Understanding the Scalability of Hyperledger Fabric

Minh Quang Nguyen
National University of Singapore
quang.nguyen@u.nus.edu

Dumitrel Loghin
National University of Singapore
dumitrel@comp.nus.edu.sg

Tien Tuan Anh Dinh
Singapore University of Technology and Design
dinhtta@sutd.edu.sg

ABSTRACT

The rapid growth of blockchain systems leads to increasing interest in understanding and comparing blockchain performance at scale. In this paper, we focus on analyzing the performance of Hyperledger Fabric v1.1 — one of the most popular permissioned blockchain systems. Prior works have analyzed Hyperledger Fabric v0.6 in depth, but newer versions of the system undergo significant changes that warrant new analysis. Existing works on benchmarking the system are limited in their scope: some consider only small networks, others consider scalability of only parts of the system instead of the whole. We perform a comprehensive performance analysis of Hyperledger Fabric v1.1 at scale. We extend an existing benchmarking tool to conduct experiments over many servers while scaling all important components of the system. Our results demonstrate that Fabric v1.1's scalability bottlenecks lie in the communication overhead between the execution and ordering phase. Furthermore, we show that scaling the Kafka cluster that is used for the ordering phase does not affect the overall throughput.

1. INTRODUCTION

Blockchain technology is moving beyond cryptocurrency applications [23, 19], and becoming a new platform for general transaction processing. A blockchain is a distributed ledger which is maintained by a network of nodes that do not trust each other. At each node, the ledger is stored as a chain of blocks, where each block is cryptographically linked to the previous block. Compared to a traditional distributed database system, a blockchain can tolerate stronger failures, namely Byzantine failures in which malicious nodes can behave arbitrarily.

There are two types of blockchains: permissionless and permissioned. The performance of both types of systems, however, lags far behind that of a typical database. The primary bottleneck is the consensus protocol used to ensure consistency among nodes. Proof-of-Work [19], for example, is highly expensive and achieves very low throughputs, whereas PBFT [13] does not scale to a large number of nodes [15]. Beside consensus, another source of inefficiency is the order-execute transaction model, in which transactions are first ordered into blocks, then they are executed sequentially by every node. This model, adopted by popular blockchains such as Ethereum and Hyperledger Fabric v0.6, is not efficient because there is no concurrency in transaction execution [20].

New versions of Hyperledger Fabric [6], namely version v1.1 and later, implement a new transaction model called execute-order-validate model. Inspired by optimistic concurrency control mechanisms in database systems, this model consists of three phases. In the first phase, transactions are executed (or simulated) speculatively. This simulation does not affect the global state of the ledger. In the second phase, they are ordered and grouped into blocks. In the third phase, called validation or commit, they are checked for conflicts between the order and the execution results. Finally, non-conflicting transactions are committed to the ledger.

The ordering phase is performed by an ordering service which is loosely-coupled with the blockchain. Hyperledger Fabric offers two types of ordering service: *Solo* which is used for development and testing, and *Kafka* service which is used for deployment in production system. The Kafka service forwards transactions to an Apache Kafka cluster for ordering [16]. By allowing parallel transaction execution, Hyperledger Fabric v1.1 can potentially achieve higher transaction throughputs than systems that execute transactions sequentially. However, it introduces an extra communication phase compared to the order-execute model, thus incurring more overhead.

In this paper, we aim to provide a comprehensive performance analysis of the execute-order-validate transaction model. To this end, we evaluate the throughput and latency of Hyperledger Fabric v1.1 in a local cluster of up to 48 nodes, running with Kafka ordering service. Our work differs from Blockbench [15], which benchmarks an earlier version of Hyperledger Fabric with order-execute transaction model. Its scope is more extensive than other recent works that examine the performance of Hyperledger Fabric v1.1 and later. For example, [9] does not consider the effect of scaling the Kafka cluster on the performance. Similarly, [10, 20, 22] fix the size of the Kafka cluster, and use fewer than 10 nodes. In contrast, we examine the impact of scaling Kafka cluster on the overall performance, using up to 48 nodes in a local cluster.

Hyperledger Caliper [5] is the official benchmarking tool

for Hyperledger Fabric. However, it offers little support and documentation on how to benchmark a real distributed Fabric setup with Kafka ordering service. Most of the documentation and scripts are considering the Solo orderer and a single client. To overcome this, we developed Caliper++ by enhacing Caliper with a set of scripts to configure, start and benchmark a distributed Fabric network with variable number and type of nodes.

In summary, our main contributions are as follows:

- We extend the Caliper, Hyperledger's benchmarking tool, by adding support for distributed benchmarking. The result is Caliper++, a benchmarking tool that can start Fabric with varying sizes and configurations.
- We perform a comprehensive experimental evaluation of Fabric v1.1 using Smallbank smart contract. We scale the number of peers involved in all three transaction phases of the execute-order-validate model, by using up to 48 nodes in a local cluster.
- We show that endorsing peer — the one that perform the execution and validation phase — is the primary scalability bottleneck. In particular, increasing the number of endorsing peers not only incurs overheads in the execution and ordering phase, but also leads to degraded performance of the Kafka ordering service. On the other hand, we observe that scaling the Kafka cluster does not impact the overall throughput.

Section 2 discusses the background on blockchain systems, and the architecture of Hyperledger Fabric v1.1. Section 3 describes related works on blockchain benchmarking. Section 4 describes our benchmarking tool called Caliper++. Section 5 presents the experiment setup and results, before concluding in Section 7.

2. BACKGROUND

Blockchain networks can be classified as either permisionless (or public) or permissioned (or private). In the former, such as Ethereum [14] and Bitcoin [19], any node can join the network, can issue and execute transactions. In the latter, such as Hyperledger Fabric, Tendermint [12], Chain [4], or Quorum [8], nodes must be authenticated and authorized to send or execute transactions.

The ledger stores all historical and current states of the blockchain in the form of blocks linked together cryptographically. To append a block, all nodes must agree. More specifically, the nodes reach agreement by running a distributed consensus protocol. This protocol establishes a global order of the transactions. The majority of permissionless blockchains use computation-based consensus protocols, such as Proof-of-Work (PoW), to select a node that decides which block is appended to the ledger. Permissioned blockchains, on the other hand, use communication-based protocols, such as PBFT [13], in which nodes have equal votes and go through multiple rounds of communication to reach consensus. Permissioned blockchains have been shown to outperform permissionless ones, although its performance is much lower to that of a database system [15]. Nevertheless, they are useful for multi-organization applications where authentication is required but participating organization do not trust each other.

2.1 Hyperledger Fabric v1.1

Hyperledger Fabric v1.1 (or Fabric) is a permissioned (or private) blockchain designed specifically for enterprise applications. A blockchain smart contract, also called chaincode in Fabric, can be implemented in any programming language. A Fabric network comprises four types of nodes: endorsing nodes (or peers), non-endorsing nodes (or peers), clients, and ordering service nodes. These nodes may belong to different organizations. Each node is given an identity by a Membership Service Provider (MSP), which is run by one of the organizations.

Endorsing and Non-Endorsing Peers. A peer in the system stores a copy of the ledger in either GolevelDB [7] or CouchDB [1] database. It can be an endorsing peer if specified so in the *Endorsement Policies*; otherwise it is non-endorsing. Endorsing peers maintain the chaincode, execute transactions, and create (or endorse) transactions to be forwarded to the ordering nodes.

Endorsement Policies. A endorsement policy is associated with a chaincode, and it specifies the set of endorsing peers. Only designated administrators can modify endorsement policies.

System Chaincodes. In addition to running chaincode specified by users, Fabric peers run a number of pre-define system chaincodes. Theare are four system chaincodes: the life cycle system chaincode (LSCC) for installing, instantiating and updating chaincodes, the endorsement system chaincode (ESCC) for endorsing transactions by digitally signing them, the validation system chaincode (VSCC) for validating endorsement signatures, and the configuration system chaincode (CSCC) for managing channel configurations.

Channel. Fabric supports multiple blockchains that use the same ordering service. Each blockchain is identified by a channel, where members may consist of different sets of peers. Transactions on a channel can only be viewed by its members. The order of transactions in one channel is isolated from those of in another channel and there is no coordination between channels [9].

Ordering Service. The Ordering Service consists of multiple Ordering Service Nodes (OSNs). The OSNs establish a global order of transactions and construct blocks to be broadcast to peers. A block is created when one of the following conditions is satisfied: (1) the number of transactions reaches a specified threshold, (2) a specified timeout is reached; (3) the block size reaches a specified threshold. Endorsing peers receive blocks directly from the ordering service, while non-endorsing peers get blocks via a gossip protocol from other endorsing peers and from the ordering service. A Solo ordering service consists of a single node (or orderer) which serves all clients. A Kafka ordering service consists of a Kafka cluster to which OSNs forward transactions for ordering.

Client. A client sends transactions to endorsing peers, wait until it receives all endorsement from these peers, then broadcasts the endorsed transactions to OSNs.

Transaction. Separating the ordering service from the peers, Hyperledger Fabric v1.1 adopts the execute-order-validate (also called simulate-order-commit) model for executing a transaction. In contrast, Hyperledger Fabric v0.6 and other blockchain systems use the order-execute transaction model. In the next section, we describe the life cycle of a transaction in Fabric v1.1.

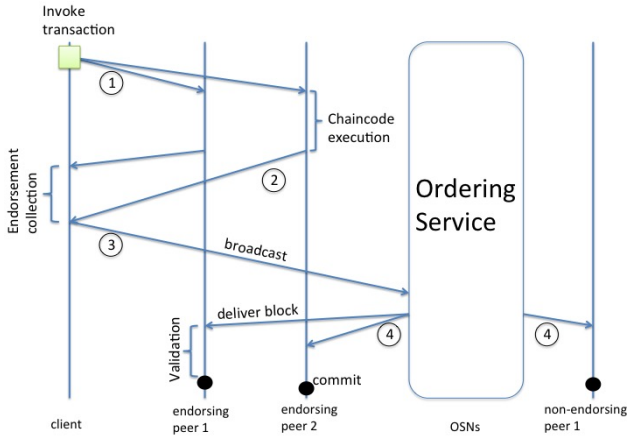


Figure 1: Transaction life cycle in Fabric v1.1.

2.2 Transaction Life Cycle in Fabric

Fabric v1.1 employs a novel execute-order-validate transaction model which comprises three phases, as depicted in Figure 1.

Execution phase. A client submits a signed transaction to the endorsing peers. Each endorsing peer verifies if the client is authorized to invoke the transaction, then speculatively executes the transaction against its local blockchain state. This process is done in parallel without coordination among endorsing peers. Output of the execution, which consists of a read set and a write set, is used to create an endorsement message. The peer signs the endorsement and sends it back to the client.

Ordering phase. After collecting enough endorsements according to the endorsement policy, the client creates an endorsed transaction and forwards it to the OSNs. The ordering service orders the transactions globally, groups them into blocks and sends them directly to the endorsing peers or gossips them to non-endorsing peers.

Validation Phase. When receiving a block, every peer validates the transactions in the block against the endorsement policy of the chaincode. After that, for every transaction, the peer checks for read-write conflict by comparing the key versions in the read set with those from the current ledger states. Any transaction that fails the validation or the conflict check is marked as invalid. Invalid transactions are discarded and their effects to the blockchain states are rolled back. Finally, the block is commited by the peer and appended to the peer's local ledger.

3. RELATED WORK

There has been considerable interest in benchmarking Hyperledger Fabric. Blockbench [15] is the first framework for benchmarking permissioned blockchains. It divides the blockchain stack into four layers: consensus, data, execution and application. It contains many micro and macro benchmarks for evaluating performance of every layer. However, Blockbench only supports Fabric v0.6 whose architecture is highly different to that of v1.1. Although the results reported in [15] cannot be extended to the new system, they present useful baseline performance of the order-execute transaction model.

E. Androulaki et al.[9] evaluate Fabric v1.1 using Kafka ordering service. They use a Kafka-based network of 5 endorsing peers, 4 Kafka nodes, 3 orderers, 3 Zookeeper nodes, and a varying number of non-endorsing peers. We note that non-endorsing peers do not play an active role in the transaction life cycle, as shown in Figure 1. Therefore, they are not a potential scalability bottleneck. We argue that the number of non-endorsing peers is not an important system parameter. As a consequence, [9] falls short in the analysis of system scalability.

P. Thakkar et al.[22] also benchmark Fabric v1.1. However, their network is small, consisting of only 8 peers, 1 orderer and a Kafka cluster. The Kafka cluster runs on a single node, which does not fully capture the communication overhead in a real system. Furthermore, they do not consider scalability in terms of the number nodes. Ankur Sharma et al.[20] use the Solo orderer which is not meant to run in a real system. The network in [20] is small, with 4 peers, 1 client and 1 orderer distributed on 6 cluster nodes. Similarly, A. Baliga et al. [10] use the Solo orderer. Although they examine the effect of scaling the number of endorsing peers, they use a simple smart contract. In contrast, we use complex smart contracts representing realistic transactional workloads, such as Smallbank or YCSB.

M. Brandenburger et al.[11] and C. Gorenflo et al.[17] separately explore pitfalls arising from Fabric v1.0 (and above), and propose techniques to improve system's performance. Similar to our findings, they identify endorsement in the execution phase as the system bottleneck. However, their experimental setup consists of very small Fabric networks, with 5 nodes at most.

In summary, related works fail to capture Fabric's performance on a realistic setup. To this end, our work provides a comprehensive benchmarking on a cluster of 48 physical nodes, when Kafka is used as ordering service.

4. CALIPER++

In this section, we present Caliper++[1], our extension to Hyperledger Caliper [5], the official benchmarking tool for Hyperledger Fabric.

4.1 Starting Up Fabric Network

The main challenge of benchmarking Kafka-based Fabric network at scale lies in starting up the distributed Fabric network. While Fabric v0.6 has only one type of nodes, represented by the peers themselves, Fabric v1.1 has six types of nodes: endorsing peer, non-endorsing peer, orderer, Kafka [2] node (or broker), Zookeeper [3] node, and client. This means there are more system parameters to consider. It also makes it more complex to automatically start up the network with all these components. In fact, we found limited official documentation on configuring a Kafka-based system. There are even fewer documentations on running Fabric on multiple physical nodes.

At its current state, Caliper [5] – the official benchmarking tool for Hyperledger blockchains – only supports testing on a local environment with a limited set of predefined Fabric network topologies, all of which use the Solo ordering service. Caliper++ supports additional functionalities for

[1]The source code and documentation are available on GitHub at `https://github.com/quangtdn/caliper-plus`

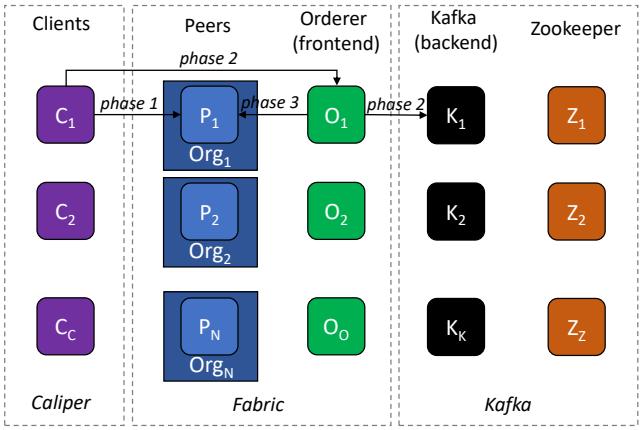


Figure 2: Hyperledger Fabric Setup

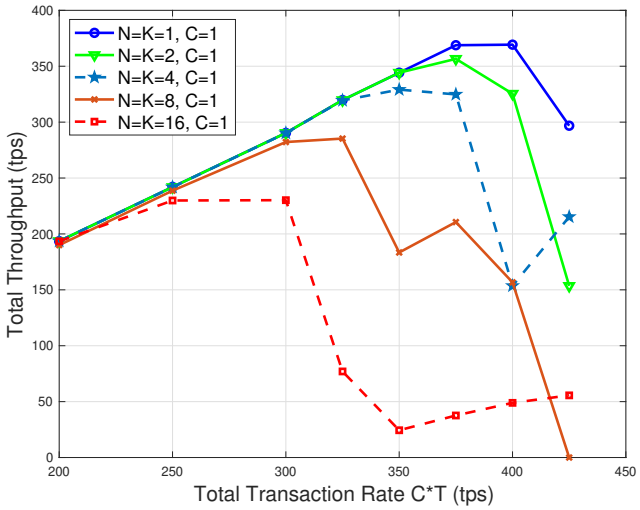


Figure 3: Throughput with a single client $C = 1$

benchmarking Kafka-based Fabric network at scale. In particular, it provides a set of scripts to (a) auto-generate configuration files for any Fabric network topology, (b) bring up large-scale Kafka-based Fabric network across cluster nodes, (c) launch additional benchmarking tools at the operating system level, such as *dstat, strace, perf*. In addition, it allows benchmarking with distributed clients.

4.2 Benchmark Driver

Caliper implements the role of the client in Figure 1. It can be configured to issue a given number of transactions with a fixed rate. After sending one transaction, it schedules sending the next transaction after such interval that ensures the specified transaction rate is met. Responses from endorsing peers received during the execution phase are processed asynchronously in the client's main event loop.

We note that Caliper implements a benchmark client (or driver) that is tightly coupled to the Fabric transaction workflow. This is different to Blockbench driver, which is separated from the blockchain network. In particular, Blockbench driver sends transactions via JSON APIs, waits for the transaction IDs from one of the blockchain nodes, then sleeps an appropriate amount of time before sending the next transaction. This driver is simply a workload generator, which is independent of the blockchain transaction processing workflow. Caliper, in contrast, implements specific logic in which it waits and validates the responses from endorsing peers against endorsement policies. Another difference of Caliper is that it processes responses from multiple nodes for each transaction, as opposed to Blockbench driver which processes transactions from a single node.

The original Caliper can run multiple drivers on the same node. However, under high load, the node is likely to be the bottleneck. To overcome this, Caliper++ contains scripts that distribute the drivers over any number of physical cluster nodes. It coordinates the drivers so that they achieve the target aggregate transaction rate. However, our experiments later show that such distribution does not improve the overall throughput.

5. PERFORMANCE ANALYSIS

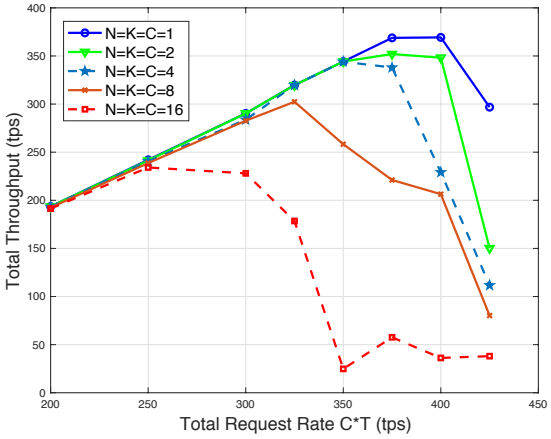
In this section, we present our analysis of Hyperledger Fabric v1.1 using Caliper++. We focus on the following metrics: throughput, latency and scalability. The throughput is measured as the number of successful transactions per second (tps). Latency is defined as the elapsed time (in second) from when the transaction is submitted to the time it is committed. In the following, we report the average latency of all successful transactions. Scalability is defined as the changes in throughput and latency as the network size increases.

5.1 Experiment Setup

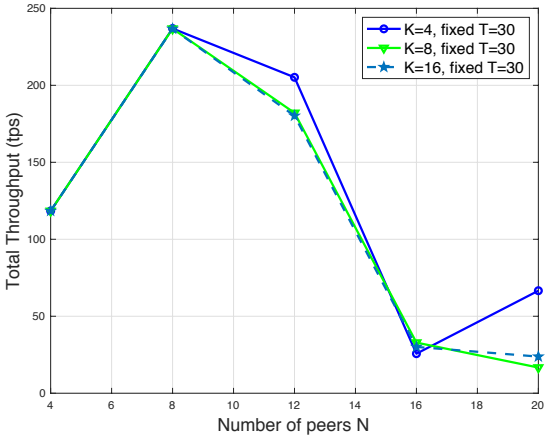
A Hyperledger Fabric network topology is defined by the (N, C, T, O, K, Z), in which N is the number of endorsing peers, C is the number of Caliper++ benchmark clients, T is the transaction rate per client, O is the number orderers, K is the number of Kafka nodes (or brokers), and Z is the number of Zookeeper nodes Z. Figure 2 illustrates the topology. Note that we do not consider non-endorsing peers, as they do not play an active role in the transaction life cycle. The endorsing peers belong to different organizations (orgs), and they use GolevelDB as the state database. The endorsement policy includes all N endorsing peers. The block size is set to 100 transactions per block, and the block timeout is set to 2s.

In a Kafka-based ordering service, the orderers act as proxies forwarding transactions to the Kafka brokers. For each Kafka broker, we set *min.insync.replicas* $= 2$, and *default.replication.factor* $= K - 1$. This configuration means that a transaction is written to $K - 1$ Kafka brokers, and committed after being successfully written to two brokers. When *default.replication.factor* $< K - 1$, there are idle Kafka nodes that do not contribute to the ordering phase. In practice, the idle nodes are reserved for ordering other channels in the network. However, our experiments use a single channel, thus we configure the replication factor so that there are no idle nodes. We note that a high replication factor also increases the fault-tolerance of the Kafka cluster, thereby providing insights into the trade-offs between performance and fault-tolerance. Unless specified otherwise, we set $O = 4, Z = 3$ and run all drivers on a single node.

We use the popular OLTP workload called Smallbank [18]. It is a transactional workload that simulates a simple bank-

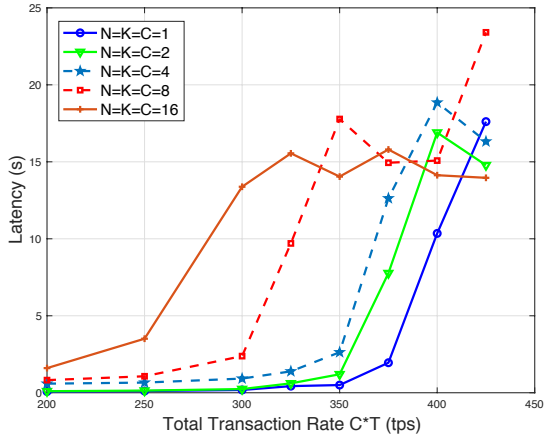


(a) Throughput for the setting $N = C = K$

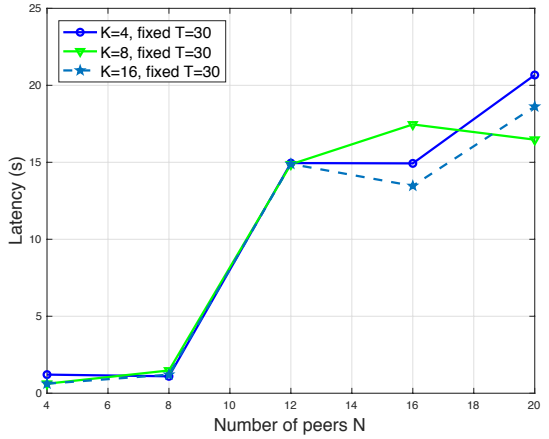


(b) Throughput for scaling clients with fixed rate $T = 30$

Figure 4: Throughput while scaling network size



(a) Latency for the setting $N = C = K$



(b) Latency for scaling clients with fixed rate $T = 30$

Figure 5: Latency while scaling network size

ing application. It is also used by Blockbench to benchmark Fabric v0.6 [15] which gives us a baseline for evaluating the new architecture of Fabric v1.1.

The experiments were run on a 48-node commodity cluster. Each node has an Intel Xeon E5-1650 CPU clocked at 3.5 GHz, 32 GB of RAM, 2 TB hard drive and a Gigabit Ethernet interface. The nodes run Ubuntu 16.04 Xenial Xerus.

5.2 Fabric's Capacity

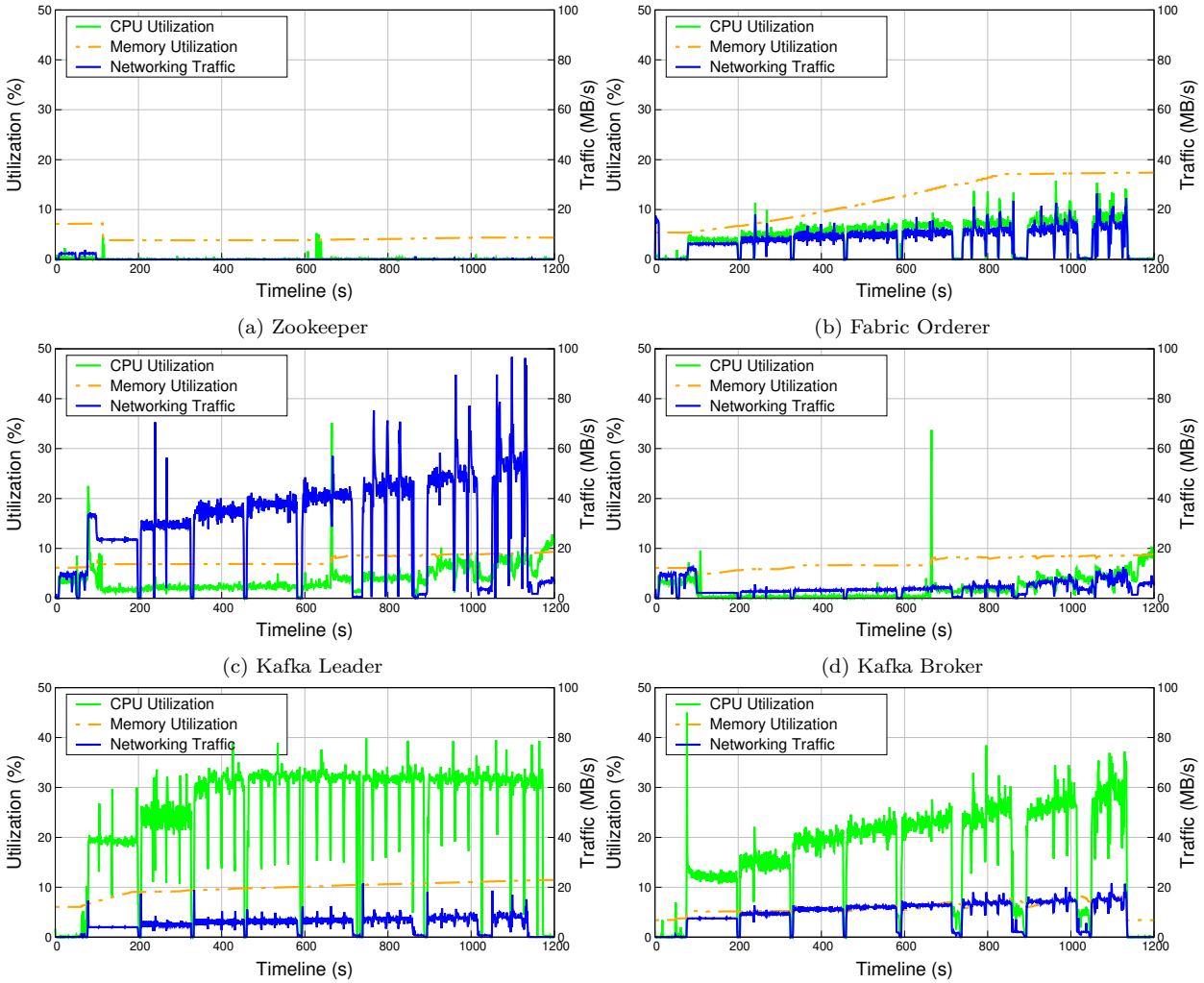
In this section, we analyze the system capacity. We vary the number of endorsing peers N, while setting $C = N$. We increase the transaction rate until the system is saturated. The saturation rate $C * T$ determined in these experiments is used in later experiments when we increase N and K.

First, we examine the system performance when the number of Kafka brokers equals the number of peers, $K = N$. The results are shown in Figure 4a and Figure 5a. We observe that for all values of N, the throughput starts to degrade when the transaction rate exceeds 400 transactions per second. The peak throughput for $N = 1$ and $N = 16$ is around 375tps and 300tps, respectively. The latency exhibits a similar behavior, except for $N = 16$ it starts to degrade from a request rate of 250 tps. From these results, we select the request rates of $C * T = 300$ and $C * T = 400$ as the points before and after saturation, respectively.

We note that this peak throughput is significantly lower than that of Fabric v0.6, and it is only comparable to that of Ethereum [15]. We attribute this difference to the new architecture of Fabric v1.1 that requires more communications among nodes of different roles. On the other hand, Fabric v1.1 is able to scale beyond 16 peers with a throughput of around 50tps and a latency of 20s, while Fabric v0.6 stops working with more than 16 peers [15].

Second, we examine the system performance with fixed number of Kafka brokers K, while fixing the transaction rate to $T = 30$ tps. Throughput and latency results are shown in Figure 4b and Figure 5b, respectively. We observe that the throughput decreases and the latency increases when $N > 8$. In particular, when the request rate is higher than 360tps, the overall throughput decreases drastically. This strengthens our observation that Fabric is saturated with a request rate between 300tps and 400tps.

Third, we examine whether the Caliper++ driver is the bottleneck. We run experiments with $C = 1$ while increasing N and setting $K = N$. The throughput, depicted in Fig-



(a) Zookeeper

(b) Fabric Orderer

(c) Kafka Leader

(d) Kafka Broker

(e) Peer

(f) Client

Figure 6: Resource utilization of different nodes in a Hyperledger Fabric network

ure 3, exhibits similar performance as the throughput with multiple drivers. In particular, we observe similar saturation points as shown in Figure 8, which indicates that the driver is not the bottleneck. We note that before saturation, the throughput when $C > 1$ is always slightly better than that when $C = 1$. This is because the single driver has to handle more messages and therefore incurs more communication overhead.

To further validate our observations, we profile the system resources with *dstat*. More specifically, we use *dstat* to measure CPU, memory and network utilization on each node. Figure 6 shows the results on different nodes in a network with $N = 8$, $K = 8$, $O = 4$, $Z = 3$ and $C = 1$ client. We vary $C * T$ in the range $[200, 425]$ and run the experiments for 120s. The trends are consistent with the throughput results. We observe that client and peer resource utilization, especially in terms of CPU and network, increases with the transaction rate, as expected.

We note that no Fabric nodes fully utilize their resources. The CPU utilization is below 40% on all nodes. While the clients and peers use up to 30% of the CPU, some Kafka brokers and Zookeeper nodes use less than 5%. Memory utilization is also low: only the front-end orderer uses up to 20% (or around 6 GB) to buffer requests. For networking, the client, peers, orderers and Kafka leaders have the highest traffic volume. In particular, Kafka leaders use 60 MB/s on average, which is below 50% the available bandwidth. The high network utilization at the orderer is explained by its double role, as a receiver of requests from the client and as a dispatcher of requests to Kafka nodes, as shown in Figure 2. Similarly, some Kafka nodes, or leaders, have high network utilization because they serve two roles: as leaders for managing the replication inside the Kafka cluster, and as contact points for the orderers.

5.3 Overhead of Fabric Orderers

In this section, we analyze the impact of scaling the number of Fabric orderers. We consider the network configurations with the lowest performance in the previous section, $N = K = C = 8$ and $N = K = C = 16$, and increase O from

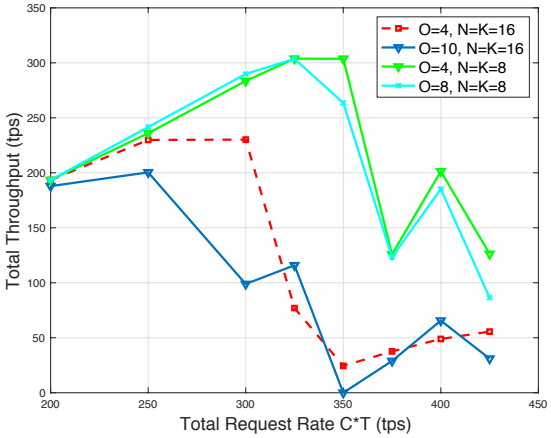


Figure 7: Impact of increasing the number of orderers

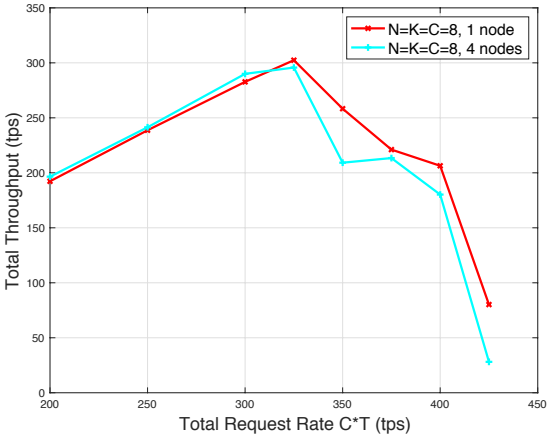


Figure 8: Impact of benchmarking with distributed clients

4 to 8 and 10[2], respectively. The throughput, plotted in Figure 7, shows that the network with more orderers performs worse, in general. Since the orderers only forward transactions to the Kafka cluster, scaling them only increases the communication overhead, which leads to lower throughput.

To assess communication overhead, we profile the execution on $N = K = C = 8$ with *dstat* and compute the total networking traffic at orderer level. With $O = 4$ orderers, the total communication volume is 42.6 GB over the entire benchmarking period. With $O = 8$ orderers, the communication volume increases by 26% to a total of 53.7 GB. To avoid unnecessary overhead, we keep the number of orderers to four during our benchmarking, unless specified otherwise.

5.4 Distributed Client Driver

In this section, we analyze the effect of distributing the client drivers onto multiple physical nodes. For these experiments, we consider a network with $N = K = C = 8$ and run the drivers on 4 cluster nodes, each node with 2 drivers. We compare the performance with running all 8 drivers on a single node. The results, shown in Figure 8, suggest that the performance difference is negligible. In other words, distributing the drivers over multiple nodes does not improve the overall throughput.

In its current version, Caliper's client is not restricted by hardware resource availability at single node level, hence, distributing it to multiple nodes is not necessary. For example, 8 client drivers running on a single node utilize 2.4 CPU cores, on average, when there are 12 available cores. The execution of 2 drivers on a single node utilizes 0.75 CPU cores, on average. At the networking subsystem, running all client drivers on a single node produces an average traffic of 9 MB/s (5.8 MB/s standard deviation), while distributing the clients results in an average traffic of 3.4 MB/s (1.4 MB/s standard deviation) at node level.

5.5 Scaling the Number of Peers

In this section, we analyze the effect of scaling the number of peers. We fix the total request rate to $C * T = 300$ and $C * T = 400$, to represent the load before and after saturation, respectively. K is set to 4, 8 or 16. The number of peers is increased up to $N = 24$. Figure 9 and 10 show the results for throughput and latency, respectively.

We observe that increasing N drastically degrades the system performance. To understand the bottlenecks, we examine the system logs and find that increasing N incurs overhead in both the execute and ordering phase. First, larger N means that each driver has to wait for endorsements from more peers before creating the endorsed transaction. In fact, we find that Caliper++ has more timeout errors when waiting for endorsements and subsequently discarding more transactions. The outcome is that the orderers receives transactions at a lower rate.

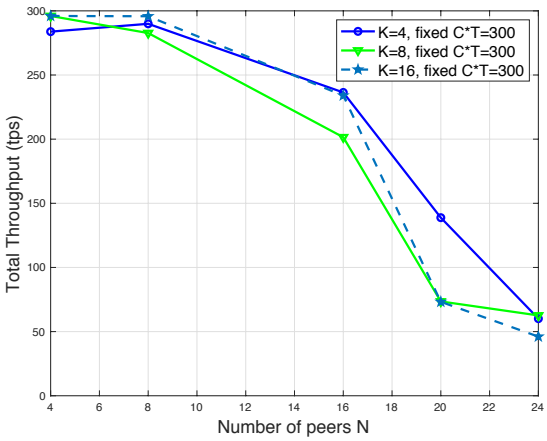
Second, larger N means higher communication overhead in the validation phase because the orderers broadcast blocks to all endorsing peers for validation. In the experiment with a fixed request rate of 400 and $N = 24$, we find that the rate at which orderers receive transactions is higher than the rate of returning confirmed transactions in the form of blocks. In particular, when broadcasting a transaction to the Kafka cluster, the orderer logs a message *"[channel: mychannel] Enqueueing envelope..."*. After the transaction is ordered by Kafka and returned to the orderer for batching blocks, it logs a message *"[channel: mychannel] Envelope enqueued successfully"*. We compute the ratio of the first over the second type of log messages. This ratio is larger than 1.7, suggesting that the Kafka-based ordering service can only complete $1/1.7 \approx 58.8\%$ of the transactions it receives.

Compared to Fabric v0.6, the throughput of Fabric v1.1 is more than 3 times lower on 8 peers, and more than 22 times lower on 24 peers (50tps versus 1100tps) [15]. The throughput of v1.1 degrades notably on more than 16, while for v0.6 it degrades at a slower pace. Interestingly, the latency of v1.1 is lower compared to v0.6 on 24 peers, 16s versus 40s. We attribute the high latency of v0.6 to its PBFT protocol that exhibits higher number of view changes on higher peer counts [15].

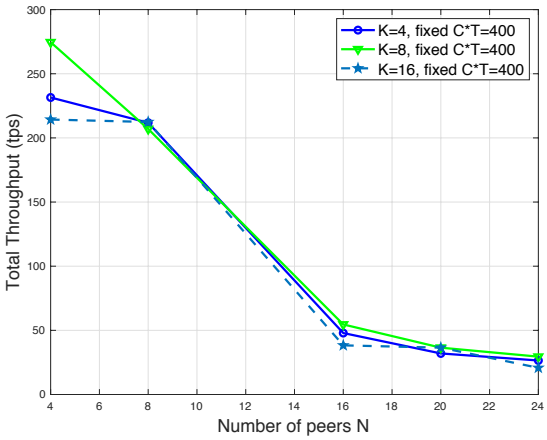
5.6 Scaling Kafka Cluster

In this section, we first analyze the effect of communication within the Kafka cluster to the overall performance. We fix $N = K = C = 16$ and consider two settings. The first,

[2]We use only 10 orderer nodes instead of 16 due to limited resources in our 48-node cluster.

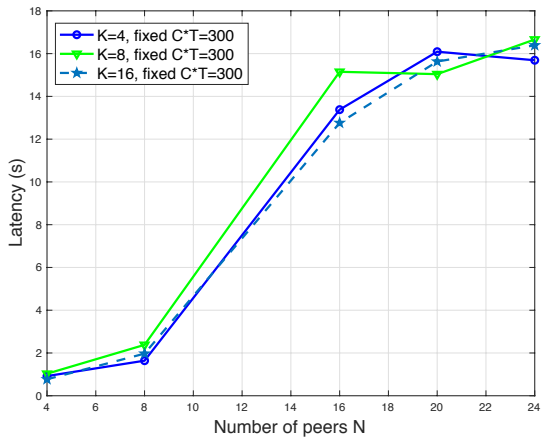


(a) Throughput for fixed request rate $C * T = 300$ tps

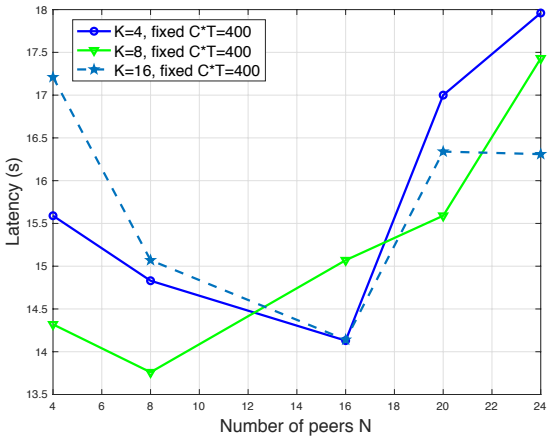


(b) Throughput for fixed request rate $C * T = 400$ tps

Figure 9: Throughput while scaling the number of peers



(a) Latency for fixed request rate $C * T = 300$ tps



(b) Latency for fixed request rate $C * T = 400$ tps

Figure 10: Latency while scaling the number of peers

default.replication.factor = 15 and *min.insync.replicas* = 14, incurs high communication cost within the Kafka cluster. In the second, we set *default.replication.factor* = 1 and *min.insync.replicas* = 1, which achieves no fault tolerance, but has the lowest communication cost because a transaction is committed immediately after being written to one broker. Even if the Kafka cluster has to write the transaction on the remaining Kafka brokers, Fabric's transaction flow does not halt for this operation. The performance difference between these two settings captures the cost of fault-tolerance.

The results, depicted in the Figure 11, show that the throughput of the first setting is always lower than that of the second. However, we note that the difference is small, and therefore conclude that Kafka communication does not affect the overall throughput.

Next, we analyze the effect of increasing the number of Kafka brokers, K. We fix N, $C = N$, and set the total request rate to $C * T = 300$ and $C * T = 400$. For each $N \in \{4, 8, 16\}$, we vary K. The average throughput and latency of each setting is shown in Figure 12 and Figure 13, respectively. We see that scaling the Kafka cluster does not change the throughput pattern or the scalability of the

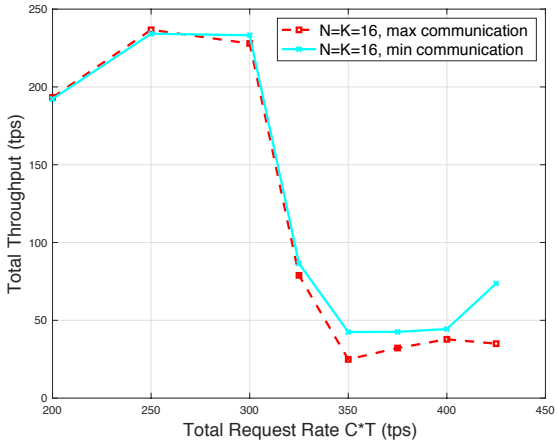
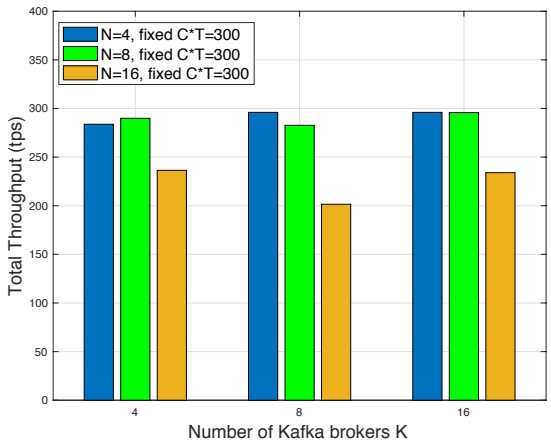
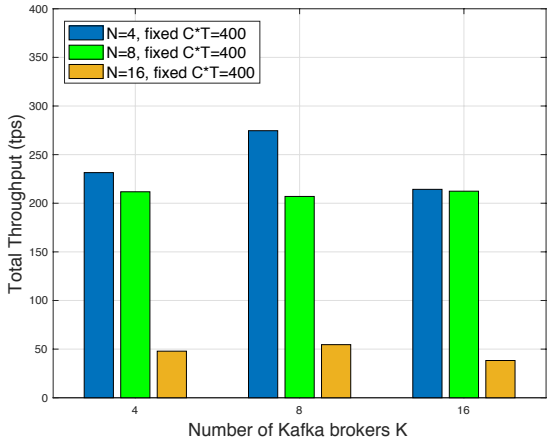


Figure 11: The effect of Kafka communication.

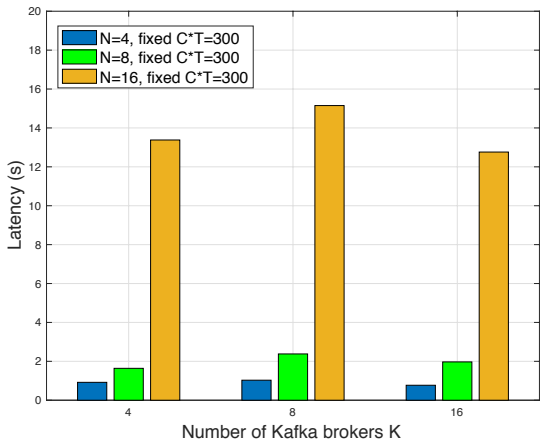


(a) Throughput for request rate $C * T = 300$ tps

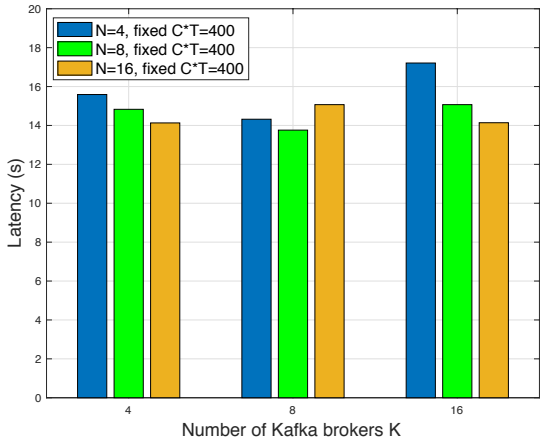


(b) Throughput for request rate $C * T = 400$ tps

Figure 12: Throughput Performance



(a) Latency for request rate $C * T = 300$ tps



(b) Latency for request rate $C * T = 400$ tps

Figure 13: Latency Performance

system. This result is consistent with that in Figure 11, both indicating that the cost of Kafka communication does not affect the overall performance.

6. DISCUSSION

As shown in section 5, the primary scalability bottleneck of Fabric v1.1 is represented by the endorsing peers, which incur overheads to both the execution and ordering phase. In this section, we discuss about these bottlenecks and potential techniques that can be applied to improve the blockchain performance.

M. Brandenburger et al.[11] study the throughput of the endorsement (or execution) phase by increasing the number of clients up to 128 and sending all transactions to a single endorsing peer. They are using a chaincode that returns immediately, such that it causes no overhead on the endorsing peer to speculatively execute it. Their experiments show that the blockchain reaches saturation with 16 to 32 clients. This suggests that the endorsement itself constitutes a bottleneck when a single endorsing peer has to serve many clients.

C. Gorenflo et al.[17] argue that endorsing peers are overloaded at high transaction rate because they are responsible for both committing blocks in the validation phase and endorsing clients' transactions in the execution phase. Hence, they propose separating the commit from endorsement, by introducing specialized committing peers. In particular, after receiving and validating blocks broadcasted by orderers, committing peers send validated blocks to a cluster of endorsing peers who only apply the changes to their replica of the ledger, without further validation.

Although the proposed solution by C. Gorenflo et al.[17] causes one extra communication round, thereby potentially increasing the latency, it represents a reasonable approach towards alleviating the pressure on the endorsing peers. A separate cluster of endorsing peers can accommodate a large number of clients, thus solving the problem identified by M. Brandenburger et al.[11].

However, C. Gorenflo et al.[17] only test this design and show that it works well in a Fabric network with a single endorsing peer, and a small Kafka ordering service. As scaling the Fabric network topology incurs more communication overhead, further benchmarking at scale is needed. On the

other hand, committing and endorsing peers still need to communicate to synchronize new blocks and the state of the ledger. It would be useful to analyze how this overhead scales with the network size. This opens a future research direction for benchmarking and improving the execute-order-validate blockchain systems.

7. CONCLUSION

In this paper, we have presented a comprehensive performance analysis of Hyperledger Fabric v1.1 on a realistic setup of up to 48 cluster nodes using a production-ready Kafka ordering service. To facilitate this analysis, we developed Caliper++ which extends Caliper, Hyperledger's official benchmarking tool. Caliper++ allows for running experiments on a large local cluster while scaling all the important Fabric components.

Our analysis, which is more extensive in scope than existing works, shows that the main scalability bottleneck is the number of endorsing peers, whereas scaling the Kafka cluster does not affect the overall performance. Compared to its previous version, v0.6, Fabric v1.1 exhibits more than three times lower throughput, in the best case, but improves slightly in terms of latency. Our results help inform developers of the performance bottlenecks, and guide users towards achieving the best performance when fine-tuning the deployed network.

This work is a first step towards understanding the scalability of permissionles blockchain systems. While the Kafka ordering service represents a practical approach to achieve scalable ordering in a large network, it may fall short from the security perspective. When the organizations participating in a Fabric network do not trust each other, there is a need for more robust consensus and ordering protocols. Hence, it would be useful to analyze the performance of Fabric with a PBFT-based ordering service [13, 21].

8. REFERENCES

[1] Apache CouchDB. http://couchdb.apache.org/.

[2] Apache Kafka. https://kafka.apache.org/.

[3] Apache ZooKeeper. https://zookeeper.apache.org/.

[4] Chain. http://chain.com.

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

[6] Hyperledger Fabric. https://www.hyperledger.org/projects/fabric.

[7] LevelDB in Go. https://github.com/syndtr/goleveldb/.

[8] Quorum. http://www.jpmorgan.com/global/Quorum.

[9] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. D. Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolic, S. W. Cocco, and J. Yellick. Hyperledger fabric: A distributed operating system for permissioned blockchains. *CoRR*, abs/1801.10228, 2018.

[10] A. Baliga, N. Solanki, S. Verekar, A. Pednekar, P. Kamat, and S. Chatterjee. Performance characterization of hyperledger fabric. pages 65–74, 06 2018.

[11] M. Brandenburger, C. Cachin, R. Kapitza, and A. Sorniotti. Blockchain and trusted computing: Problems, pitfalls, and a solution for hyperledger fabric. *CoRR*, abs/1805.08541, 2018.

[12] E. Buchman. Tendermint: Byzantine fault tolerance in the age of blockchains, Jun 2016. Accessed: 2017-02-06.

[13] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, pages 173–186, Berkeley, CA, USA, 1999. USENIX Association.

[14] C. Dannen. *Introducing Ethereum and Solidity: Foundations of Cryptocurrency and Blockchain Programming for Beginners*. Apress, Berkely, CA, USA, 1st edition, 2017.

[15] 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 *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 1085–1100, New York, NY, USA, 2017. ACM.

[16] N. Garg. *Apache Kafka*. Packt Publishing, 2013.

[17] C. Gorenflo, S. Lee, L. Golab, and S. Keshav. Fastfabric: Scaling hyperledger fabric to 20, 000 transactions per second. *CoRR*, abs/1901.00910, 2019.

[18] M. J. Cahill, U. R. Roehm, and A. David Fekete. Serializable isolation for snapshot databases. volume 34, pages 729–738, 01 2008.

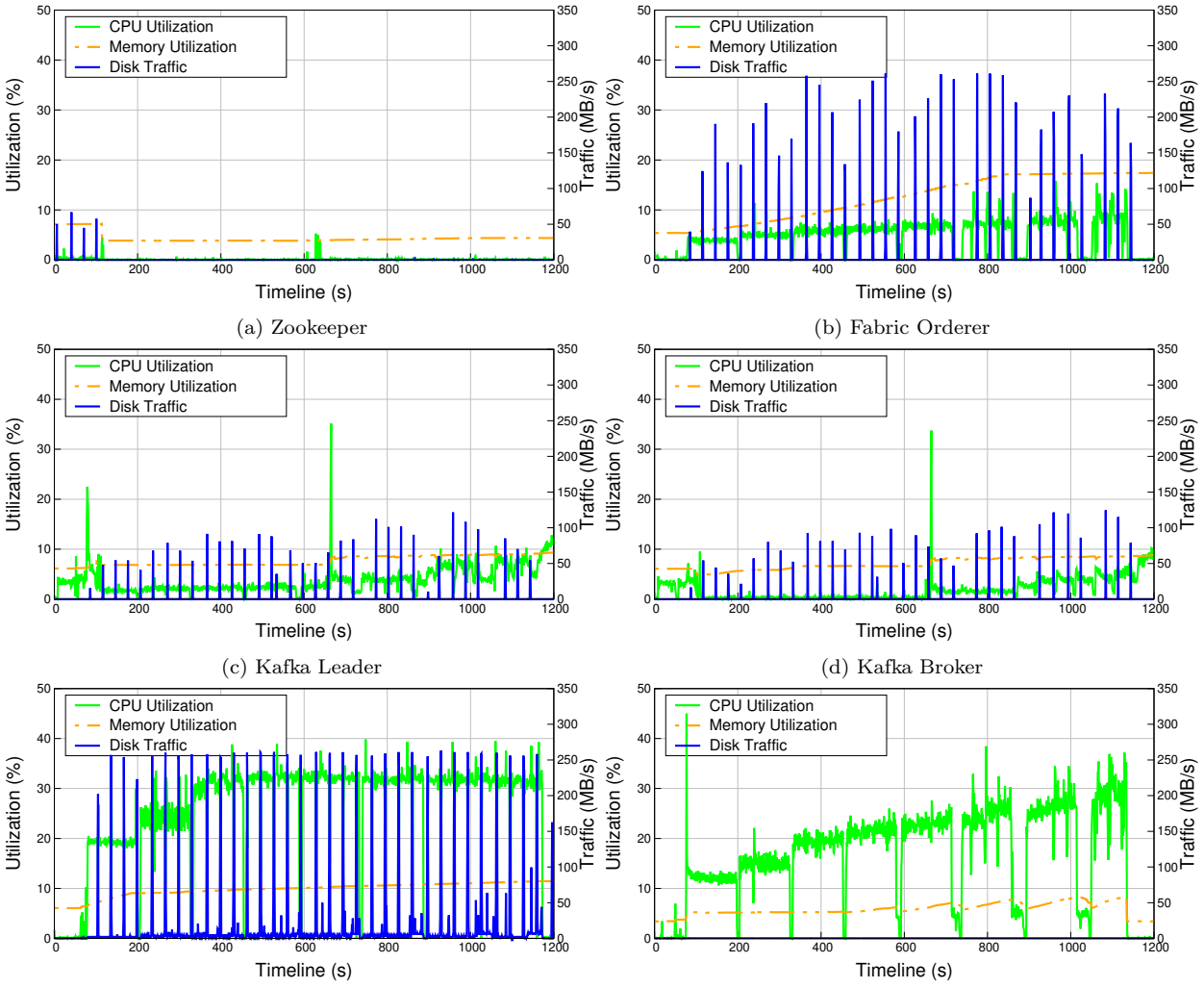
[19] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Cryptography Mailing list at https://metzdowd.com*, 03 2009.

[20] A. Sharma, F. M. Schuhknecht, D. Agrawal, and J. Dittrich. How to databasify a blockchain: the case of hyperledger fabric. *CoRR*, abs/1810.13177, 2018.

[21] J. Sousa, A. Bessani, and M. Vukolic. A byzantine fault-tolerant ordering service for the hyperledger fabric blockchain platform. In *Proceedings of the 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 51–58, June 2018.

[22] P. Thakkar, S. Nathan, and B. Viswanathan. Performance benchmarking and optimizing hyperledger fabric blockchain platform. In *26th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, MASCOTS 2018, Milwaukee, WI, USA, September 25-28, 2018*, pages 264–276, 2018.

[23] S. Underwood. Blockchain beyond bitcoin. *Commun. ACM*, 59(11):15–17, Oct. 2016.



(a) Zookeeper

(b) Fabric Orderer

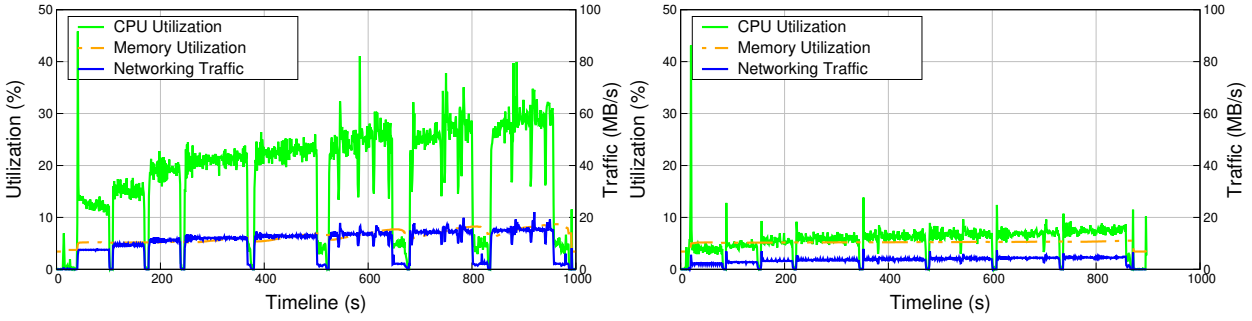
(c) Kafka Leader

(d) Kafka Broker

(e) Peer

(f) Client

Figure 14: Resource utilization (CPU, memory, disk) of different nodes in a Hyperledger Fabric network



(a) Client driver runs on a single node

(b) Client driver runs on four nodes

Figure 15: Resource utilization of one client node when $N = K = C = 8$

APPENDIX

A. ADDITIONAL RESULTS

In this appendix, we include additional performance measurements. Figure 14 shows disk traffic, in addition to CPU and memory utilization, for all types of nodes in a Fabric network and for the same experiment presented in Figure 6. We observe higher disk traffic on Fabric's peer and orderer, compared to other types of nodes. We attribute this high traffic to read/write operations from/to the ledger. Kafka nodes have the second highest disk traffic which is attributed to the replication mechanism.

Complementary to the analysis in Section 5.3, Figure 15 compares the resource utilization of a cluster node running Fabric v1.1 client on the $N = K = C = 8$ topology. In particular, Figure 15a shows the resource utilization of a node running all 8 client drivers, while Figure 15b shows the case when one cluster node runs only 2 drivers. Both plots indicate that client drivers under-utilize the hardware.

Figure 16 presents the number of transaction timeouts, compared to the total number of transactions issued, when the number of endorsing peers is increased from 8 to 24. On the system with 24 peers, all transactions fail when the request rate is higher than 325tps, some of them in the verification, others due to timeout. We observe that for a transaction rate smaller or equal to 325tps, the system with 24 peers exhibits more timeouts compared to the system with 8 peers. This result is presented in Section 5.5.

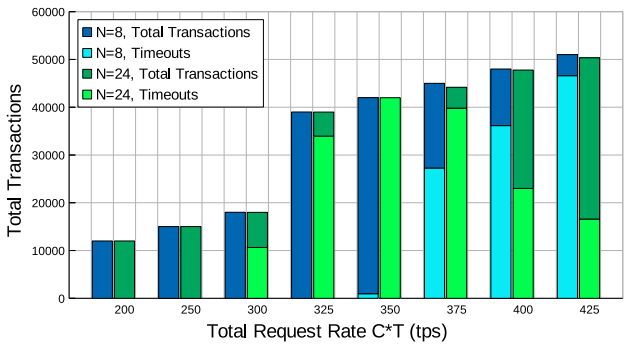


Figure 16: Transaction timeouts