that the number of broker nodes in Kafka and orderer nodes in Raft and PBFT is set to 10 because they should perform the same role of data replication.

For faults, we inject failures on broker nodes for Kafka and on orderer nodes for Raft and PBFT. Specifically, we generate node crashes all at once 10 s after transactions begin, as in previous studies [29]. The fault-tolerable ranges of Kafka, Raft, and PBFT are seven, four, and three, respectively (Table 1). Although consensus algorithms are designed to operate within fault-tolerable ranges, the goal of this set of experiments is to observe how the performance and resource consumption are affected when the fault nodes change. Thus, we increase the number of faults of Kafka, Raft, and PBFT to eight, five, and four, respectively, so that our settings cover both inside and outside the fault range. The other configurable parameters are fixed: batch size and payload size are set to be 100 and 100 bytes each.

V. EVALUATION AND ANALYSIS

In this section, we present the evaluation results of the four sets of experiments.

A. BATCH SIZE CHANGES

1) THROUGHPUT

Fig. 7a illustrates the throughput for three consensus algorithms as the batch size increases. We first discuss the results of Kafka and Raft. The transaction rate in the experiments is 1000, which means that 1000 transactions are generated per second. Here, the throughputs of the two algorithms increase by 7% and 6%, respectively, between batch sizes 10 and 100. This is because, with these batch sizes, transactions that arrive at orderer nodes fill the blocks of the batch sizes (i.e., 10 to 100). Thus, the number of generated blocks decreases

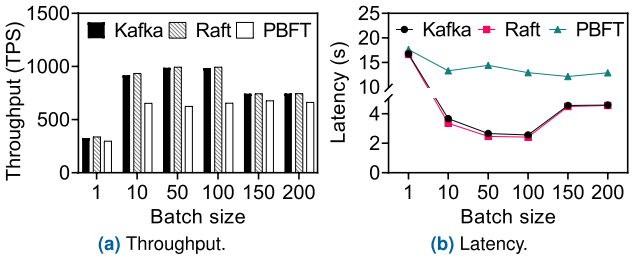


FIGURE 7. Performance when batch size increases.

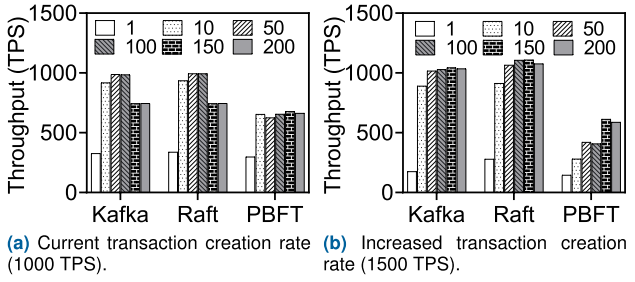


FIGURE 8. Throughput of Kafka and Raft by changing transaction creation rates.

as the batch size increases. Thus, the required processing and latency decrease, which leads to improved throughputs.

When the batch sizes are between 150 and 200, the throughputs of Raft and Kafka decrease by 25% on average (Fig. 7a). We find that the orderer nodes wait for sufficient transactions to arrive to fill the blocks, which causes an additional delay; thus, the throughput is reduced. We experiment to see how the transaction creation rate affects the results by increasing the transaction rate from 1000 to 1500, and the results are presented in Fig. 8b. In contrast to Fig. 7a, the throughput does not decrease with the batch size of 150 and 200.

Looking at the details, our intention in Fig. 8 is to observe how the throughputs of Kafka and Raft change over the transaction rate. Each bar in Fig. 8 represents the throughput for batch sizes from 10 to 200. With the transaction rate of 1000, Fig. 8a shows that both algorithms achieve the highest throughputs at batch sizes of 50 and 100. With the transaction rate of 1500, Fig. 8b shows that both algorithms achieve the highest throughputs at the batch size of 150. In fact, their throughputs between the batch sizes 50 and 200 are quite similar: 1031 and 1089 on average. These results reveal that, as long as enough transactions fill the blocks, the throughputs do not drop. Our finding is that the TPS increase can be achieved with a higher transaction rate with bigger batch size.

Next, we explain the results of PBFT. The throughput values of PBFT (Fig. 7a) range from 625 (batch size 50) to 678 (batch size 150). The difference between the minimum and maximum values is only 5%. Comparing the throughputs of the other two consensus algorithms, PBFT is 25% and 26% lower than those of Kafka and Raft, respectively. This is due to the verification of PBFT in that m orderer nodes (i.e., three in our experiments) should perform verifications on blocks,

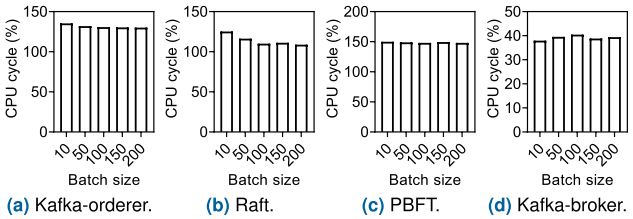


FIGURE 9. CPU cycle when batch size increases.

which is an additional overhead compared to Kafka or Raft (Section III-C1).

2) LATENCY

Fig. 7b displays the latency for the three consensus algorithms as the batch size increases. For Kafka and Raft, the delays decrease from the batch sizes 10 to 100 (3.67 s to 2.56 s in Kafka and 3.36 s to 2.42 s in Raft). The improved delay results from the reduced number of block generations, as discussed in the throughput section.

However, in Kafka and Raft, when the batch size exceeds a threshold (i.e., 100 in Fig. 7b), the delay increases rapidly—1.78× and 1.86×, respectively. We find that this increased delay is due to the orderer nodes waiting for enough transactions to fill the blocks (up to the batch size).

In terms of PBFT, the latency exhibits fluctuations between −8% and +10% from the average latency of 13 ms. These fluctuations are lower than those of Kafka and Raft—up to 20% and 34.7%, respectively. The results indicate that the delay of PBFT is less affected by the batch size than those of Kafka and Raft.

3) CPU CYCLE

Fig. 9 shows the CPU cycles of the consensus algorithms as the batch size increases. The bars in the graph represent the average values of all orderer nodes. From batch size 10 to 200, the average CPU cycles of Kafka, Raft, and PBFT decrease by about 5.3% (Fig. 9a), 16.4% (Fig. 9b), and 1.9% (Fig. 9c), respectively. Although the CPU cycles decrease continuously as the batch size increases, considering the average CPU cycles (131.6%, 114.3%, and 148.9% for Kafka, Raft, and PBFT, respectively), this reduction accounts for a small portion of the total CPU cycles. Thus, the results indicate that the batch size is not a critical parameter for the CPU consumption of consensus algorithms. Because Kafka replicates transactions via brokers, we measure the CPU cycles of the brokers (named Kafka-broker) in addition to the orderer nodes. Fig. 9d shows the results of Kafka-broker where the CPU cycle shows a maximum difference of 3% (basis: batch size 100) from the average CPU cycles (i.e., 39%). This means that the CPU cycles of Kafka-broker also do not change much over the batch size, which is similar to Kafka-orderer.

4) COMMUNICATION BANDWIDTH

Figs. 10a and 10b illustrate the TX and RX of communication bandwidth as the batch size increases. Each bar represents

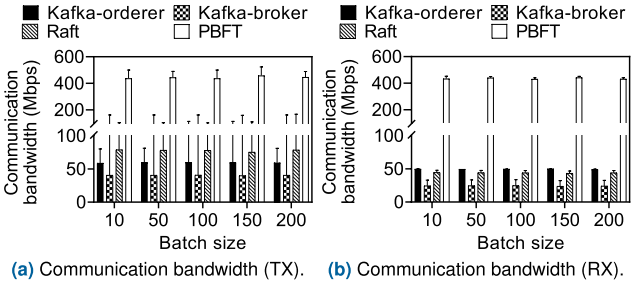


FIGURE 10. Communication bandwidth when batch size increases.

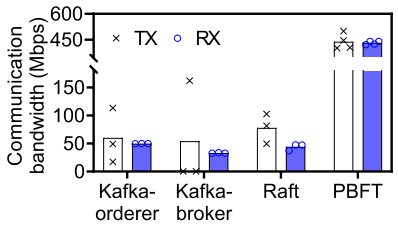


FIGURE 11. Communication bandwidth of individual orderer node when batch size is 100.

the average bandwidth that orderer or broker nodes transmit and receive. The results show that bandwidth does not increase nor decrease significantly over the batch size—the maximum fluctuations of TX bandwidth of Kafka-orderer, Kafka-broker, Raft, and PBFT are 1.2%, 1.2%, 3.3%, and 1.7%, respectively. The fluctuations of RX bandwidth are 0.82%, 2.4%, 1.6%, and 1.1%. The primary reason we find is, even if the batch size increases, the total number of transactions sent and received through the block remains the same. As such, resource consumption does not change significantly.

Next, we compare the TX and RX bandwidth. The TX bandwidth of Kafka-orderer, Kafka-broker, and Raft is higher than the RX bandwidth—1.2×, 1.7×, and 1.8×, respectively. On the other hand, PBFT exhibits quite a similar network bandwidth between TX and RX—in particular, the value of TX is 1.02× higher than RX. We further explore the TX and RX bandwidth of individual orderer nodes. For this experiment, we set all orderer nodes to receive transactions from peer nodes and one leader orderer node.

Fig. 11 shows the individual bandwidth values when the batch size is 100. We omit the results of other batch sizes because the results are similar. The values expressed using x marks represent the TX bandwidth of individual nodes, while those expressed using circle marks represent the RX values. Based on the results, we observe that the individual TX bandwidth demonstrates large deviations compared with RX—the standard deviation values of TX and RX of Kafka-order are 49 and 0.15 (320× difference), respectively. Kafka-broker and Raft likewise demonstrate similar differences between TX and RX (i.e., 163× and 4.5× difference for Kafka-broker and Raft, respectively). This is because the TX bandwidth is primarily used for transaction or block replication between nodes, so it is generally used by the

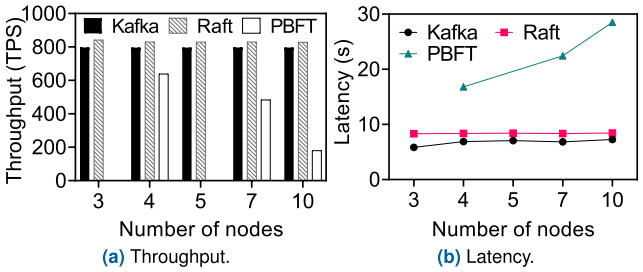


FIGURE 12. Performance when node number increases.

leader orderer node. Thus, the TX bandwidth is consumed dominantly by one node. On the other hand, the RX bandwidth is used by all orderer nodes to receive replicated transactions or blocks, so the RX differences between nodes are not so high.

In terms of PBFT, the TX has higher bandwidth (which results in a 4.7× difference in TX and RX). However, in PBFT, the orderer nodes additionally consume TX bandwidth to send commit messages during the transaction verification, so the TX bandwidth of orderer nodes is relatively higher than that of Kafka-orderer, Kafka-broker, and Raft. Specifically, when comparing the coefficient of variation that shows the standardized dispersion between different data, the TX of Kafka-orderer, Kafka-broker, Raft, and PBFT has values of 0.8, 1.7, 0.34, and 0.1, respectively, indicating that PBFT exhibits a minor deviation.

Regarding Kafka-orderer and Kafka-broker, they exhibit a similar tendency in resource consumption. Hereafter, we present and discuss the measurement results of Kafka-orderer. Kafka-broker results are only shown if their results demonstrate a considerable difference.

B. NODE NUMBER CHANGES

1) THROUGHPUT

Fig. 12a illustrates the throughput of Kafka, Raft, and PBFT according to the number of orderer nodes (x-axis). PBFT requires the orderer nodes to be multiples of three plus one (Section III-C2). So, we set the x-axis as 3, 4, 5, 7, and 10, and for 3 and 5, there are no PBFT results in the figure.

Even though the number of orderer nodes increases, in Fig. 12a, the throughput of Kafka and Raft remains unchanged. The result means that the replication overhead of the consensus algorithm does not increase. The results are explained as follows. For this set of experiments, the batch size is set to 100 so that the number of generated blocks remains the same. Also, the replication is done with the broadcasting of blocks, which takes the same amount of time regardless of the number of orderer nodes.

On the other hand, in PBFT, the throughput decreases significantly as the number of orderer nodes increases. With 10 nodes, the throughput is reduced by 70% compared to that with four nodes. This is because the blocks have to go through verification in the orderer nodes (Section III-C2); thus, more orderer nodes lead to more verification, resulting in a deeper decrease in throughput.

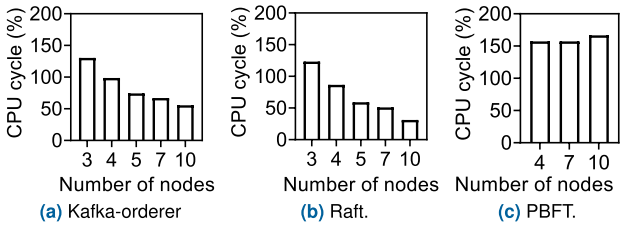


FIGURE 13. CPU cycle when node number increases.

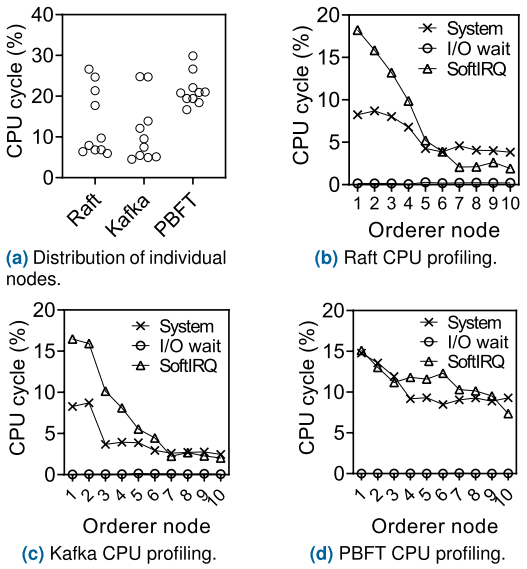


FIGURE 14. CPU profiling results (10 orderer nodes).

2) LATENCY

Fig. 12b shows the latency of the consensus algorithms as the number of orderer nodes increases. Similar to the throughput in Fig. 12a, the latency values of Kafka and Raft remain unchanged. This is because the data replication performed by the leader node of Kafka and Raft is processed with the broadcasting of blocks, regardless of the number of orderer nodes. On the other hand, the latency of PBFT increases by up to 71% (with 10 orderer nodes). We find this is because PBFT requires additional block verifications as the number of orderer nodes increases.

3) CPU CYCLE

Fig. 13 shows the average CPU cycles of orderer nodes as the number of orderer nodes increases. We depict the resource consumption of Kafka-orderer, Raft, and PBFT in Figs. 13a, 13b, and 13c, respectively. We first discuss the results of Kafka and Raft. As the number of orderer nodes increases, the average CPU cycles of Kafka and Raft decrease by 56.4% and 59.8%, respectively. We analyze the reason for the results.

We measure the kernel-level CPU cycle of individual orderer nodes when the number of orderer nodes is 10 (Fig. 14a). Each circle in Fig. 14a indicates the CPU cycles of individual nodes. For Raft, for example, four orderer nodes consume high CPU cycles (23% on average in the y-axis of Fig. 14a), while the remaining orderer nodes have low CPU cycles (near 7%).

We further investigate the reason why four orderer nodes consume the high CPU cycles. We profile orderer nodes using *mpstat* to analyze in detail how each orderer node consumes the CPU cycles. In Fig. 14b, Fig. 14c, and Fig. 14d, we present the kernel-level profile results (y-axis) of Raft, Kafka, and PBFT over orderer nodes 1–10 (x-axis). We classify the CPU cycles of an orderer node as system (line with circles), I/O wait (line with rectangles), and SoftIRQ (line with triangles). ''System'' indicates CPU cycles used in Linux kernel except interrupt processing. Also, ''SoftIRQ'' is the CPU cycles spent on interrupt processing that handles packets. We sort the x-axis of orderer nodes in the order of CPU cycle consumption (e.g., node1 in the x-axis is the highest).

We first discuss the results of Raft. Fig. 14b shows the profiling results of 10 orderer nodes of Raft. Orderer nodes 1–4 demonstrate higher SoftIRQ CPU cycles than the others. This result corresponds to that in Fig. 14a, in which the results of Raft reveal four orderer nodes of high CPU cycles. We find that the four orderer nodes are the orderer nodes responsible for transaction receptions.[10] These nodes perform more network communication than other orderer nodes to deliver transactions from clients. These additional operations generate more packet processing, thus resulting in high CPU cycle consumption. In addition, orderer node 1 acts as the leader orderer node that performs block replication on the follower orderer nodes. The block replication requires network communication, and thereby orderer node 1 shows the highest SoftIRQ.

Fig. 14c shows the profiling results of Kafka. Orderer nodes 1–2 exhibit SoftIRQ CPU cycles higher than the others. This result corresponds to that in Fig. 14a, which indicates that two orderer nodes have high CPU cycles. Similar to Raft (Fig. 14b), we set the number of orderer nodes for transaction receptions as four (orderer nodes 1–4). However, although we set the number of orderer nodes for transaction reception as four, we observe that the Kafka implementation utilizes two nodes for transaction receptions with up to two nodes. So, orderer nodes 1–2 show high CPU cycles relative to orderer nodes 3–4.

Moreover, as with Raft, we find that orderer node 1 of Kafka is also the leader that performs replication. In Kafka, however, the role of the leader is to decide on the timing of block creation, which does not require heavy network communication. Thus, the effect of the leader orderer node's operation on its CPU cycle is small—9% CPU cycle difference between orderer nodes 1 and 2 (in Fig. 14a).

In the remainder of this paper, we denote the orderer nodes that consume higher CPU cycles than the others as ''high orderer nodes'' and the others as ''low orderer nodes.'' Note that the concept of high orderer nodes (i.e., nodes for

[10]We set the number of orderer nodes performing transaction receptions as four because it is the maximum number that the benchmark allows without any errors.

transaction reception and block replication) also exists in PBFT, as discussed below.

We now consider the results of PBFT. In Fig. 13c, PBFT presents increasing CPU cycles (up to 6%) as the orderer nodes increase. Fig. 14a depicts the CPU cycles of individual orderer nodes in PBFT (with 10 orderer nodes).

We perform the profiling shown in Fig. 14d on the PBFT and find the following observations. First, in PBFT, orderer nodes 1–2 consume higher CPU cycles for SoftIRQ than the others (Fig. 14d). These two nodes correspond to the PBFT results shown in Fig. 14a, where two orderer nodes show higher than 180% CPU cycles. We also set four orderer nodes for transaction receptions, but similar to Kafka, only two nodes show particular high SoftIRQ cycles. The orderer node 1 of PBFT is also the leader orderer node, so it consumes 1.16× higher CPU cycle for SoftIRQ than orderer node 2. Also, note that in Fig. 14d, the CPU cycles of "system" are high compared to those of Kafka and Raft. This is because PBFT is implemented using a network file system [24], the kernel component. Thus, for PBFT, all the orderer nodes exhibit high CPU cycles for "system" as well.

Second, the CPU cycles of all orderer nodes of PBFT are higher than those of Raft (Fig. 14b) and Kafka (Fig. 14c)—3.8× and 2.7×, respectively. This is because the orderer nodes of PBFT perform block verifications. PBFT requires each block of at least m numbers of successful verifications (prepared messages) from orderer nodes. When the numbers of orderer nodes are 4, 7, and 10, m becomes 3, 5, and 7, respectively. For example, when the number of orderer nodes is 10, the required number of prepared messages is 7.

4) COMMUNICATION BANDWIDTH

Fig. 15a and Fig. 15b illustrate the communication bandwidth of TX and RX of the orderer nodes. The bars represent the average values, and the box-and-whisker plots represent the range of the communication bandwidth. In Fig. 15a, the average TX of the orderer nodes for Kafka and Raft decreases by 68.1% and 42.9%, as the number of nodes increases from 3 to 10, respectively. On the other hand, the TX of PBFT increases up to 37.3%. In terms of RX (Fig. 15b), the bandwidth of the orderer nodes for Kafka and Raft decrease by 22.9% (from 51 Mbps to 39 Mbps) and 18% (from 50 Mbps to 41 Mbps), respectively, while that of PBFT increases by 43.3%. The results show that Kafka consumes the lowest network bandwidth. As the tendencies of the bandwidth changes of TX and RX are similar, we discuss the reasons for TX in detail.

We investigate the average TX bandwidth values of high and low orderer nodes using lines with x marks and lines with circle marks, respectively, in Fig. 16. The TX bandwidth values of high orderer nodes in Kafka, Raft, and PBFT increase by up to 1.06×, 1.42×, and 1.5× as the number of orderer nodes increases. On the other hand, for the low orderer nodes, the TX bandwidth values of Kafka and Raft are almost constant (ranging from 0.15 to 0.4 Mbps), but the value of PBFT changes from 550 to 514 Mbps, which

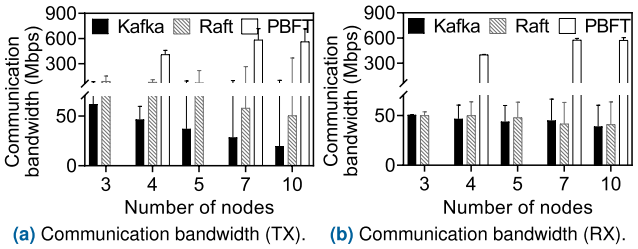


FIGURE 15. Average communication bandwidth when node number increases.

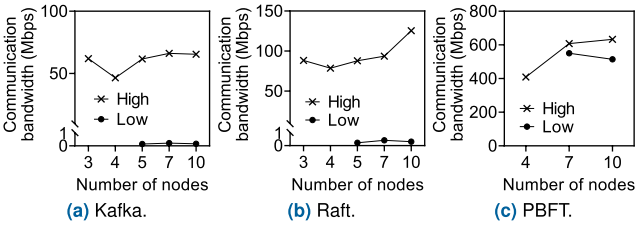


FIGURE 16. Average TX communication bandwidth per high and low orderer nodes.

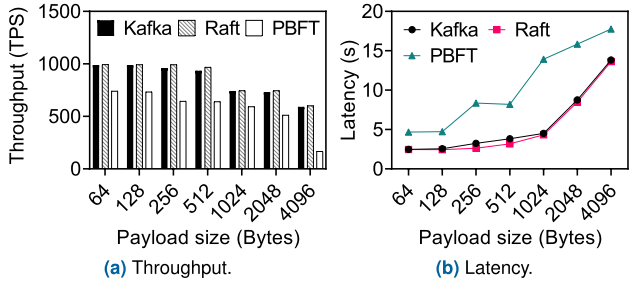


FIGURE 17. Performance when payload size increases.

is much higher than the values in Kafka and Raft. Moreover, the average TX bandwidth values of high orderer nodes are 335×, 290×, and 1.03× higher than those of low orderer nodes for Kafka, Raft, and PBFT, respectively.

Based on the TX bandwidth of Fig. 16, we explain the results shown in Fig. 15. The number of high orderer nodes remains constant for Kafka and Raft. As such, when the number of orderer nodes increases, the number of low orderer nodes increases. Then, for Kafka and Raft, the average TX bandwidth values in Fig. 15 decrease as the low orderer nodes show considerably lower TX bandwidth than the high orderer nodes. On the other hand, for PBFT, the low orderer nodes show similar TX bandwidth to high orderer nodes, so the TX bandwidth values of PBFT in Fig. 15 increase.

C. PAYLOAD SIZE CHANGES

1) THROUGHPUT

Fig. 17a illustrates the results of the third set of experiments—the throughput of the three consensus algorithms as the payload size increases (the parameters listed in Table 2). The results show that the throughput is the highest at the smallest payload size, 64 bytes for all three algorithms.

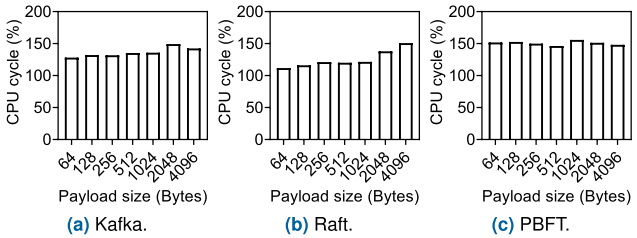


FIGURE 18. CPU cycle when payload size increases.

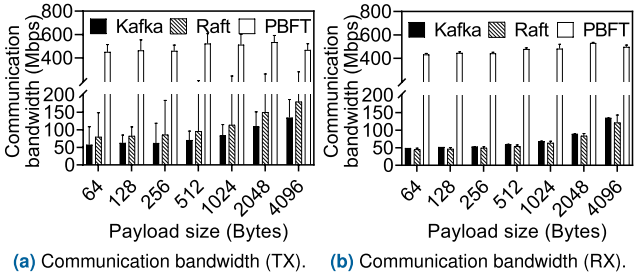


FIGURE 19. Average communication bandwidth when payload size increases.

As the payload size increases, the throughputs of Kafka, Raft, and PBFT gradually decrease—by up to 40.3%, 39%, and 77.5% at the 4K payload size, respectively. When the payload size increases, we do not change the transaction rate and batch size (the number of transactions per block). Thus, the block size increases as the payload size increases. Note that the number of generated blocks is the same regardless of payload size, but the increased block size leads to a gradual decrease in throughput.

Notably, the results also indicate that the PBFT throughput decreases more sharply than those of Kafka and Raft because PBFT performs more processing on blocks (i.e., block verification from all orderer nodes). Thus, the impact of the increased payload size on the throughput of PBFT is higher.

2) LATENCY

Fig. 17b shows the latency as the payload size increases. Comparing the latency between 64 and 4096 bytes, the latency values increase by up to 5.6×, 5.5×, and 3.8× for Kafka, Raft, and PBFT, respectively. The reason for this increased delay is the larger block sizes, thereby prolonging the time for replication and block generation.

3) CPU CYCLE

Figs. 18a, 18b, and 18c show the average CPU cycles of orderer nodes of Kafka, Raft, and PBFT, respectively. As the payload size increases from 64 B to 4 KB, Kafka CPU cycles increase by 11.3% (Fig. 18a) and those of Raft increase by 35% (Fig. 18b). These results indicate that the increased payload size consumes more CPU cycles for orderer nodes. However, for PBFT (Fig. 18c), the CPU cycles are relatively constant—2% difference over the payload size.

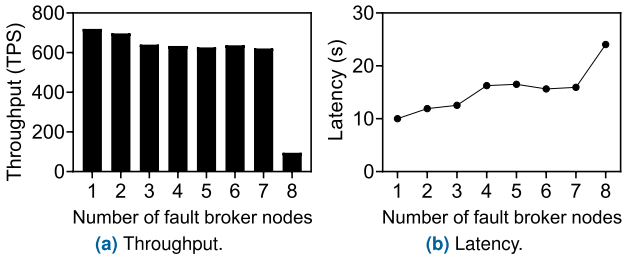


FIGURE 20. Performance of Kafka when fault nodes increase.

4) COMMUNICATION BANDWIDTH

Figs. 19a and 19b present the average TX and RX bandwidth of orderer nodes when the payload size increases. The TX bandwidth (Fig. 19a) of Kafka and Raft increase by 134.6% and 125.8%, respectively, as the payload size increases from 64B to 4096B. On the other hand, the TX values of PBFT are relatively constant—an average of 488 Mbps, with fluctuations of −36.5 Mbps and +47.2 Mbps. In terms of the RX bandwidth (Fig. 19b), Kafka and Raft show continuously increasing values by 174.5% and 168.7%, respectively. The values of PBFT are relatively constant—an average of 4873 Mbps, with fluctuations of −40 Mbps and +57 Mbps. These results are because the size of transactions to be transmitted or received by orderer and broker nodes increases as the payload size increases.

D. FAULT BROKER NODE CHANGES–KAFKA

1) THROUGHPUT

For the fourth set of experiments, the number of fault nodes is varied. These experiments are divided for Kafka, whose results are discussed in this section, and for Raft and PBFT, which are discussed in Section V-E. Since Kafka uses broker nodes for replication, the x-axis of Fig. 20 displays the number of fault broker nodes. With 10 broker nodes, the fault-tolerable range becomes 7, since the default value of m is 3. Fig. 20a shows the throughput when the number of fault broker nodes increases. First, as the number of fault broker nodes increases from one to seven, the throughput decreases by 13.7%.

The throughput decrease is due to Zookeeper. When a broker node fails, Zookeeper finds another broker node to take the place of the failed broker node for data replication and updates the metadata used for data replication between broker nodes and orderer nodes. Then, the Kafka algorithm performs a remapping between broker nodes and orderer nodes for data replication. Therefore, whenever broker nodes fail, remapping between broker nodes and orderer nodes occurs [22], which causes an additional delay in processing transactions so that the throughput decreases.

2) LATENCY

Fig. 20b illustrates the latency. From one to seven fault nodes, the latency increases by 19% due to the remapping caused by broker faults, as explained above. When the number

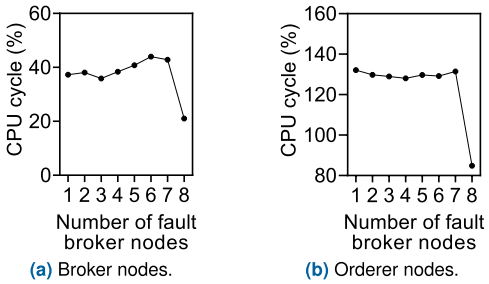


FIGURE 21. CPU cycle of Kafka when fault nodes increase.

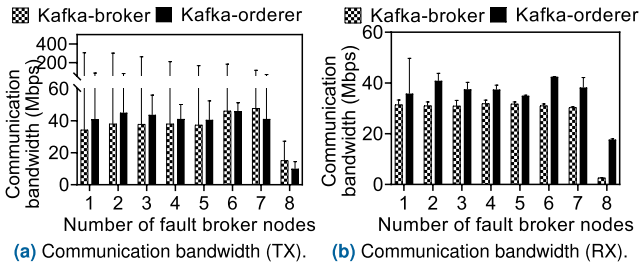


FIGURE 22. Average communication bandwidth of Kafka when fault nodes increase.

of fault broker nodes increases from seven to eight, the latency increases by 51%. This is because Kafka ceases to operate when the number of fault broker nodes exceeds the fault-tolerable range of seven. In Fig. 20a, when eight broker nodes fail, the throughput drove down by up to 87%.

3) CPU CYCLES

We measure the CPU cycles of broker nodes and orderer nodes when the number of fault broker nodes varies. Fig. 21a depicts the average CPU cycles of the broker nodes without faults (normal) in the y-axis. Fig. 21b displays the average CPU cycles of all orderer nodes when the number of fault broker nodes increase. Note that orderer nodes do not have any faults.

In Fig. 21a, the normal broker nodes consume approximately 40% CPU cycles. The CPU cycles of normal broker nodes increase up to 1.14× until the number of fault broker nodes is seven. Then with eight fault broker nodes, they drop to 21%. This is because Kafka ceases its operation when the number of fault broker nodes increases to eight. In Fig. 21b, the CPU cycles of orderer nodes are steady within the fault-tolerable range of seven (129% on average) as the orderer nodes perform push and pull operations.

4) COMMUNICATION BANDWIDTH

Fig. 22a and Fig. 22b show the average communication bandwidth TX and RX of normal Kafka-broker and Kafka-orderer nodes. When the number of fault broker nodes increases within the fault-tolerable range, the TX bandwidth of broker nodes shows fluctuations around the average value of 39.9 Mbps (−5.6 Mbps to +7.8 Mbps). For Kafka-orderer, TX is 42.6 Mbps on average, and the fluctuations are in the range of −2 Mbps to +3.3 Mbps. Additionally, the RX bandwidth of broker nodes exhibits 31 Mbps as the average, with 1.37% average fluctuation (2.9% as the maximum fluctuation at seven fault broker nodes). Meanwhile, Kafka-orderer has an average of 38 Mbps with fluctuations of 5% (maximum fluctuation of 10% at six fault broker nodes). In this experiment, the number of transactions remains constant so that the communication bandwidth does not have a high correlation with the number of fault broker nodes.

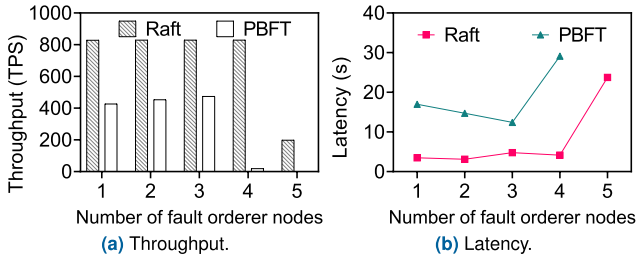


FIGURE 23. Performance when the fault orderer nodes increase.

E. FAULT ORDERER NODE CHANGES—RAFT AND PBFT

1) THROUGHPUT

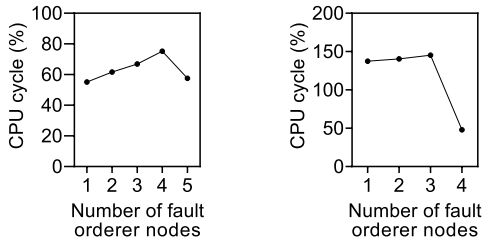
For Raft and PBFT, Fig. 23 presents the throughput and latency when the number of fault nodes increases. We set the x-axis to be 1 to 5 because, with 10 orderer nodes, Raft tolerates up to four fault orderer nodes, while PBFT tolerates up to three fault orderer nodes. Fig. 23a shows that within the fault-tolerable range, the throughputs of Raft and PBFT do not deteriorate.

Notably, we observe that the throughput of PBFT increases by up to 11% (807.6 to 827.8) when the number of fault orderer nodes increases from one to three. This is because PBFT requires that all orderer nodes verify the blocks. Thus, the amount of verification in PBFT decreases as the number of fault orderer nodes increases, resulting in a moderate increase in throughput.

When the number of fault orderer nodes exceeds the fault-tolerable range, the throughput values of Raft drops to 199 TPS from 829 TPS and of PBFT to 20 TPS from 427 TPS, which are reductions of 76% and 95.8%, respectively. This sharp decline is consistent with Fig. 20, where the throughput of Kafka drops to 95 TPS, which is an 87% reduction. By comparing the three algorithms, PBFT shows the sharpest decrease in TPS.

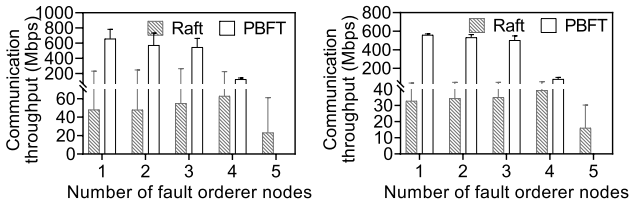
2) LATENCY

In Fig. 23b, the PBFT latency decreases when the number of fault nodes increases within its fault-tolerable range (up to three nodes). Comparing the latency between the faults of one node and three nodes, the latency decreases by up to 27%. This is because only normal orderer nodes participate in block verification. Thus, when the number of orderer nodes decreases, this reduces block verification. The Raft latency remains quite steady (i.e., 3.8 s).



(a) Raft. (b) PBFT.

FIGURE 24. CPU cycle of Raft and PBFT when fault orderer nodes increase.



(a) Communication bandwidth (TX). (b) Communication bandwidth (RX).

FIGURE 25. Average communication bandwidth of Raft and PBFT when fault orderer nodes increase.

When the number of fault orderer nodes exceeds the fault-tolerable ranges, the latencies of Raft and PBFT increase by up to 5.7× and 2.3×, respectively. Similar to TPS, PBFT demonstrates the most rapid increase in latency.

3) CPU CYCLE

Fig. 24 illustrates the CPU cycles of the Raft and PBFT over the number of fault nodes. We mark the average of the normal orderer nodes. The CPU cycles of Raft (Fig. 24a) increase up to 1.36× within the fault-tolerable range and drop significantly (0.7×). Note that the TPS of Raft is constant within the fault-tolerable range (829.2 on average) in Fig. 23a. This means that the numbers of transactions successfully committed are identical, although the number of fault nodes increases within the fault-tolerable range. As the number of fault nodes increases, the number of normal nodes decreases. Then, the number of transactions that each normal node should process increases. So, the average CPU cycle of the individual orderer node increases.

In terms of PBFT (Fig. 24b), the CPU cycles increase up to 1.1× within the fault-tolerable range and then drop (0.6×). PBFT shows increasing TPS (11%, Fig. 23a) within the fault-tolerable range. So, the number of transactions that each normal orderer node processes increases as the number of fault nodes increases; therefore, the CPU cycles increase. However, because block verification is reduced as the number of fault nodes increases, the rate of CPU increase gradually increases.

4) COMMUNICATION BANDWIDTH

Figs. 25a and 25b display the average communication bandwidth TX and RX, respectively. For Raft, when the number of fault orderer nodes increases within the fault-tolerable range (from one to four), the average communication bandwidth of TX decreases from 43 Mbps to 37 Mbps (12% decrease). The value of RX decreases from 29 Mbps to 26 Mbps (11% decrease). For PBFT, within the fault-tolerable range, TX decreases from 597 Mbps to 411 Mbps (31% decrease), while RX decreases from 513 Mbps to 380 Mbps (26% decrease).

F. SUMMARY

We summarize the major findings from our comprehensive evaluation as follows:

- From the resource provisioning perspective, Raft consumes fewer CPU than Kafka and PBFT. For the network, Kafka consumes the least bandwidth. So, in a given Hyperledger Fabric configuration, Raft achieves higher TPS than Kafka and PBFT. Overall, PBFT has more idle time than the others due to the block verification of orderer nodes.
- In terms of configurable parameters, the batch size affects the performance (Section V-A1). For example, the throughputs of Kafka and Raft increase as the batch sizes increase but decrease at a certain threshold. In our experiments, Kafka and Raft exhibit their best throughput at batch sizes of 100 and 150 when the transaction creation rates are 1000 and 1500, respectively. So, when a target TPS is given, it is important to find the batch size threshold to maximize the Hyperledger Fabric configuration.
- High orderer nodes consume more CPU cycles than low orderer nodes (Section V-B3). CPU schedulers like Linux CFS demote such high-CPU consuming tasks for fair CPU sharing, which means that high orderer nodes are scheduled less frequently. So, if a new CPU scheduler is designed to take care of high orderer nodes, it would be possible to increase TPS further. We leave this as future research work for the Blockchain community.
- Within the fault-tolerable range, latency remains stable for Kafka and Raft, while the latency of PBFT decreases by up to 27% when the fault nodes increase (Section V-D1 and Section V-E1). This is because the orderer nodes participating in block verification and their associated network communication decrease. So, it is beneficial to operate only m orderer nodes for BaaS services.

VI. RELATED WORK

Considering the importance of consensus algorithms in blockchain systems, various studies have investigated consensus algorithms. Previous studies are summarized in Table 3 and compared with this study. We categorize such studies into two: 1) performance evaluation and 2) algorithm improvement. We summarize the main differences of this study compared to the existing studies are as follows.

First, in terms of the analysis metrics, this study is the first to analyze the resource consumption. Considering that

TABLE 3. Related work comparison.

Category	Study	Platform	Consensus algorithm	Analysis metric				Configurable parameter				Environment	
				Performance		Resource consumption		Fault nodes	Number of nodes	Batch size	Payload size	Workload	Cluster
				Throughput	Latency	CPU cycle	Communication bandwidth						
Performance evaluation	[25]	Hyperledger Fabric 1.4	Kafka, Raft	◯	◯	×	×	×	◯ (3, 9, 12)	×	×	Custom (1 B)	20 machines
	[26]	Hyperledger Fabric 1.4	Kafka, Raft	◯	×	×	×	×	×	×	×	Caliper (smallbank)	N/A
	[27]	Simulation	Raft	×	×	×	×	×	◯(3, 5, 41, 101)	×	×	Custom	N/A (simulation)
	[28]	Simulation	PBFT	◯	×	×	×	×	◯(5, 10, 15, 20, 25)	×	×	Custom	N/A (simulation)
	[29]	Hyperledger Fabric 1.4, Iroha	Raft, YAC	×	×	×	×	◯ (1, 2, 3)	×	×	×	NA	Single node
	[30]	Quorum	Raft, IBFT-Quorum	◯	×	×	×	×	×	×	◯(1, 10, 20, 30 KB)	Quorum	3 machines
Algorithm improvement	[31]	Hyperledger Fabric 1.2	Kafka, BLOXY+ PBFT, BFT-SMaRt	◯	◯	×	×	×	×	×	×	Caliper (simple)	4 machines
	[32]	Hyperledger Fabric (Version N/A)	BDRBFT, PBFT, FDRBFT	◯	◯	×	×	◯ (1, 2)	◯ (1, 2)	◯ (1, 30, 350, 400)	×	Caliper (simple)	N/A
	[34]	Hyperledger Fabric (Version N/A)	BFT-SMaRt	◯	◯	×	×	×	◯(4, 7, 10)	×	◯(40, 200 B, 1, 4 KB)	Custom (1, 4 KB)	3 machines
This study		Hyperledger Fabric 1.4	Kafka, Raft, PBFT	◯	◯	◯	◯	◯ (1, 2, 3, 4, 5, 6, 7, 8)	◯(3, 4, 5, 7, 10)	◯(10, 50, 100, 150, 200)	◯(64, 128, 256, 512, 1024, 2048, 4096 B)	Caliper (smallbank)	7 machines

blockchain services are widely deployed in the cloud as the form of BaaS, understanding resource consumption is critical. Especially, consensus algorithms deploy multiple orderer (or broker) nodes, which tradeoffs resource consumptions with system reliability. However, to the best of our knowledge, none of the previous studies have analyzed the increased amount of computational resources. Our study provides the first fluent analysis on resource consumptions—CPU cycle and communication bandwidth.

Second, regarding the system parameter, existing studies changed only one or two types of system parameters, indicating limited parts of system variations were identified. As the Section V in this paper presented, the performance and resource consumption show various patterns upon the parameters, so this study differs from existing studies in that providing knowledge on performance and resource consumption on BaaS services with comprehensive configurable parameters. The evaluation cases that this paper covers are about 355 cases, which is the most comprehensive and diverse as far as we know.

Third, our study reveals the performance and resource consumption under practical node faults. From the nine existing studies, only two studies have generated node faults for their evaluations. Thus, the comparison between the widely deployed consensus algorithms is missing in the existing literature. In addition, both the existing number of orderer nodes and the fault nodes in previous studies were only two or three. Consensus algorithms allow a user to increase the number of orderer nodes flexibly to configure the fault-tolerable range (Section III). Hence, it is natural to analyze to understand how the performance and resource consumption change depending on the desired fault tolerance. Moreover, there has been no practical analysis of fluctuations in the resource consumption in situations when the number of faults exceeds the fault-tolerable range.

Apart from the studies mentioned above, other studies that profiled the Hyperledger Fabric system have been conducted [41]–[44]. However, the papers analyzed the validation steps among the three operations in Hyperledger Fabric systems (i.e., execution, ordering, and validation). Compared to the validation step with a fluent and comprehensive analysis, the consensus step has not been investigated comprehensively, which also motivates us to profile the consensus step thoroughly.

VII. DISCUSSION

A. SELECTION OF BaaS PLATFORM AND CONSENSUS ALGORITHMS

We explain why we selected Hyperledger Fabric and the three consensus algorithms as the targets of analysis. The purpose of this paper is to provide a comprehensive analysis 1) of the practical BaaS platform and 2) on both performance and resource consumption of consensus algorithms. First, to select practical BaaS platform, we investigated the existing platforms for running BaaS. Notably, most existing clouds (e.g., Alibaba Cloud, IBM, Microsoft, and AWS) offer BaaS with Hyperledger Fabric as the underlying platform [45]. In academia, several studies on BaaS platforms have been

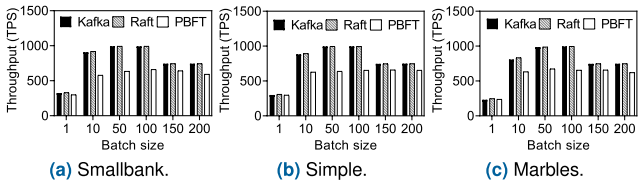


FIGURE 26. Throughput when batch size changes between workloads.

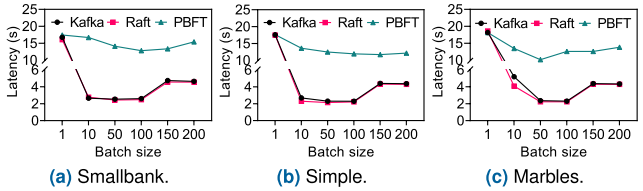


FIGURE 27. Latency when batch size changes between workloads.

conducted, such as uBaaS [46] and FBaaS [47]. However, such studies aimed to unify BaaS in diverse clouds [46] or enable BaaS in serverless computing [47]. In addition, these platforms cannot be profiled because they are not disclosed. Thus, we chose Hyperledger Fabric as the focus of this study.

Next, we reviewed consensus algorithms in Hyperledger Fabric for measuring both performance and resource consumption. First, Hyperledger Fabric officially supports two consensus algorithms, Kafka and Raft [48]. Second, there are other consensus algorithms (e.g., PBFT, proof-of-work, delegated proof of stake, and ripple protocol consensus algorithm). We chose PBFT because it is the base algorithm that serves as the foundation of other algorithms, such as spinning BFT and redundant BFT [49]. Implementing a consensus algorithm for Fabric requires non-trivial effort because it has to be interoperable with internal components within Hyperledger Fabric [50]. Thus, rather than analyzing consensus algorithms that require non-trivial system tuning, we focus on the comprehensiveness in analyzing three key algorithms, which has not yet been presented in the literature. In future work, we plan to analyze other BaaS platforms and consensus algorithms using the methodology in this study.

B. ANALYSIS OF OTHER CHAINCODES

Although smallbank has been frequently used in previous studies, other chaincodes may have different characteristics of performance and resource consumption. Here, we conduct experiments with other representative chaincodes. The chaincodes used in the experiment are simple and marbles [39]. Simple chaincode is a simplified version of the smallbank. Marbles is a workload that stores client asset information (e.g., owner, asset type, id, etc.).

For simple and marbles chaincodes, we conduct experiments on batch size changes (with the same settings as Section IV.B). We compare the results with those of the smallbank. Fig. 26 shows the throughput of the three workloads. In each figure, the throughput values of three consensus algorithms are illustrated as the batch size increases. The

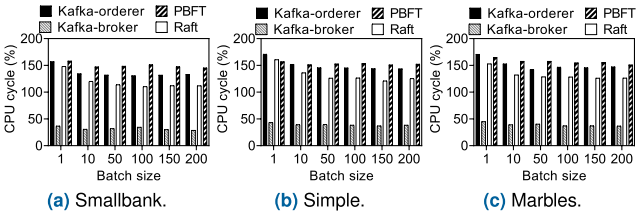


FIGURE 28. CPU cycle when batch size changes between workloads.

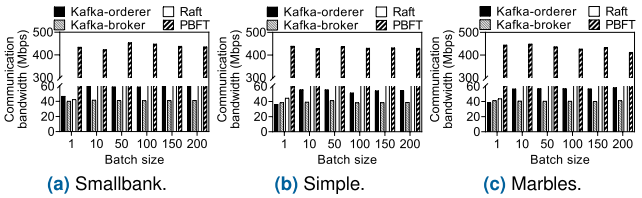


FIGURE 29. Average communication bandwidth (TX) when batch size changes between workloads.

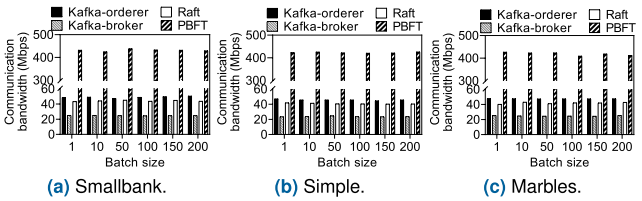


FIGURE 30. Average communication bandwidth (RX) when batch size changes between workloads.

three graphs of smallbank (Fig. 26a), simple (Fig. 26b), and marbles (Fig. 26c) show quite similar results. To judge the similarity objectively, we calculate the cosine similarity of the measurement results of simple and marbles workloads to those of smallbank. In general, the cosinfig:e similarity is in the range of -1 to 1, and the closer to 1, the two data are more similar. The cosine similarities of simple and marbles compared with smallbank are 0.9996 and 0.998, respectively. This means that the results are very similar to those of smallbank; so, we believe that the results and analyses in Section V are applicable for simple and marbles chaincodes.

Additionally, we present the experiment results for the latency (Fig. 27), CPU cycle (Fig. 28), and communication bandwidth (TX in Fig. 29 and RX in Fig. 30). For the latency, the cosine similarity values of simple and marbles compared to smallbank are 0.998 and 0.994, respectively. The cosine similarities for CPU cycle and communication bandwidth values also exceed 0.99 for both simple and marbles, which indicates that the results and analyses of smallbank are quite similar to those of simple and marbles.

VIII. CONCLUSION

This study presents comprehensive analyses on the consensus algorithms for running BaaS in clouds. State-of-the-art studies on BaaS exhibit severe limitations, such as 1) lack of resource consumption analysis, 2) inclusion of only one or two configurable parameters, and 3) quite a limited number of faults (e.g., two to three) in evaluating systems.

To address these limitations, we conduct comprehensive experiments with variations on the number of orderer nodes, batch sizes, payload sizes, and fault nodes. To the best of our knowledge, consensus algorithms have not yet been investigated at this level of details. Based on the experiment results, we provide key findings on 1) resource consumption of the three consensus algorithms, 2) the effect of block size and block creation rate on performance, and 3) the existence of high and low orderer nodes in terms of CPU cycles. This study contributes to understanding the characteristics of resource consumption, which we hope that enables proper provisioning of consensus algorithms on BaaS.

REFERENCES

[1] T. Aste, P. Tasca, and T. D. Matteo, "Blockchain technologies: The foreseeable impact on society and industry," *Computer*, vol. 50, no. 9, pp. 18–28, Jan. 2017.

[2] W. Wang, D. T. Hoang, P. Hu, Z. Xiong, D. Niyato, P. Wang, Y. Wen, and D. I. Kim, "A survey on consensus mechanisms and mining strategy management in blockchain networks," *IEEE Access*, vol. 7, pp. 22328–22370, 2018.

[3] *Blockchain Technology and Applications | Microsoft Azure*. Accessed: Sep. 4, 2021. [Online]. Available: https://azure.microsoft.com/en-in/solutions/blockchain/

[4] *AWS Blockchain Partner Spotlight*. Accessed: Mar. 12, 2020. [Online]. Available: https://aws.amazon.com/partners/blockchain/

[5] *How IBM Blockchain Technology Powers Digital Health Pass | IBM*. Accessed: Aug. 11, 2021. [Online]. Available: https://www.ibm.com/watson/health/resources/digital-health-pass-blockchain-explained/

[6] *Singapore Exchange Case Study*. Accessed: Feb. 19, 2021. [Online]. Available: https://aws.amazon.com/ko/solutions/case-studies/singapore-exchange-case-study/?c=bl&sec=cs4

[7] M. Hu, T. Shen, J. Men, Z. Yu, and Y. Liu, "CRSM: An effective blockchain consensus resource slicing model for real-time distributed energy trading," *IEEE Access*, vol. 8, pp. 206876–206887, 2020.

[8] M. A. Bouras, Q. Lu, S. Dhelim, and H. Ning, "A lightweight blockchain-based IoT identity management approach," *Future Internet*, vol. 13, no. 2, p. 24, Jan. 2021. [Online]. Available: https://www.mdpi.com/1999-5903/13/2/24

[9] E. Bandara, D. Tosh, P. Foytik, S. Shetty, N. Ranasinghe, and K. De Zoysa, "Tikiri—Towards a lightweight blockchain for IoT," *Future Gener. Comput. Syst.*, vol. 119, pp. 154–165, Jun. 2021.

[10] D. Miehle, D. Henze, A. Seitz, A. Luckow, and B. Bruegge, "PartChain: A decentralized traceability application for multi-tier supply chain networks in the automotive industry," in *Proc. IEEE Int. Conf. Decentralized Appl. Infrastruct. (DAPPCON)*, Apr. 2019, pp. 140–145.

[11] G. Yang, B.-Y. Yu, H. Jin, and C. Yoo, "Libera for programmable network virtualization," *IEEE Commun. Mag.*, vol. 58, no. 4, pp. 38–44, Apr. 2020.

[12] Y. Jararweh, M. Al-Ayyoub, E. Benkhelifa, M. Vouk, and A. Rindos, "Software defined cloud: Survey, system and evaluation," *Future Gener. Comput. Syst.*, vol. 58, pp. 56–74, May 2016.

[13] Y. Duan, G. Fu, N. Zhou, X. Sun, N. C. Narendra, and B. Hu, "Everything as a service (XaaS) on the cloud: Origins, current and future trends," in *Proc. IEEE 8th Int. Conf. Cloud Comput.*, Jun. 2015, pp. 621–628.

[14] Y. Yoo, G. Yang, M. Kang, and C. Yoo, "Adaptive control channel traffic shaping for virtualized SDN in clouds," in *Proc. IEEE 13th Int. Conf. Cloud Comput. (CLOUD)*, Oct. 2020, pp. 22–24.

[15] G. Yang, C. Shin, Y. Yoo, and C. Yoo, "A case for SDN-based network virtualization," in *Proc. 29th Int. Symp. Modeling, Anal., Simulation Comput. Telecommun. Syst. (MASCOTS)*, Nov. 2021, pp. 1–8.

[16] E. Androulaki, A. Barger, V. Bortnikov, and C. Cachin, "Hyperledger fabric: A distributed operating system for permissioned blockchains," in *Proc. 13th EuroSys Conf.*, 2018, pp. 1–15.

[17] T. T. A. Dinh, J. Wang, G. Chen, R. Liu, B. C. Ooi, and K.-L. Tan, "BLOCKBENCH: A framework for analyzing private blockchains," in *Proc. ACM Int. Conf. Manage. Data*, 2017, pp. 1085–1100, doi: 10.1145/3035918.3064033.

[18] Y. Hao, Y. Li, X. Dong, L. Fang, and P. Chen, "Performance analysis of consensus algorithm in private blockchain," in *Proc. IEEE Intell. Vehicles Symp. (IV)*, Jun. 2018, pp. 280–285.

[19] S. Pongnumkul, C. Siripanpornchana, and S. Thajchayapong, "Performance analysis of private blockchain platforms in varying workloads," in *Proc. 26th Int. Conf. Comput. Commun. Netw. (ICCCN)*, Jul. 2017, pp. 1–6.

[20] D. K. Meena, R. Dwivedi, and S. Shukla, "Preserving patient's privacy using proxy Re-encryption in permissioned blockchain," in *Proc. 6th Int. Conf. Internet Things: Syst., Manage. Secur. (IOTSMS)*, Oct. 2019, pp. 450–457.

[21] R. Arenas and P. Fernandez, "CredenceLedger: A permissioned blockchain for verifiable academic credentials," in *Proc. IEEE Int. Conf. Eng., Technol. Innov. (ICE/ITMC)*, Jun. 2018, pp. 1–6.

[22] J. Kreps, N. Narkhede, and J. Rao, "Kafka: A distributed messaging system for log processing," in *Proc. NetDB*, vol. 11, 2011, pp. 1–7.

[23] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, 2014, pp. 305–319.

[24] M. Castro B. Liskov, "Practical byzantine fault tolerance," in *Proc. OSDI*, vol. 99, 1999, pp. 173–186.

[25] C. Wang and X. Chu, "Performance characterization and bottleneck analysis of hyperledger fabric," in *Proc. IEEE 40th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Nov. 2020, pp. 1281–1286.

[26] H. Yusuf and I. Surjandari, "Comparison of performance between Kafka and Raft as ordering service nodes implementation in hyperledger fabric," *Int. J. Adv. Sci. Technol.*, vol. 29, no. 7s, pp. 3549–3554, 2020.

[27] D. Huang, X. Ma, and S. Zhang, "Performance analysis of the raft consensus algorithm for private blockchains," *IEEE Trans. Syst., Man, Cybern., Syst.*, vol. 50, no. 1, pp. 172–181, Jan. 2020.

[28] Y. Meshcheryakov, A. Melman, O. Evsutin, V. Morozov, and Y. Koucheryavy, "On performance of PBFT blockchain consensus algorithm for IoT-applications with constrained devices," *IEEE Access*, vol. 9, pp. 80559–80570, 2021.

[29] B. Podgorelec, V. Kersic, and M. Turkanovic, "Analysis of fault tolerance in permissioned blockchain networks," in *Proc. XXVII Int. Conf. Inf., Commun. Autom. Technol. (ICAT)*, Oct. 2019, pp. 1–6.

[30] A. Baliga, I. Subhod, P. Kamat, and S. Chatterjee, "Performance evaluation of the quorum blockchain platform," 2018, *arXiv:1809.03421*.

[31] S. Rusch, K. Bleeke, and R. Kapitza, "Bloxy: Providing transparent and generic BFT-based ordering services for blockchains," in *Proc. 38th Symp. Reliable Distrib. Syst. (SRDS)*, Oct. 2019, pp. 305–30509.

[32] A. Song, J. Wang, W. Yu, Y. Dai, and H. Zhu, "Fast, dynamic and robust byzantine fault tolerance protocol for consortium blockchain," in *Proc. IEEE Int. Conf. Parallel Distrib. Process. Appl., Big Data Cloud Comput., Sustain. Comput. Commun., Social Comput. Netw. (ISPA/BDCloud/SocialCom/SustainCom)*, Dec. 2019, pp. 419–426.

[33] S. Biswas, K. Sharif, F. Li, S. Maharjan, S. P. Mohanty, and Y. Wang, "PoBT: A lightweight consensus algorithm for scalable IoT business blockchain," *IEEE Internet Things J.*, vol. 7, no. 3, pp. 2343–2355, Mar. 2020.

[34] P. Thakkar, S. Nathan, and B. Viswanathan, "Performance benchmarking and optimizing hyperledger fabric blockchain platform," in *Proc. IEEE 26th Int. Symp. Modeling, Anal., Simulation Comput. Telecommun. Syst. (MASCOTS)*, Sep. 2018, pp. 264–276.

[35] W. Viriyasitavat and D. Hoonsopon, "Blockchain characteristics and consensus in modern business processes," *J. Ind. Inf. Integr.*, vol. 13, pp. 32–39, Mar. 2019.

[36] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "ZooKeeper: Wait-free coordination for internet-scale systems," in *Proc. USENIX Annu. Tech. Conf.*, 2010, vol. 8, no. 9, pp. 1–14.

[37] J. Kim, K. Lee, G. Yang, K. Lee, J. Im, and C. Yoo, "QiOi: Performance isolation for hyperledger fabric," *Appl. Sci.*, vol. 11, no. 9, p. 3870, Apr. 2021.

[38] A. Baliga, N. Solanki, S. Verekar, A. Pednekar, P. Kamat, and S. Chatterjee, "Performance characterization of hyperledger fabric," in *Proc. Crypto Valley Conf. Blockchain Technol. (CVCBT)*, Jun. 2018, pp. 65–74.

[39] *Hyperledger/Caliper: A Blockchain Benchmark Framework to Measure Performance of Multiple Blockchain Solutions*. Accessed: Nov. 2, 2020. [Online]. Available: https://wiki.hyperledger.org/display/caliper and https://github.com/hyperledger/caliper

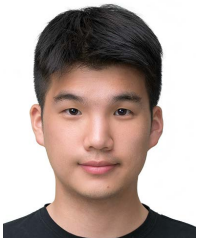
[40] D. C. Nguyen, P. N. Pathirana, M. Ding, and A. Seneviratne, "Integration of blockchain and cloud of things: Architecture, applications and challenges," *IEEE Commun. Surveys Tuts.*, vol. 22, no. 4, pp. 2521–2549, 4th Quart., 2020.

[41] T. Guggenberger, J. Sedlmeir, G. Fridgen, and A. Luckow, "An in-depth investigation of performance characteristics of hyperledger fabric," 2021, *arXiv:2102.07731*.
[42] H. Javaid, C. Hu, and G. Brebner, "Optimizing validation phase of hyperledger fabric," in *Proc. IEEE 27th Int. Symp. Modeling, Anal., Simulation Comput. Telecommun. Syst. (MASCOTS)*, Oct. 2019, pp. 269–275.
[43] T. S. L. Nguyen, G. Jourjon, M. Potop-Butucaru, and K. L. Thai, "Impact of network delays on hyperledger fabric," in *Proc. IEEE Conf. Comput. Commun. Workshops (INFOCOM WKSHPS)*, Apr. 2019, pp. 222–227.
[44] J. A. Chacko, R. Mayer, and H.-A. Jacobsen, "Why do my blockchain transactions fail? A study of hyperledger fabric," in *Proc. Int. Conf. Manage. Data*, 2021, pp. 221–234.
[45] J. Song, P. Zhang, M. Alkubati, Y. Bao, and G. Yu, "Research advances on blockchain-as-a-service: Architectures, applications and challenges," *Digit. Commun. Netw.*, Feb. 2021.
[46] Q. Lu, X. Xu, Y. Liu, I. Weber, L. Zhu, and W. Zhang, "uBaaS: A unified blockchain as a service platform," *Future Gener. Comput. Syst.*, vol. 101, pp. 564–575, Dec. 2019.
[47] H. Chen and L.-J. Zhang, "FBaaS: Functional blockchain as a service," in *Blockchain–(ICBC)*, S. Chen, H. Wang, and L.-J. Zhang, Eds. Cham, Switzerland: Springerg, 2018, pp. 243–250.
[48] *The Ordering Service—Hyperledger-Fabricdocs Main Documentation*. Accessed: Jun. 13, 2022. [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/release-2.2/orderer/ordering_service.html
[49] I. M. Coelho, V. N. Coelho, R. P. Araujo, W. Yong Qiang, and B. D. Rhodes, "Challenges of PBFT-inspired consensus for blockchain and enhancements over neo dBFT," *Future Internet*, vol. 12, no. 8, p. 129, Jul. 2020.
[50] S. D. Palma, R. Pareschi, and F. Zappone, "What is your distributed (Hyper)ledger?" in *Proc. IEEE/ACM 4th Int. Workshop Emerg. Trends Softw. Eng. Blockchain (WETSEB)*, May 2021, pp. 27–33.

GYEONGSIK YANG (Member, IEEE) received the B.S., M.S., and Ph.D. degrees in computer science and engineering from Korea University, Seoul, South Korea, in 2015, 2017, and 2019, respectively. He was a Research Intern at Microsoft Research Asia, in 2018. He is currently a Research Professor with Korea University. His research interests include operating systems, network virtualization, AI systems, datacenter systems, and SDN.

KWANHOON LEE received the B.S. degree in civil engineering from the Seoul National University of Science and Technology, Seoul, South Korea, in 2018, and the M.S. degree in computer science from Korea University, Seoul, in 2021. He was a Research Intern at the Korea Institute of Science and Technology (KIST). His research interests include blockchain and cloud computing.

KYUNGWOON LEE received the B.E. degree from the School of Electronics Engineering, Kyungpook National University, Daegu, South Korea, and the M.S. and Ph.D. degrees in computer science from Korea University, Seoul, South Korea. Currently, she is an Assistant Professor with the School of Electronics Engineering, Kyungpook National University. Her research interests include resource scheduling in cloud computing, container and server virtualization, and kernel networking stack.



YEONHO YOO (Graduate Student Member, IEEE) received the B.S. degree in computer science from Kookmin University, Seoul, South Korea, in 2017, and the M.S. degree in computer science and engineering from Korea University, Seoul, in 2021. He is currently pursuing the Ph.D. degree with Korea University. His current research interests include network virtualization, SDN, datacenter systems, and AI systems.

HYOWON LEE received the B.S. degree in computer science and engineering from Hanyang University, Ansan, South Korea, in 2020. She is currently pursuing the M.S. degree with Korea University, Seoul, South Korea. Her research interests include container virtualization and resource scheduling in cloud computing.

CHUCK YOO (Member, IEEE) received the B.S. and M.S. degrees in electronic engineering from Seoul National University and the M.S. and Ph.D. degrees in computer science from the University of Michigan, Ann Arbor. He was a Researcher at Sun Microsystems. Since 1995, he has been with the College of Informatics, Korea University, where he is currently a Professor. His research interests include operating systems, cloud computing, and AI infrastructure.