In the First Crypto Valley Conference on Blockchain Technology (CVCBT 2018)

Performance Characterization of Hyperledger Fabric

Arati Baliga, Nitesh Solanki, Shubham Verekar, Amol Pednekar, Pandurang Kamat and Siddhartha Chatterjee

Persistent Systems Ltd.
Bhageerath, 402 E, Senapati Bapat Road, Pune, India
Contact: blockchain@persistent.com

[1] ***Abstract*—Hyperledger Fabric is a permissioned ledger platform designed to be highly modular and extensible, delivering confidentiality, privacy and scalability to enterprise blockchains. With Fabric's production grade availability in mid-2017, enterprises are experimenting with Fabric for building real-world blockchain applications.**

In this paper, we characterize the performance and scalability features of the current production release of Fabric (v1.0). This paper takes an experimental approach, where we study the throughput and latency characteristics of Fabric by subjecting it to different sets of workloads. Through a suite of micro-benchmarks, custom-built for Fabric, we tune different transaction and chaincode parameters and study how they affect transaction latencies. Finally, we also conduct experiments to study Fabric's performance characteristics while increasing the number of chaincodes, channels and peers.

I. Introduction

Blockchain technology has opened up opportunities for new kinds of applications that enable elegant data sharing across organizational boundaries where all entities can collectively own and manage the shared data. Though often confused as an alternative for relational databases or big data solutions, blockchains are not a solution or a replacement for them. Blockchains are particularly attractive for applications that require multi-party reconciliation, trusted intermediaries, and high degree of transparency, auditability and integrity. While blockchains are currently popular in the public permissionless space with Bitcoin [1], Ethereum [2] and other cryptocurrencies making headlines [3]–[5], enterprise blockchain applications are emerging, and may soon be deployed at production scale.

Blockchain platforms enable speedy development of applications by abstracting out the low-level details about transactions, blocks and ledger structure, underlying protocols and consensus models, allowing developers to focus on application design and development. The success of real-world applications will depend on providing a frictionless user experience as well as good performance and scalability.

Hyperledger Fabric [6] is one of the popular permissioned blockchain platform hosted by the Linux Foundation's Hyperledger project [7]. This platform is built for business consortia interested in building and deploying their blockchain applications with shared data. It is designed to meet the confidentiality, privacy and scalability requirements of applications. Fabric requires that the participating organizations are known a priori and that each participating organization is provisioned into the network during blockchain set-up. While Fabric is built with scalability in mind, little data exists on what kind of performance and scalability one can expect from this platform while targeting real-world applications. Through this work, we strive to shed some light on these questions.

This paper makes the following contributions:

- We present an experimental approach to characterize the throughput and latency of Hyperledger Fabric 1.0, the first production grade release.
- Through a suite of micro-benchmarks custom-built for Fabric, we study its performance characteristics, thereby highlighting different aspects of transaction and chaincode parameters that have an impact on transaction latencies.
- We also perform scalability experiments where we study how Fabric performs when scaling chaincodes, channels and the number of peers.
- The toolkit that we built and used in our experiments is open-sourced and publicly available for use by the community [8].

This paper is organized as follows. In Section II, we provide a brief overview of Hyperledger Fabric 1.0 describing its architecture, transaction flow, ordering and chaincode execution mechanisms, providing enough background for the reader to follow the experiments described in the rest of the paper. In Section III, we characterize the throughput and latency of the Fabric platform with controlled workloads. In Section IV, we describe our micro-benchmarking experiments that measure the effect of tuning different transaction and chaincode parameters on transaction latencies. We cover some scaling experiments in Section V, describe related work in Section VI and finally describe some key insights and conclude in Section VII.

II. Hyperledger Fabric Overview

Hyperledger Fabric hosted by the Linux Foundation was the first consortium blockchain platform with production-

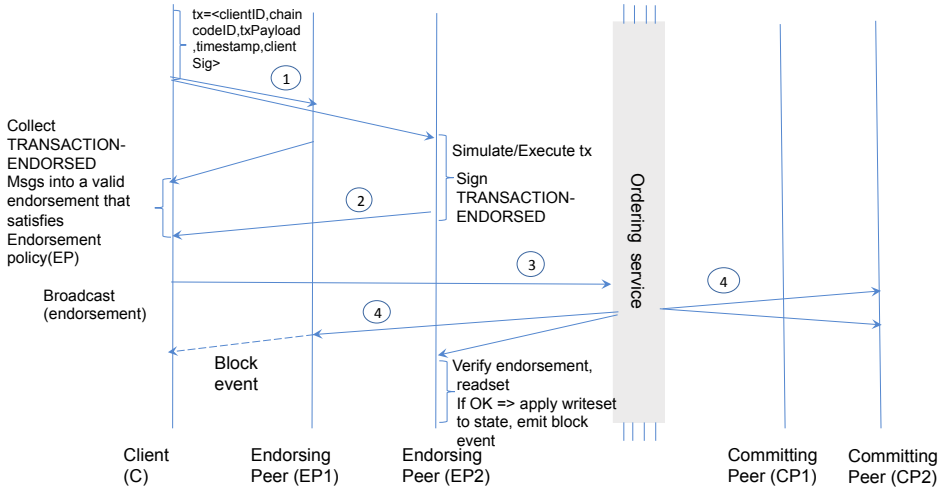


Fig. 1: Fabric transaction flow diagram

grade availability of its offering. Subsequently, Multichain [9], CORDA [10] and Quorum [11] also released their production-grade offerings. Fabric with its smart contract support is generically well-suited for a range of applications across several domains. A more detailed understanding can be obtained from the Fabric documentation available online [12].

Fabric enables participating organizations within consortia to build and deploy blockchain applications. The blockchain network consists of several nodes (or peers) that host the blockchain, execute smart contracts (known as chaincode) and collectively maintain the state of the ledger. Chaincodes can be shared by all entities within a consortium or could be privately deployed to be accessible to a subset of entities. Private chaincodes are run only on peers with whom the chaincode is shared and is inaccessible to others. This is achieved via a concept of channels in Fabric where all chaincode and data on the channel is only accessible to entities that are part of the channel. In the setup phase, the peers need cryptographic material that is generated to identify and authenticate the peers to the blockchain network. In this way, it can be determined whether a given peer belongs to a particular channel. In addition to peers, the Fabric network also needs an ordering service/orderer. The ordering service performs a total ordering of the transactions accepted by the Fabric network on a per-channel basis. The current production version does not support any form of consensus algorithm for ordering. It is expected to be incorporated in the future versions. Note that the older version of Fabric (v0.6) supported Practical Byzantine Fault Tolerance [13] based consensus, which was later removed in the production version.

Transactions in Fabric are invocations of chaincode methods. The chaincode itself is run within a Docker container thus isolating itself from the Fabric code as well as other chaincodes running on the same peer machine. Each chaincode has a persistent state called the key-value store. Chaincode methods manipulate the values of the key-value store using `put` and `get` methods that essentially allow it to write and read from the key-value store. The key-values are stored internally within a LevelDB database [14] on the same node. Fabric also has support for CouchDB [15] as an alternate database implementation that can be used to store key-value pairs.

Transactions in Fabric go through the following steps as shown in Figure 1:

1) *Client initiates a transaction:* A client prepares a request proposal to invoke a chaincode function. The request is signed by the client and sent on the channel where the chaincode is deployed. The number of endorsements that it expects to receive is as per the endorsement policy of the chaincode.
2) *Endorsing peers verify signature and execute the transaction:* The endorsing peers perform all the validity checks for well-formedness, authenticity, replay-protection and client authorization. If all checks are successfully cleared, the peers execute the transaction against their own key-value stores and produce a response that include read-write sets generated as a result of chaincode execution. These values, signed by the peers are sent back to the client as a proposal response or endorsement. No changes to the ledger are made at this point in time.
3) *Client collects endorsements and sends to the ordering service:* The client examines and compares all the endorsements and verifies that it has met the endorsement policy requirements of the chaincode. If the request was a read request, it does not send a request to the ordering service. If the request is a chaincode invoke (or write), it assembles the endorsements into a transaction and sends it to the ordering service for inclusion into the blockchain. The ordering service verifies transactions and orders them per channel.
4) *Transaction is validated and committed:* Ordered transactions within blocks are delivered to all peers on the

channel by the ordering service. The peers verify the transaction and endorsement policy fulfillment; if all checks go through, the peers add the block to the ledger. Note that all peers have to commit the transaction (and therefore play the role of a committing peer), while endorsement can be delegated to only a subset of peers on the channel and are referred to as endorsing peers (EP), as shown in the Figure 1.

The ordering service can be run in two modes - the solo mode and the kafka mode. In the solo mode, a single ordering service node performs the ordering of all transactions in the network. In the kafka mode, ordering service nodes can be distributed and use a Kafka cluster to produce and consume transactions. A pub/sub topic in the Kafka cluster corresponds to a channel within the blockchain network. The kafka mode provides crash fault tolerance and is therefore recommended in real-world deployments.

III. Characterizing Latency and Throughput

Transaction throughput is defined as the number of transactions per second successfully processed by the blockchain network. A transaction is successfully processed when it is included in a block and committed as part of the ledger. *Transaction latency* is the time elapsed between when a request is sent, to the time when the response is received by the client. For read transactions, it is the time taken to receive the response for a read query. For invoke transactions, it is the time elapsed between the request and an event confirmation as received by the client after the transaction is confirmed on the blockchain.

A. Experimental Setup

Fig 2 shows the experimental setup we used in all our experiments (exceptions are noted in the respective section). A blockchain consortium was setup with two organizations Org 1 and Org 2; each contributed two peers to the blockchain network. The ordering service was run on a separate node run by a third-party Org 3; Org 3 being a neutral entity. The endorsement policy on transactions was set to include signatures from at least one peer from each organization to successfully commit on the blockchain. Both organizations had a single channel that was set up between them and all chaincodes were deployed on this channel. The Ordering Service Node (OSN) was run in the solo mode. The ordering service parameters used were the default parameters, where the `BatchSize` was 500 and `BatchTimeout` was 1 second. The peers were run on four hardware machines within our local network. Each machine had 8 vCPUs (4 cores at 3.6 GHz with hyperthreading) and 16 GB RAM. We used four more machines with the same configuration to run the clients. All nodes had the Ubuntu 14.04 LTS operating system and were connected to each other with a 1 Gbps switch.

B. Client configuration

We use the Caliper benchmarking tool [26] developed by Huawei Technologies to generate the client workload. Using

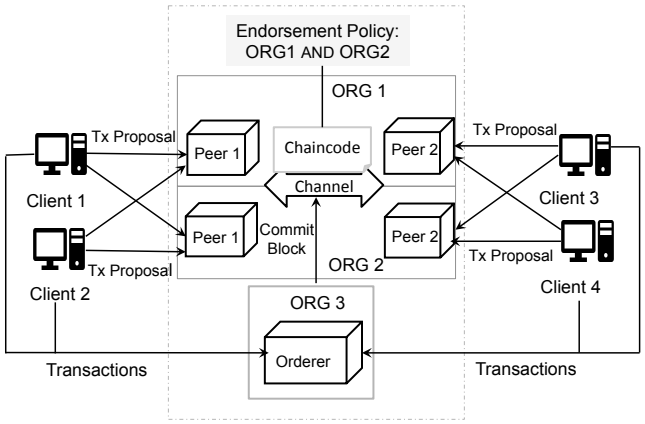


Fig. 2: Experimental setup

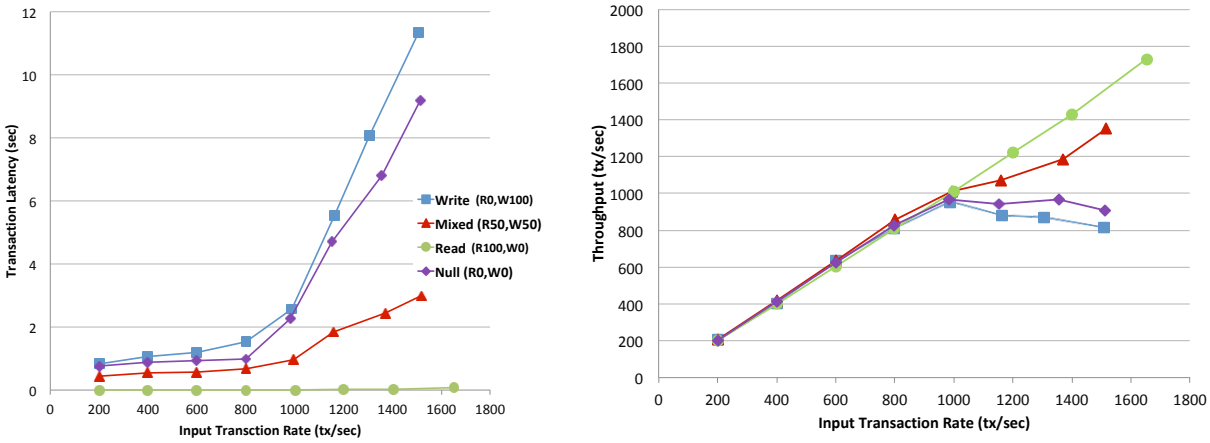
Caliper, we can send controlled workloads to the blockchain platform and measure the resulting transaction throughput and latencies. Caliper runs on the client machines and broadcasts transactions on the Fabric channel. It listens to block events from peers to check for transaction confirmations on the blockchain and assigns those transactions a completion timestamp. It calculates transaction throughput and latencies using the transaction timestamps.

We made the following changes to the Caliper code to be able to successfully launch controlled workloads at high send rates:

1) The load generated by the client was equally balanced across all the consortium peers within the consortium network.
2) The Caliper client was missing out certain block events at higher transaction rates resulting in failed transactions. This was because the Caliper client itself was single-threaded and was performing both functions of sending transactions as wells as processing block events. We modified the client to spawn a new process that essentially splits the the two functions into separate processes. A newly spawned process only listens to block events and inserts them in a messaging queue to be processed later by Caliper's main process. By having a separate process listen continuously to block events, we have completely eliminated occurrence of failed transactions due to missed block events at higher transaction rates.

C. Load Generation

Caliper is run on all the client machines. Each client sends transactions to at least one peer from each organization within the consortium. Each experiment sends transactions with a send rate starting from 25 tx/sec to 400 tx/sec, which was the maximum capacity for client nodes used in our experiments. The blockchain network therefore collectively experiences a transaction load ranging from transactions received by all the clients. Each client sends transactions at a specified send rate,



(a) Transaction latency

(b) Transaction throughput

Fig. 3: Latency and throughput measurements for varying input load

halts for 5 secs, and starts the next round. The experiment is repeated for three rounds. At the end of the third round, an average of the throughput and latencies is calculated. The blockchain network is subject to a maximum of 48000 total transactions. We also note the CPU utilization and memory consumed on the peers for the duration of the experiment.

D. Workloads

For all workloads, the chaincode is deployed and pre-loaded with key-value pairs. We used the following workloads.

- *Write workload (R0,W100):* The write workload comprises of all write transactions that update a value for a randomly selected key in the key-value store of the chaincode. In Fabric, this involves calling the `invoke` function within the chaincode. Write generates a transaction that goes through the three phases of endorsement, ordering and commitment.
- *Null workload (R0,W0):* The null workload comprises of transactions that call a function within the chaincode that simply returns. The null `invoke` function also goes through the same three phases of endorsement, ordering and commitment and therefore represents a baseline of how write functions perform.
- *Read workload (R100,W0):* The read workload comprises of read transactions that read the values for randomly selected keys from the key-value store within the chaincode. The read workload is generated by all clients sending their transactions to a single peer. This design is intentional as reads are served locally by the peer by performing lookups within its local database. Reads do not generate a transaction that confirms on the blockchain.
- *Mixed workload (R50,W50):* Read-write workload has a 50-50 mix of reads and writes. This mix submits all transactions with the same endorsement policy as used for writes. While the write transactions undergo the normal transaction flow, read queries are sent to, and need to receive responses from two peers (instead of one as in the case of the read workload) as per the endorsement policy of the chaincode.

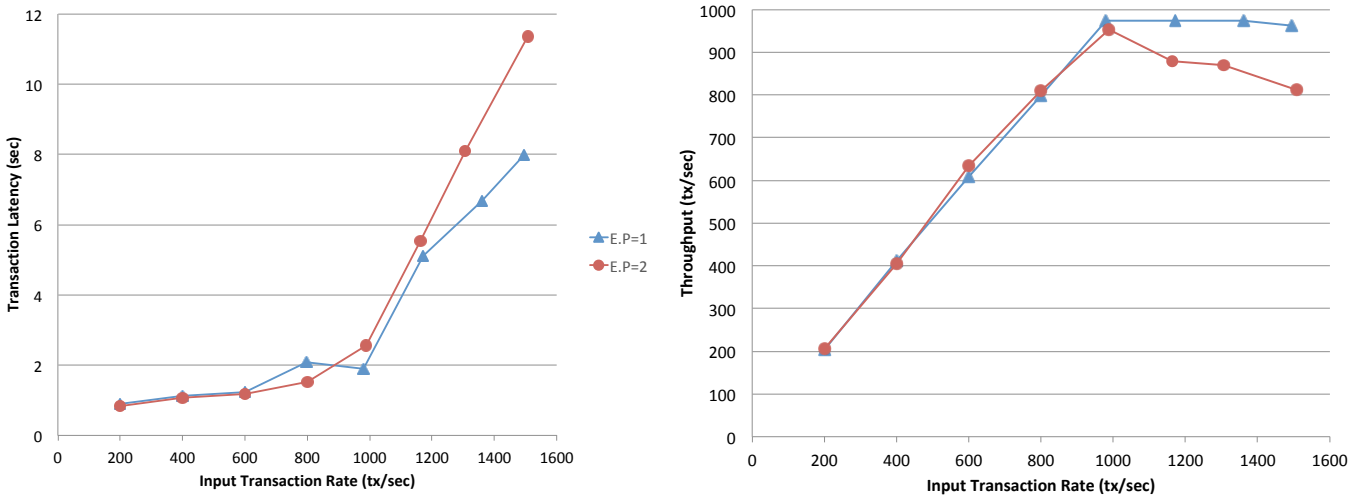
E. Throughput and Latency Measurements

Figure 3 shows the latency and throughput measurements for all the four workloads. The maximum load of 1600 tx/sec was generated on the blockchain network with four clients, each generating a load of 400 tx/sec. The results show that for the given range of transaction rates, read throughput scales linearly for the entire range. This is expected as reads are served locally by the peer machine from its LevelDB database, which is highly optimized for lookups. It is also seen from the figure that the write workload latencies mimic the null workload, clearly indicating that the delays involved are mainly in completing the three phases of the write transaction, with a smaller fraction of the time spent in actually updating the key-value state. This is also partly true because each transaction performs only a single write operation into the key-value store. We study how the number of key-value entries read and written affect transaction latencies in Section IV-A.

For the write workload, the throughput rises almost linearly until a load of 1000 tx/sec, with 968 tx/sec being the highest achievable throughput. When the load is increased beyond this point, the performance starts to degrade. The write workload performs almost the same as the null workload closely up until a load of 1000 tx/sec, after which the write workload gives a slightly lower throughput compared to the null workload. Additionally, after 1000 tx/sec, transaction latencies experience a steep increase. Note that these experiments were carried out with the default orderer settings as described in Section III-A.

F. Tuning Orderer Batch Size

The orderer `BatchSize` parameter dictates the number of transactions that get bundled into a block. To study the effect of throughput degradation after a transaction load of 1000 tx/sec, we tuned the orderer `BatchSize` parameter (specifically changing `BatchSize.MessageCount` and tuning



(a) Transaction latency

(b) Transaction throughput

Fig. 4: Effect of endorsement policy on transaction latency and throughput

out the other parameters) to check if a larger batch size helps in improving the throughput. The ordering service uses other parameters within `BatchSize` and `BatchTimeout`, which together control when blocks are created from transactions. The `BatchTimeout` is the amount of time the Orderer waits before creating a block irrespective of number of transactions in it. The orderer creates a block when either the `BatchSize` has the specified number (or size) of transactions in it or the time window has elapsed in which case it batches all available transactions at that point in time. By default, the `BatchSize.MessageCount` is set to 500 and the `BatchTimeout` is set to 1 sec by Caliper.

Input Load (tx/s)	Batch Size	Throughput (tx/s)
1200	250	731
1200	500	801
1200	1000	1161
1200	1200	917
1400	500	869
1400	1400	1227
1600	500	812
1600	1600	1078

TABLE I: Effect of Orderer `BatchSize` on throughput

In this set of experiments, we tried to alter the `BatchSize` of the orderer to accommodate larger number of transactions within a block for higher transaction loads. We tune out the `BatchTimeout` parameter by setting it to a very high number. Table I shows an improvement in throughput for the higher input transaction loads of 1200, 1400 and 1600 where the throughput was declining after 1000 tx/sec. This indicates that the orderer BatchSize setting has a significant influence on the resulting throughput of the system. On the other hand, having a smaller `BatchSize` (e.g. refer to Row 1 of Table I) reduces throughput as more number of blocks and block events are generated. In Fabric, this setting is specified when the orderer is bootstrapped and can be updated at a later time. Fabric could benefit significantly if this parameter could be tuned dynamically at run-time by the orderer based on the perceived instantaneous total load on the system.

G. *Effect of Endorsement Policy*

In the previous experiments, the endorsement policy of the chaincode was set such that one peer from each organization needed to endorse every transaction. This represents a basic chaincode between two business entities where both the organizations need equal amount of control in admitting incoming requests. Hence, the client has to send the transaction and await responses from two endorsing peers (EP).

In this experiment, we study the effect of endorsement policy on the latency and throughput at different send rates. While it is expected that there will be an decrease in the latency and an increase in throughput with fewer endorsements, we wanted to quantify the impact. For these set of experiments, we changed the endorsement policy such that a single peer from either organization could endorse the transaction. The resulting latency and throughput measurements are shown in Figure 4. It can be seen that for up to an input load of 1000 tx/sec, there is almost no difference either in throughput or latencies with either endorsement policy. Beyond that point, the transactions with single endorsements perform better that those that need two.

Though tweaking the order batch size and endorsement policy improves throughput after the input rate increases beyond 1000 tx/sec, the throughput does not increase linearly. We see that the bottleneck exists at the committing peer as this version of Fabric does not utilize all available CPU cores to commit transactions in parallel. Therefore at higher rates, the throughput plateaus to the maximum of what a single core can process per second.

IV. Microbenchmarking Experiments

In these sets of experiments, we use our custom-built suite of micro-benchmarks, which tune different transaction

and chaincode related parameters to observe their effects on transaction latencies. The suite has a total of five benchmarks that vary read-write set sizes, populate different size key-value stores, vary chaincode and event payload sizes, measure the chaincode execution overhead and perform chaincode to chaincode invocations.

For each of the experiments described below, we run the micro-benchmarks and record transaction latencies averaged across a large number of runs. These experiments used only a single client to submit transactions and a single peer which acts as the endorser as well as the committing peer. A separate orderer running in the solo mode orders the transactions. The orderer setting is such that it immediately creates a block when a single transaction is received. End-to-end latencies are measured by sending one transaction at a time. The latencies are broken down into three components which show the time taken for the three different steps as measured by the client. The c2e latency is the amount of time taken to send the transaction to the endorsing peer and receive a response for it. The e2o latency is the time taken by the client to contact the ordering service and get an acknowledgement. The o2v latency is the time taken for the transactions to be ordered to form blocks and confirmed by the committing peers.

A. *Transaction Read-Write Set Size*

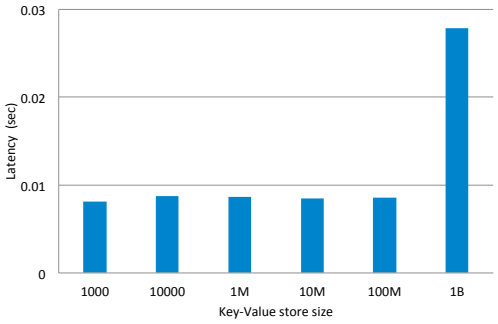
Each blockchain transaction contains a read-write set. The read set includes the set of keys that are read by the transaction and the write set contains a set of key-value pairs that are written by the transaction. A read-write set of 2x entries comprises of a read set of x entries and a write set of x entries as indicated in column 1 of Table II.

The goal of this experiment was to study how transaction latencies vary with increasing read-write set sizes. This experiment was conducted as follows. The key-value store is initialized prior to the experiment. The client generates the appropriate load by calling the chaincode function to perform the correct number of reads and writes to the chaincode key-value store.

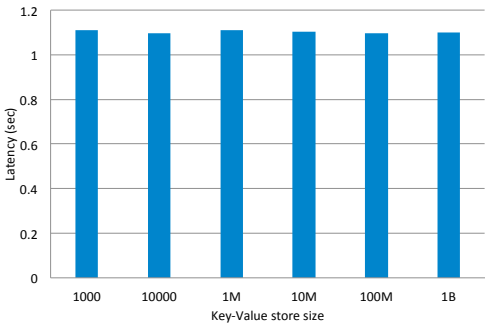
RW Set size (#entries)	Latencies (s)			
	c2e	e2o	o2v	Total latency
2 x 1000	0.238	0.008	0.120	0.366
2 x 0.3 Million	73.520	0.953	5.680	80.153
2 x 0.6 Million	147.650	1.887	12.630	162.167
2 x 1 Million	252.769	3.076	25.472	281.317

TABLE II: Transaction latency variation with RW set size

Table II shows how the latencies vary with different sizes of the read-write set. The maximum latency is the c2e latency. This is time consuming because the endorsement peer runs the chaincode to generate the read-write set. The larger the number of entries read or written, the longer it takes to execute the chaincode. The other two latencies, i.e. e2o and o2v are not as sensitive to the size of the read-write set. The total latency however is quite significant for a read-write set size of 2 million entries.



(a) Read latencies



(b) Write latencies

Fig. 5: Read and write latencies for varying sizes of key-value store

B. *Chaincode Key Value Store Size*

This experiment is designed to answer the question: Does the size of the key value store affect read and write latencies of chaincode data? To answer this question, we pre-populated chain code data ranging from 1000 entries to 1 Billion entries in different experiments. We recorded transaction latencies for individual reads and writes over 1000 separate transactions. The reads and writes were uniformly spread across the entire data set. The experiment was repeated for different key-value store sizes.

Figure 5a shows that reads are relatively unaffected by the key-value store size up until 100 Million entries[2]. This shows that peers use a highly optimized database for key-value lookups and updates. At 1 Billion, the read latency almost doubles and 53.8 GB of the disk is utilized by the KV-store (as opposed to only 5.17 GB at 100M). The increase in latency is due to many of the reads include fetching key-value entries from disk. Figure 5b shows the write latencies, which do not show any consistent pattern. Write latencies are dominated mainly by network latencies involved in completing the three

[2]Note that read latencies are in milliseconds while write latencies are in seconds.

stages of the transaction lifecycle[3].

C. Chaincode and Event Payload Sizes

Each chaincode function can be passed a payload during invocation. The payload is processed by the function, which might in turn generate an event. Events are generated by chaincode functions that clients can attach listeners to. The goal of this experiment is to study the effect of payload sizes on the transaction latency. These experiments are set up to measure the latencies for two cases: (a) Varying payloads are sent to a chaincode function that uses the payloads as values to be inserted in the chaincode KV store, and (b) In addition to updating the KV store, the payload is passed back in an event generated from the chaincode. The payload is passed from the peer to the client listener.

Payload Size(MB)	c2e(s)	e2o(s)	o2v(s)	Total latency(s)
1	0.114	0.101	0.221	0.436
10	0.902	0.981	1.627	3.510
20	1.791	1.958	3.045	6.794
30	2.747	2.899	4.936	10.582
40	6.432	3.861	10.778	21.071

TABLE III: Transaction latency with variable sized chaincode payloads

Payload Size(MB)	c2e (s)	e2o(s)	o2v(s)	Total latency(s)
1	0.132	0.149	0.296	0.577
10	1.092	1.454	2.442	4.988
20	2.117	2.896	4.585	9.598
30	5.323	4.335	10.497	20.155

TABLE IV: Transaction latency with variable sized event payloads

Tables III and IV summarize the findings in both sets of experiments. The results show that the transaction latencies significantly increase with each 10MB increase in the payload in both cases with most amount of time being spent in ordering and committing the entry into the ledger.

D. Chaincode runtime overhead

Fabric runs each chaincode within a separate Docker container [24]. Docker ensures isolation between different chaincodes running on the same machine. When multiple chaincodes are invoked concurrently, the peers run multiple chaincodes within their respective chaincode containers. The goal of this experiment is to quantify the runtime overhead of the Docker container by comparing the latencies of the chaincode running within Fabric versus running natively on the same machine. This experiment uses sorting as a representation of a cpu-intensive task performed within the chaincode. A chaincode function initializes a large array of 100 million numbers in the descending order and subsequently sorts it using the Quicksort algorithm. The same GoLang code is run on the natively on the same machine and latencies are measured for that operation. The Docker container adds an execution overhead of $<3.1\%$.

[3]This experiment was run with different orderer settings to accommodate the 1Billion data point, which took about 48 hours to populate. For the same reason the write latencies for all the data points are slightly higher than actuals.

E. Inter Chaincode Calls

Modularized real-world business logic often requires a system of chaincodes to inter-operate and produce the desired result. Each module within the application might be mapped to an individual chaincode. Such inter-operation often requires chaincodes to invoke methods within other chaincodes to complete a transaction. In this set of experiments, we study how multi-level invocation of chaincodes compares with a single monolithic chaincode incorporating all the functions. Experiments are conducted for invocation depths of 2, 3 and 4. A depth of 1 indicates a client invoking a chaincode function.

	Chaincodes		
Call depth	Single (sec)	Multiple (sec)	Overhead (%)
2	0.094	0.101	7.45
3	0.093	0.105	12.90
4	0.096	0.111	15.60

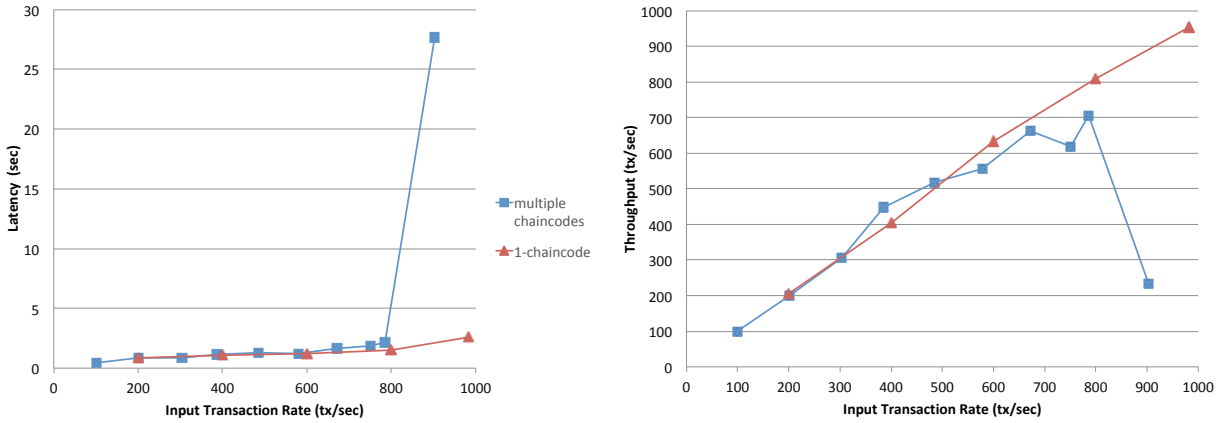
TABLE V: Inter-chaincode versus single chaincode transaction latencies

For multi-chaincode invocations, the experiment generates a transaction where a random key-value pair is generated by the client and sent to the first level chaincode C1. Depending on the call depth, this key-value pair is passed by chaincodes to the last level chaincode which updates its state. All chaincode invocations are treated as part of the same transaction on the blockchain. Our experiment measures the latency of this transaction. For the single chaincode scenario, all functions are implemented within the same chaincode. The same functionality is retained and the latency for this transaction is measured. Table V shows the actual latencies measured for call depths of 2, 3 and 4 in both scenarios. The chaincode to chaincode scenario incurs an execution overhead between the range of 5.2% to 7.45% for each added level of call depth.

In the above experiments, all chaincodes were deployed on the same channel. We also studied the latencies of chaincode to chaincode invocation of depth 2 on the same channel and compared it to one chaincode invoking another chaincode on a different channel. We noticed that inter-channel invocations did not add any additional overhead to the transaction latency. The current version of Fabric supports only read calls across channels and therefore we were able to only experiment with inter-channel chaincode-to-chaincode read calls.

V. Scalability Experiments

The scalability experiments study the limits on the system. In this section, we present results of experiments on scaling the number of chaincodes, maximizing the number of channels and increasing the number of peers to represent large consortiums on the blockchain network.



(a) Latency

(b) Throughput

Fig. 6: Scaling the number of chaincodes

A. Scaling chaincodes

Chaincodes encode business logic required to be executing as part of the blockchain application, making changes to the data (key-value state) stored within it. In this experiment, we study how the number of chaincodes deployed affect the throughput and latency of the system when all of them are deployed on the same channel and all (or most) of them are invoked at the same time by clients.

The experiment is setup with four peers using the hardware setup mentioned in Section III-A. Two clients are used to generate the input transactions to the system with a balanced load across all the peers. The number of transactions is increased to match the number of deployed chaincodes such that for each data point, roughly one transaction is sent to each deployed chaincode in the system. For example, when there are 100 chaincodes in the system, we use an input send rate of 100 transactions per second where each request is sent to one chaincode. When each chaincode is invoked within the same time window, the peer runs the chaincode and executes the invoke function, thereby consuming resources on the peer nodes.

Figure 6 shows the latency and throughput of the system when the number of chaincodes deployed is the same as the total transaction rate on the blockchain network. The throughput almost matches the system throughput when a single chaincode is deployed up until 786 tx/sec. When the number of chaincodes exceeds 900 and the transaction rate is above 900 tx/sec, we see a sharp increase in the latencies and the throughput drops significantly. The maximum number of chaincodes we were able to deploy with this setup was 1000, out of which only 903 could be successfully invoked concurrently. Increasing the chaincode count at this point caused a failure in transactions and timing out of the RPC connection between client and peers.

B. Scaling channels

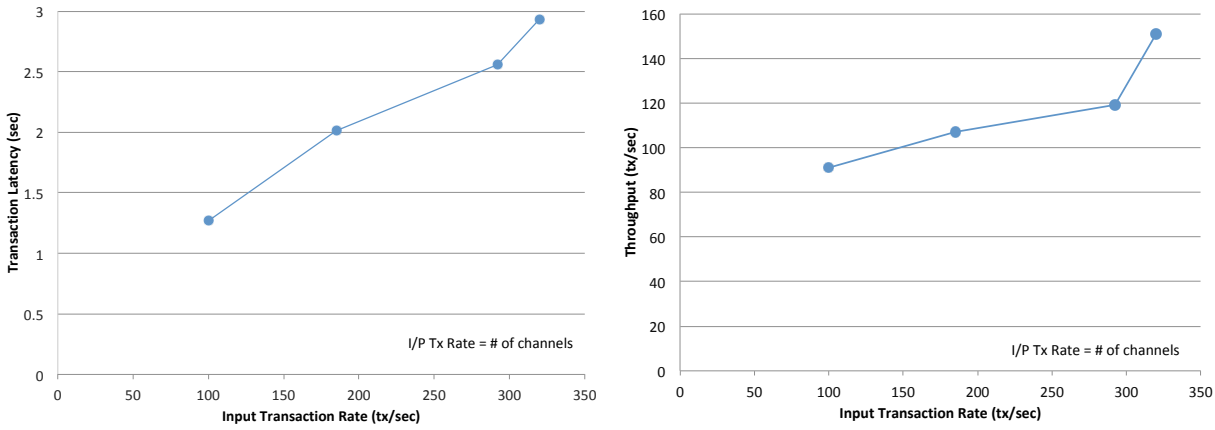
Channels in Fabric can be set up for confidentiality and privacy of transactions amongst a smaller subset of transacting parties within a large consortium. It allows several entities, some of whom might be potential competitors, to co-exist on the blockchain network. The chaincodes deployed on a channel execute only on the concerned peers that are members of the channel. Members also participate in advancing the blockchain and therefore maintain a copy of the ledger. The ledger and chaincodes are completely invisible to other members who are not part of the channel.

In this experiment, the goal was to identify the maximum number of channels that could be created with our hardware setup as described in Section III-A. Two clients were used to generate the input transaction load on the system. The chaincodes are pre-instantiated with one chaincode per channel. All chaincodes are invoked in parallel by the client load. For example, when there are 100 channels in the system, an input send rate of 100 transactions per second will invoke all the 100 chaincodes, by sending each transaction to a separate chaincode.

The maximum number of channels that we were able to create was 320. Figure 7 shows the throughput and latency of the system when the number of channels deployed matches the total input transaction rate on the blockchain network. The throughput degrades ranging from 9% reduction at 100 tx/sec upto 52.8% degradation when the number of channels is increased to 320, compared to the single channel scenario. Latencies also correspondingly increase compared to the single channel scenario (Figure 3).

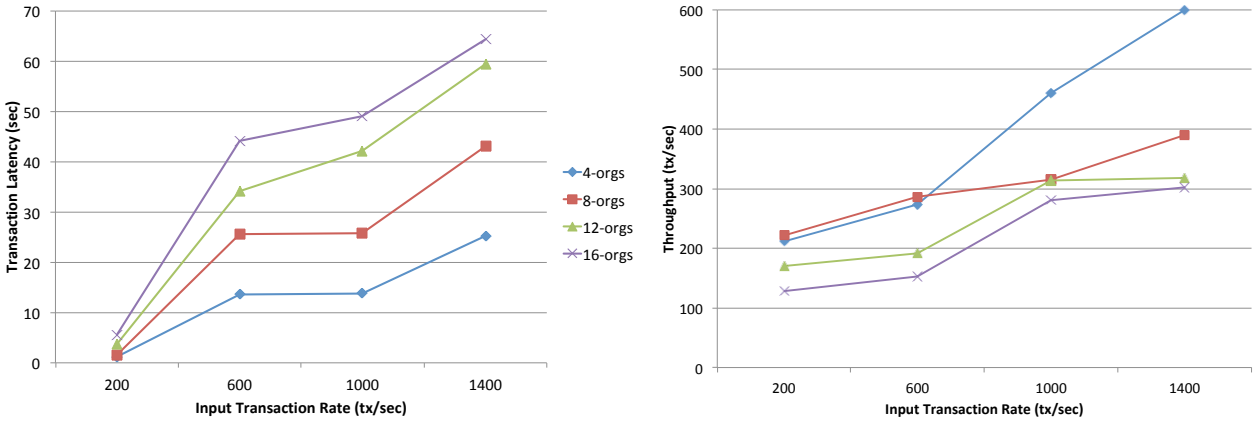
C. Scaling peers

Peers are run by participating entities within the consortium. For larger consortiums, considering each organization is a partner, ideally each of them would run at least one peer to contribute to the blockchain. In this set of experiments, we study the effect on the performance of the system when



(a) Latency (b) Throughput

Fig. 7: Scaling the number of channels



(a) Latency (b) Throughput

Fig. 8: Scaling the number of peers

consortium sizes are large. We use AWS EC2 [19] dedicated instances of type *c4.2xlarge* (8 vCPUs and 16 GB memory) [18] as peers on the Fabric blockchain network.

We evaluate the latency and throughput for consortium sizes of 4, 8, 12 and 16. The endorsement policy is set such that all peers within the consortium setup on a single channel have to endorse a transaction for it to be included on the blockchain.

Figure 8 shows the throughput and latencies for transactions with different send rates up to 1400 tx/sec. For a lower transaction rate of 200 tx/sec, with default configuration parameters set, the largest consortium setting yields the lowest throughput and incurs the highest latencies. This gap significantly widens for higher throughput rates of 1400 where the 16-peer setup achieves about half of the throughput as the 4-peer setup.

VI. Related Work

The Caliper tool [26] was developed by Huawei Technologies to measure the throughput and latency of permissioned blockchain platforms. We have adapted Caliper to measure the latency and throughput results described in this paper. The modifications that we made to Caliper to tailor it to our requirements is explained in Section III-B. Blockbench [17] is proposed as a framework for benchmarking the performance of private blockchain platforms. However it focuses on the performance comparison of Fabric (v 0.6), Ethereum and Parity, where Ethereum and Parity are not private blockchain platforms. The micro-benchmarks proposed are generic and do not evaluate the intricacies of Fabric. To the best of our knowledge, ours is the first work that is focussed on characterizing the performance and scalability of the production grade Hyperledger Fabric 1.0 platform. A recent paper from the Hyperledger Fabric group [16], contains a couple of experimental results. The paper claims to achieve a throughput of 3000 tx/sec, on the Fabric 1.1-preview version, which has one significant improvement over v1.0 that we use in our experiments. Commitment of transactions in the 1.1-preview version is done in parallel where all the available cores can be used to leverage speedy commitment of transactions, which most likely results in the higher throughput results that are reported. In Fabric v1.0, the commitment phase uses only

one CPU core where requests are queued up and wait to get committed on the ledger.
Other work in this field has focussed mainly on improving the performance and scaling issues of public blockchains such as Bitcoin and Ethereum and other cryptocurrencies, where the issues involved are quite different from permissioned platforms [20]–[23], [25].

VII. Conclusions

In this paper, we presented a structured experimental approach to characterize the performance and scalability of the Hyperledger Fabric 1.0 blockchain platform. The throughput of the system is linear for reads. For writes, it is almost linear below a transaction rate of 1000 tx/sec (for our setup), after which the throughput degrades and transaction latencies increase significantly. We showed that the throughput of the system is sensitive to the orderer settings and therefore can be further improved if the system can dynamically tune these settings based on the total experienced load on the system. Reducing the number of endorsements needed (a chaincode setting) also provides an improvement in throughput and latencies at higher loads. While this may help performance, it lowers the security of the system by weakening its anti-collusion properties. It might however be acceptable in certain scenarios where higher levels of trust already exist; for example, an already trusted intermediary running peers on behalf of its enterprise clients or a single organization hosting peers for its subsidiaries.

A significant drawback in Fabric 1.0 is that the committing peer does not process transactions in parallel, failing to take advantage of multiple vCPUs present on the system and therefore becoming a significant bottleneck. We believe this issue is addressed and fixed in the latest release of Fabric (v1.1) [16]. This release should therefore provide an improvement in throughput.

Through our micro-benchmarking experiments, we showed that transaction latencies are significantly affected by the read-write set size of the transaction (Section IV-A), and chaincode and event payload sizes (Section IV-C), both of which are application-specific. At this time there is inadequate understanding of what typical blockchain application runtimes, data access patterns or payloads might be, but having an understanding of the latencies involved can guide application designers to make informed choices. We also show that read and write latencies are relatively unaffected by the size of the data stored in the chaincode (Section IV-B). This implies that designers can have considerable flexibility while choosing data set sizes. Modularity is another design consideration while mapping application modules to chaincodes. Through our micro-benchmark, we showed that inter-chaincode calls add an additional execution overhead of 5.2% to 7.45% for each added level of call depth (Section IV-E). Therefore, application designers need to make the right tradeoff between modularity and speed. Finally the scalability experiments (Section V) demonstrated good scaling characteristics for chaincodes and small number of channels, which is expected to be the most common case usage scenario. While large consortiums can be built, the endorsements per chaincode should be limited to a smaller subset of peers, with an eye on performance.

References

[1] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf, Dec 2008.

[2] Vitalik Buterin. Ethereum: A next-generation smart contract and decentralized application platform. https://github.com/ethereum/wiki/wiki/White-Paper, 2014.

[3] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers Eran Tromer, Madars Virza. Zerocash: Decentralized Anonymous Payments from Bitcoin. 2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014 https://doi.org/10.1109/SP.2014.36

[4] Monero. The Monero Cryptocurrency. https://getmonero.org/, 2018.

[5] The Litecoin Cryptocurrency. https://litecoin.com/.

[6] Hyperledger Fabric. Hyperledger Fabric. https://www.hyperledger.org/projects/fabric, 2017.

[7] Linux Foundation. Linux foundation hyperledger project. https://www.hyperledger.org/, 2017.

[8] Gauge - Performance Benchmarking Toolkit for Permissioned Blockchains. Persistent Systems Ltd. https://github.com/persistentsystems/gauge, May 2018.

[9] Dr. Gideon Greenspan. Multichain private blockchain - white paper. https://www.multichain.com/download/MultiChain-White-Paper.pdf, 2016.

[10] Mike Hearn. Corda: A distributed ledger. https://docs.corda.net/_static/corda-technical-whitepaper.pdf, 2016.

[11] J. P. Morgan Chase. A Permissioned Implementation of Ethereum. https://github.com/jpmorganchase/quorum, 2018.

[12] Hyperledger Project. Hyperledger Fabric Documentation. https://media.readthedocs.org/pdf/hyperledger-fabric/latest/hyperledger-fabric.pdf, 2018.

[13] Miguel Castro and Barbara 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] Level DB. Level DB Database. http://leveldb.org/, 2018.

[15] Couch DB. Couch DB Database. http://couchdb.apache.org/, 2018.

[16] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstatinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. Proceedings of Eurosys 2018, April 2018. https://doi.org/10.1145/3190508.3190538

[17] Tien Tuan Anh Dinh, Ji Wang, Gang Chen, Rui Liu, Beng Chin Ooi, and Kian-Lee Tan. BLOCKBENCH: A framework for analyzing private blockchains. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 1085–1100, 2017.

[18] Amazon EC2. Amazon EC2 Instance Types. https://aws.amazon.com/ec2/instance-types/, 2017.

[19] Amazon EC2. Amazon Elastic Compute Cloud. https://aws.amazon.com/ec2/, 2017.

[20] Ittay Eyal, Adem Efe Gencer, Emin Gün Sirer, and Robbert Van Renesse. Bitcoin-ng: A scalable blockchain protocol. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, pages 45–59, Berkeley, CA, USA, 2016. USENIX Association.

[21] Arthur Gervais, Ghassan O. Karame, Karl Wüst, Vasileios Glykantzis, Hubert Ritzdorf, and Srdjan Capkun. On the security and performance of proof of work blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 3–16, New York, NY, USA, 2016. ACM.

[22] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 51–68, New York, NY, USA, 2017. ACM.

[23] Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. A secure sharding protocol for open blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 17–30, New York, NY, USA, 2016. ACM.

[24] Dirk Merkel. Docker: Lightweight linux containers for consistent development and deployment. *Linux J.*, 2014(239), March 2014.

[25] Rafael Pass and Elaine Shi. Hybrid Consensus: Efficient Consensus in the Permissionless Model. In Andréa W. Richa, editor, *31st International Symposium on Distributed Computing (DISC 2017)*, volume 91 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 39:1–39:16, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[26] Huawei Technologies. Caliper: A Blockchain Benchmark framework. https://github.com/Huawei-OSG/caliper/, 2017.