Scaling Hyperledger Fabric Using Pipelined Execution and Sparse Peers

ABSTRACT

Many proofs of concept blockchain applications built using Hyperledger Fabric, a permissioned blockchain platform, have recently been transformed into production. However, the performance provided by Hyperledger Fabric is of significant concern for enterprises due to steady growth in network usage. Hence, in this paper, we study the performance achieved in a Fabric network using *vertical scaling* (i.e., by adding more vCPUs) and *horizontal scaling* (i.e., by adding more nodes) techniques. We observe that network scales very poorly with both of these techniques. With vertical scaling, due to serial execution of validation & commit phases of transactions, the allocated vCPUs are under-utilized. With horizontal scaling, due to redundant work between nodes, allocated resources are wasted though it is utilized. Further, we identify these techniques to be unsuited for dynamically scaling a network quickly to mitigate an overload situation, and hence, it results in a 30% drop in the performance.

To increase the CPU utilization and hence the performance, we re-architect Fabric to enable pipelined execution of validation & commit phases by introducing dirty state management using a trie data structure. Additionally, we facilitated the validation phase to validate transactions in parallel by introducing a *waiting-transactions dependency graph*. To avoid redundant work performed between nodes and to quickly scale up a network, we propose a new type of peer node called *sparse peer*, which selective commits transactions. Overall, we improved the performance by at least 3× and reduced the time taken to scale up a network by 96%.

ACM Reference Format:

. 2019. Scaling Hyperledger Fabric Using Pipelined Execution and Sparse Peers. In *Proceedings of* . ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

, ,

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Blockchain technologies gained popularity as it provides a way to get rid of the intermediary and decentralizes the application. A blockchain is a ledger that records transactions and data which is replicated across multiple peer nodes where one node does not trust another. Each node holds an identical copy of the ledger as a chain of blocks, with each block being a logical sequence of transactions. A block is created by executing a consensus protocol among the nodes, and each block encloses the hash of its immediate previous block, thereby guaranteeing the immutability of the ledger.

For an enterprise use-case, which involves interaction between multiple organizations, a permissioned blockchain network is suitable as it supports authenticated participants, and privacy for transactions, data, and users. Different organizations own nodes in a permissioned network, and transaction logics are implemented using smart-contracts. Hyperledger Fabric [20] is the most popular permissioned blockchain platform hosted by The Linux Foundation [8]. Many proofs of concept blockchain applications built using Hyperledger Fabric for various industries, such as Food Trust [7], Chain-m [2], openIDL [10], TradeLens [14], WeTrade [17], SecureKey [13], CLSNet [3], and everledger [5] to name a few, have recently been transformed to production.

However, the performance of Hyperledger Fabric is a significant concern for enterprises due to steady growth in the network usage. For example, to track the provenance of ingredients used in food products, such as protien bars, chocolates, other packaged foods, we need to store lots and lots of records at a high rate. In the current form, Fabric cannot provide the performance needed by large provenance use-cases and finance industry—stock exchanges, credit card companies, such as Visa [16], and mobile payment platform, such as AliPay [1] (a peak of 325k tps). Further, with innovations in tokens [15, 22] and related applications such as decentralized marketplace [4, 9, 11, 12], the throughput requirement is only going to increase. Hence, it is necessary to improve the performance of Fabric to support ongoing growth proactively. Even with the current performance provided by Fabric, nodes are overprovisioned to satisfy a peek load as there exists no mechanism to scale up/down a network. As a result, operational cost increases unnecessarily.

Although many recent efforts [23, 25, 27, 30, 34, 35] have proposed various optimizations to improve the performance of Fabric, none of them have studied the impact of scaling

techniques, such as *vertical* and *horizontal* scaling, on the performance of Fabric. Hence, in this paper, we study the efficiency of various scaling technique and identify bottlenecks. Then, we re-architect Fabric transparently to improve the performance. In general, the consensus layer is assumed to be the bottleneck. Though it is valid for the permissionless blockchain such as Ethereum [21], Bitcoin [31] due to Proof of Work (PoW) consensus, we found that the validation and commit of a block is the bottleneck in Fabric, not the consensus layer as it uses Raft [33] for ordering transaction. Our four major contributions are listed below:

(1) We conducted experiments to understand the scalability of Hyperledger Fabric using various techniques such as (a) scaling by vCPUs, (b) scaling by peers, (c) scaling by channels, (d) scaling by peers and channels, and (e) scaling by endorsement policy. We identified that these techniques either (i) scales a Fabric network poorly due to the serial execution of validation & commit phases, and duplication of CPU & IO intensive task or (ii) create data silos or (iii) result in a weaker trust model. Further, to scale a network dynamically for handling an overload situation, we identified these techniques to be unsuitable.
(2) We re-architected Fabric to enable pipelined execution of validation & commit phases without violating the serializability isolation. In addition to this, we also facilitated validation phase to validate multiple transactions, which are belong to different blocks, in parallel by introducing a *waiting-transactions dependency graph*, which tracks six types of dependencies between transactions. As a result, the performance improved by 1.36× while increasing the CPU utilization from 50% to 70%.
(3) We introduced a new type of node called *sparse peer*, which selectively commits transactions, that helps in avoiding the duplication of CPU & IO intensive task. Thus, the performance improved by 2.4×. Overall, our approaches improved the performance by at least 3×.
(4) We built an auto-scaling framework, which can split a full peer into multiple *sparse peers* or merge multiple *sparse peers* into a full peer. This helps in quickly scaling up a network to handle an overload situation and reduce the number of transactions invalidation. Our approach reduced the time taken to scale-up a network by 96% while increasing the scale-down time.

The remaining of this paper is structured as follows. §2 provides a background on Hyperledger Fabric and motivates our work by performing various experiments. §3 describes our proposed architecture for Fabric. §4 presents a framework that can aid the dynamic scaling of a Fabric network. §5 briefly describes the implementation details. §6 evaluates our proposed architecture against vanilla Fabric and showcase the improvement achieved. §7 presents related work while §8 concludes this paper.

2 BACKGROUND AND MOTIVATION

2.1 Hyperledger Fabric Architecture

Hyperledger Fabric consists of three entities—client, peer, and orderer. The transaction flow in Fabric involves all three entities and composes of four phases— simulation, ordering, validation, and commit, as depicted in Figure 1. Figure 2 shows the various components in a peer, which involves in the execution of simulation, validation, and commit phase. The communication between different entities happens via google remote procedure call (gRPC) [6].

Phase 1—Simulation. A client submits a transaction proposal to a peer to invoke a smart-contract which implements the transaction logic. The *endorser* validates and passes the input present in the proposal message to the appropriate *smart-contract*. Depending on the input, the *smart-contract* executes the logic and reads/writes the states it manages by issuing `GetState()`, `PutState()`, & `GetStateByRange()` calls back to the *endorser*. For a read request, i.e., `GetState()`, the *endorser* reads the state from the *state DB* and adds it to the read set of the transaction maintained at the *endorser* before passing it to the *smart-contract*. For a write request, i.e., `PutState()`, the *endorser* stores the value in the write set of the transaction without modifying the *state DB*. For a range read query, i.e., `GetStateByRange()` with a *start* and *end* key, the *endorser* returns an iterator to the smart-contract. By calling the `Next()` on the iterator, the smart-contract reads states. Instead of storing these states in the read set, the *endorser* stores the *start* key, *end* key, and a list of keys read in a separate data structure called *range query info*. Once the transaction execution completes, the *endorser* cryptographically signs/endorse the transaction response (which includes the transaction proposal, read-write set and range query info) before sending it to the client.

The client can submit the transaction proposal to multiple peers simultaneously depending upon the endorsement policy [29] defined for the *smart-contract*. For example, the `OR`(*OrgA.peer*, `AND`(*OrgB.peer*, *OrgC.peer*)) policy requires either a signature from organization A's peer or two signatures, i.e., from organization B's and organization C's peer. Each *smart-contract* maintains its states. One *smart-contract* can access/modify states maintained by another contract by invoking it. A *smart-contract* invoking another enables the application designer to split a monolithic contract into multiple micro contracts, each performing a specific task.

Phase 2—Ordering. The client submits the endorsed transaction response to the ordering service. An ordering service, which consists of orderer nodes from different organizations, employs consensus protocol [33] to order the received transactions and create a block. Each block has a sequence number called *block number*, hash of the previous block, hash of the current block, a list of ordered transactions

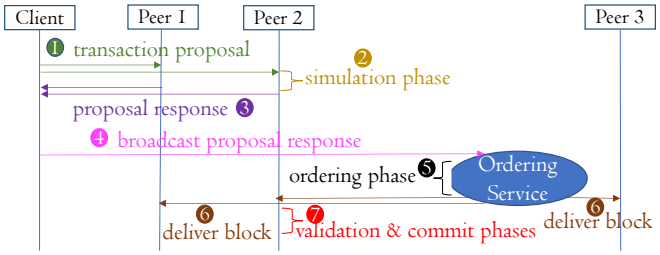

Figure 1: Transaction flow in Hyperledger Fabric

(i.e., commit order) and the orderer's signature. The orderer nodes broadcast the created block to peers.

Phase 3—Validation. The *gossip* component of the peer receives blocks and stores them in a *block queue*. The peer verifies the following two condition per transaction in a block before committing them: (1) collected endorsements satisfies the endorsement policy specified for the smart-contract; (2) transaction is serializable. The *endorsement policy validator* verifies the endorsements against the endorsement policy by checking whether the set of signatures on the transaction response is signed by the set of organizations specified in the endorsement policy. If a transaction invoked multiple smart-contracts (via a feature that allows a smart-contract to invoke another), the policy of each of the smart-contract must be satisfied. A transaction is marked invalid when it does not have adequate endorsements.

Once all transactions are validated by the *policy validator*, the *serializability validator* applies optimistic concurrency control (OCC) [28] using the read-write set present in the transaction response of valid transactions. To facilitate OCC, the Fabric adds a version identifier to each state stored in the blockchain. The version for a state is nothing but a combination of a block number and the transaction number within the block which last updated/created the state. All the validator checks for is if the state that a transaction has read, to decide on its write-set, have not been modified by preceeding valid transactions in the block and already committed transactions. Further, the validator re-executes range queries present in the *range query info* to detect any phantom read as it violates serializability property. Note that the *policy validator* parallely validates transactions in a block whereas the *serializability validator* serially validates each transaction such that it can take into account of write-set of previous valid transactions within the block while executing OCC.

Phase 4—Commit. After validation phase, first, the *committer* stores the block in the *block store*, which is a chain of blocks stored in a file system along with the transaction validity information. Second, the *committer* applies the write set of all valid transactions to the *state database*, which maintains all active states. Third, it stores all valid and invalid transactions' write set to the *history database*, which maintains both active and inactive states. The disk writes to *block store*, *state DB* and *history DB* are serial and synchronous to handle recovery after a failure. Note that the *history DB* only maintains an index in the form of state to a list of {*block number*, *transaction number* within the block} such that the actual history of each state can be retrieved from the *block store* using the transaction's write-set.

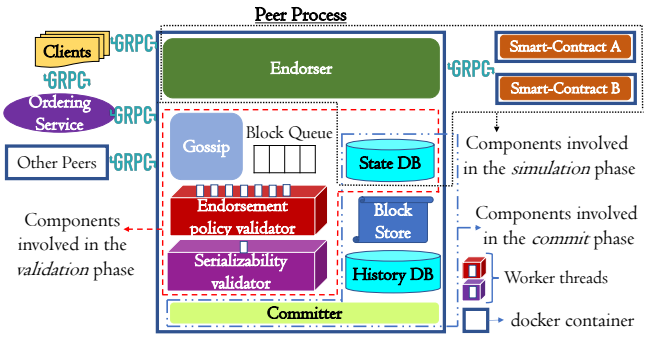

Figure 2: Various components in peer involved in simulation, validation, and commit phase.

Fabric blockchain network & channel. A blockchain network created using Fabric consists of multiple organizations each owning a set of clients, peers and orderer nodes. Fabric introduces a concept called *channel* to provide a private subnet of communication between organizations' peers. Transactions on a channel are only seen by the members of the channel. The states and smart-contracts are on a per-channel basis. Further, the consensus is applicable on a per-channel basis, i.e., there is no defined order for transaction across channels. Each channel maintains its blockchain, i.e., a separate chain of blocks on the file system and a separate database for holding the active states. As per the privacy requirements, a Fabric network can run multiple channels.

2.2 Scalability of Fabric and Bottlenecks

In this section, we present different methods to scale a Fabric blockchain network to get higher transaction throughput. Further, we quantify the scalability by conducting various experiments and find out bottlenecks which limits the scalability. Traditionally, a distributed system can be scaled vertically, i.e., by adding more power (vCPUs) to existing nodes or horizontally, i.e., by adding more number of nodes. In addition to this, the *channel* concept, and a simplification of endorsement policy also would help in scaling a network.

Setup. Figure 3 presents the blockchain network topology used for all experiments. It consists of four organization, each hosting N number of peer nodes, ordering service based on Raft consensus protocol [33] with five nodes, and M clients to generate load on the network. The value of M and N were changed depending upon the experiment. Each node is hosted on a virtual machine in a datacenter (refer to Figure 3 for resource configuration). We measure the throughput as the primary performance metrics. The *throughput* is the

Table 1: Parameters used to study the performance of a Fabric network using various scaling approaches

scaling by	*#peers per organization*	*#channels*	*per peer channel count*	*#vCPUs*	*endorsement*	*throughput*	*drawback*
vCPUs	1	1	$\lceil \frac{\#channels}{\#peers} \rceil$	1, 2, 4, 8, 16	complex	1790	underutilization of CPU
peers	1, 2, 4			16		2019	redundant work
channels	1	1, 2, 4, 8				3034	limited data access
peers and channels	1, 4					9964	absolute data silos
simple endorsement	1	1			simple	3120	weaker trust
our approach	4				complex	**6328**	—

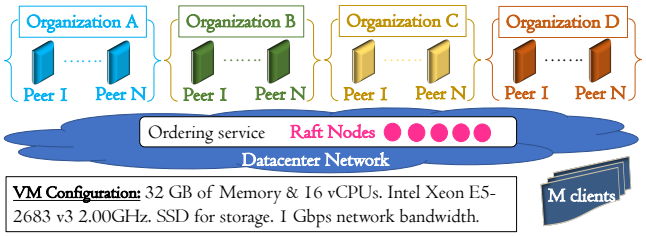

Figure 3: Experimental setup.

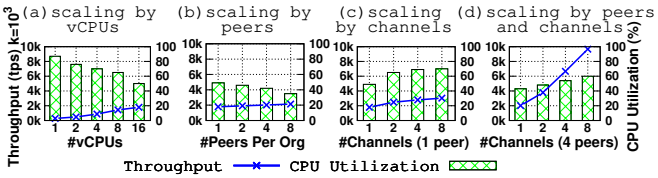

Figure 4: Performance of various scaling approaches.

peak rate at which transactions are committed in the network while maintaining a *latency* lesser than or equal to 500 *ms*. The latency is the time taken from a client sending the transaction proposal to the transaction commit. Note that we only use four organizations in a blockchain network as it represents a dominant number of use-cases. For example, in a supply chain solution, the participats would be a buyer, manufacture, supplier, and carrier. In a vechile insurance solution, the participants would be an insurance company, police department, repair shop, and auto part store. Moreover, our results hold for a network with a large number of organizations too. This is because the bottlenecks identified in this section are not related to network size. When the number of organizations increases, it majorly increases the network utilization between orderer and peers as orderer nodes need to send the blocks to multiple peers. This can be resolved by increasing the number of orderer nodes [20, 35].

Workload and Configuration. Unless specified otherwise, in the default configuration, each organization hosted a peer and all organizations are a member of a single channel hosting eight smart-contracts each implementing the smallbank benchmark [18]. We use smallbank (as defined by BlockBench [24]) as the primary benchmark throughout this paper as it is a popular benchmark for permissioned blockchain systems. The workload consists of 6 operations on a set of 100k accounts. Of the 6, 5 involve both reads and writes and one involves only reads. To generate load, we choose one of the operations uniformly at a specific rate. We used the following two endorsement policies: (1) `simple`—any one organization in the channel can execute and endorse the transaction; (2) `complex`—all organizations in the channel must execute and endorse the transaction. The block size was 100 transactions for all experiments. Tables 1 presents the parameter configuration used to study the performance of a Fabric network while employing six different scaling approaches (including our proposed approach in §3). Table 1 also summarizes the maximum throughput achieved along with the drawback of each approach.

Base Case Performance. Instead of using the vanilla Hyperledger Fabric v1.4 as the base case, we used an optimized version which avoids the redudant block serialization and deserialization at various phases within a peer using a block serialization cache as proposed in [26]. This is because, the Hyperledger Fabric community is aware of this and plan to fix it in the upcoming release. With the block cache, we observed an improvement in throughput by 1.15×, i.e., from 1555 tps to 1790 tps using the default configuration.

(1) Scaling By vCPUs. Figure 4(a) plots the throughput and CPU utilization over different number of allocated vCPUs. With an increase in the allocated vCPUs from 1 to 16, the throughput increased disproportionately from 280 tps to 1790 tps (an increment of 6.4×) while CPU utilization decreased from 87% to 50%. When the number of vCPUs was low, both the endorser and validator contended for the CPU resources, which increased the CPU utilization. With an increase in the number of vCPUs, the CPU contention reduced and the *endorsement policy validator* utilized multiple vCPUs to parallely validate transactions which resulted in higher throughput. Moreover, during the execution of commit phase, which is IO heavy, the validation phase, which is CPU heavy, does not execute, and vice-versa which resulted in an underutilization of CPU. This is because the validator cannot validate block $(i + 1)$ in parallel with the commit of block (i) as the state updates performed during the commit could make the validator to utilize the stale state. As mentioned in §2.1, the *serializability* validator validated each transaction serially which further reduced the CPU utilization.

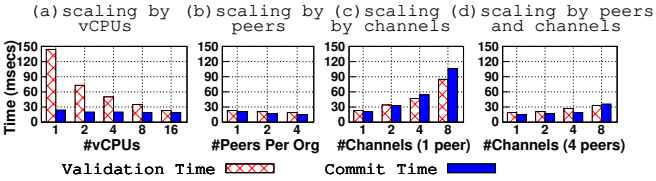

Figure 5: Time taken by the validation & commit phases.

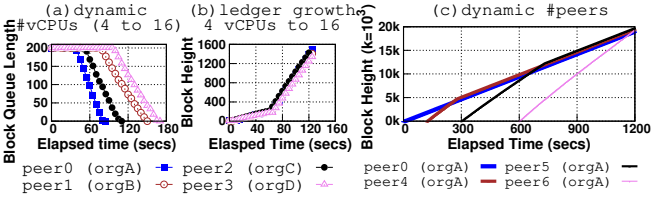

Figure 6: Impact of dynamic scaling by vCPUs & peers.

Figure 5(a) plots the time taken by the validation and commit phases to process a block of size 100 transactions. As expected, the time taken by the validation phase reduced significantly from 144 *ms* to 23 *ms* with an increase in the number of vCPUs. No such reduction was observed with the time taken by the commit phase (≈20 ms) as it majorily dependent on the IO performance not CPU.

Takeaway 1. *The vertical scaling of peers can help to a level, but it is necessary to have pipelined execution of validation and commit phases to utilize the full potential of allocated vCPUs.*

(2) Scaling By Peers. Figure 4(b) plots the throughput and CPU utilization over a different number of peers per organization. Compared to 1 peer per organization, 4 peers increased the throughput by only 1.1×. This is because all 4 peers within each organization validated and committed each block, i.e., redundant work within an organization. With an increase in the number of peers, only the endorsement for transactions were load balanced among the peers by clients, which also reduced the CPU utilization. As the validation and commit phases are costlier than the simulation phase due to multiple signature verifications and synchronous disk IO, the increment in the throughput is not significant. As load balancing of endorsement requests increased the throughput, we wanted to find the peak throughput of a peer with zero endorsement requests. Hence, for a peer, we did not submit any endorsement requests while using others peers to load balance the endorsement requests. We observed that the non-endorsing peer achieved a peak throughput of 3000 tps, which was the maximum achievable throughput by validation and commit phases. Figure 5(b) plots the time taken by the validation and commit phases over a different number of peers. With an increase in the number of peers, the validation and commit time reduced from 23 *ms* to 19 *ms* and 21 *ms* to 15 *ms*, respectively. This is because of the reduction in the endorsement load per peer, which resulted in a tiny reduction in contention on the CPU and IO.

Takeaway 2. *The horizontal scaling of a Fabric network by adding more peers can help in reducing the load on the endorser, i.e., simulation phase, but it does not help the validation and commit phases due to redundant work. The redundant work is needed across organizations as one organization does not trust another but the same does not applies for peers within an organization. It is necessary to avoid redundant work between peers within an organization to improve the performance.*

(3) Scaling By Channels. Figure 4(c) plots the throughput achieved and CPU utilization over a different number of channels. With an increase in the number of channels from 1 to 8, the throughput increased from 1790 tps to 3034 tps (an increment of 1.69×). This is because each channel is independent of others and maintains its chain of blocks. Hence, the validation and commit phases of multiple blocks (one per channel) executed in parallel, which increased the CPU utilization that resulted in higher throughput. Figure 5(c) plots the time taken by the validation and commit phases over a different number of channels. With an increase in the number of channels per peer, the time taken by both validation and commit phases increased from 23 *ms* to 85 *ms* and 21 *ms* to 106 *ms*, respectively, due to increased contention on the CPU and IO with the parallel processing of blocks.

Takeaway 3. *Though adding more channels increased the throughput, it leads to limited data access. A smart-contract hosted on a channel can invoke a smart-contract hosted on another channel within the same peer for ready only operation but not for write operations. Further, the serializability validation would not be done across channel which might not satisfy serializable property for a transaction that touches multiple smart-contract across channels. Hence, the channel may not be a suitable technique for scaling a Fabric network.*

(4) Scaling By Channels and Peers. Figure 4(d) plots the throughput achieved and CPU utilization over a different number of channels when the number of peers per organization was 4 (refer to Figure 4(c) for the case of 1 peer per organization). Compared to scaling by only channels, this approach increased the throughput significantly from 1790 tps to 9668 tps (an increment of 5.4×). This is because when a single peer per organization hosted all channels, both the CPU & IO contention on the peer were high due to the parallel processing of multiple blocks (one per channel). With an increase in the number of peers per organization, the per peer channel count decreased which reduced both the CPU & IO contention on peers that resulted in higher throughput.

Figure 5(c) plots the validation and commit time. With an increase in the number of channels from 1 to 4, the validation and commit time increased from 19 *ms* to 27 *ms* and 15 *ms* to 19 *ms*, respectively. This is because of the increment in the endorsement load per peer. Due to the uniform distribution of channels among the peers, with an increase in the number of channels, the number of peers handling a channel reduced,

which increased the endorsement load per peer. When the number of channels was 8, the validation and commit time increased to 33 *msecs* and 36 *msecs*, respectively. This is because each peer handled two channels, which resulted in parallel validation and commits of two blocks.

Takeaway 4. *The scaling of a Fabric network by adding more number of peers and channels increased the throughput significantly. However, a smart-contract hosted on a channel cannot invoke the smart-contract hosted on another channel on another peer (even within the same organization) for both read and write operations which results in an absolute data silos. The channels are preferred only when there is a need for the complete user, transaction, and data privacy. Hence, scaling by both peers and channels may not be a suitable technique to achieve a higher throughput in a Fabric network.*

(5) Scaling by Endorsement Policy. Figure 7(a) and 7(b) plot the throughput achieved and CPU utilization over a different number of organizations in a channel when the endorsement policy was `simple` and `complex`, respectively. Note that each organization ran a peer. Compared to the `simple` endorsement policy, the `complex` policy resulted in a lower throughput as the number of signatures verification increased. With an increase in the number of organizations with `simple` policy, the endorsement requests were load-balanced among more peers which increased the throughput and decreased the CPU utilization. However, with an increase in the number of organizations with `complex` policy, the throughput decreased while CPU utilization increased. This is because the validator had to verify an additional number of signatures with an increased number of organizations.

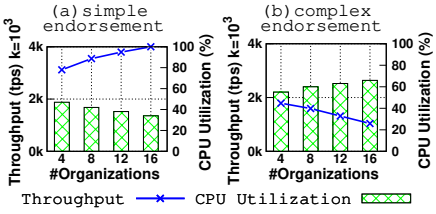

Figure 7: Impact of policies.

Takeaway 5. *The performance can be improved significantly using a simple endorsement policy. However, it would result in a weaker trust model. Further, the endorsement policy is more dependent on the application scenario and the required trust model. In the rest of this paper, we only consider the complex endorsement policy as it encapsulates all trust models.*

2.3 Dynamic Scaling

The scaling approaches are mainly used to dynamically scale up or down a distributed system to manage the load arrival rate and reduce the operation cost by avoiding overprovisioning. Hence, we measure the time taken to scale up a network by adding more vCPUs and peers to existing organizations.

Dynamic scaling by adding more vCPUs. To study the impact of dynamic vCPU scaling on the performance, we overloaded peers in a network by generating more load than it can handle at 4 vCPUs. We then increased the number of vCPUs to 16, one peer at a time, after the length of the *block queue* reached 200. Figure 6(a) plots the time taken to reduce the block queue length from 200 to 0 for each peer across organizations. Though peers scaled immediately, it took around 50 *secs* to 70 *secs* to reduce the queue length to 0. Figure 6(b) plots the ledger block height over time, while the number of vCPUs is increased from 4 to 8. The block height is nothing but the last committed block number. As expected, the ledger commit rate increased. Though the queue size became 0 within 70 *secs* and the ledger commit rate increased, there was a significant impact on the number of failed transactions due to serializability violations (as shown in Figure 8). As more blocks were waiting in the queue to update the ledger state, new transactions were endorsed using very stale data. As a result, many transactions in each block got invalidated later by the *serializability validator* due to mistmatch in the version of states present in the read set againsts the committed ledger state. After scaling the peers, it took 180 seconds to reduce the number of failed transactions to less than 2.

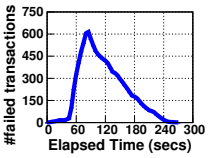

Figure 8: Failed transactions due to an overloaded situation.

Dynamic scaling by adding more peers. To study the time taken to add a new peer in an existing organization, we ran a peer per organization and then added a new peer at 2^{nd} minute, 5^{th} minute, and 10^{th} minute. Figure 6(c) plots the time taken by new peers to sync up with the existing peers such that they can process endorsement requests and new blocks. We observed that the time taken was proportional to the block height of existing peers as the new peer had to fetch all old blocks, validate, and commit them one by one. As there were no endorsement requests on the new peer during the sync up, it was able to catch up with the existing peer by committing transactions at a rate of 3000 tps (while with endorsement, it was only 1790 tps). However, it could take hours when the block height of existing peer is too high.

Takeaway 6. *The dynamic scaling by vCPU is efficient as it can quickly react to the increased load. However, this approach is limited by the number of available vCPUs in a server. With the dynamic scaling by peers, the new peer took a significant amount of time to sync up with existing peers to become available for handling the load. This is because the new peer validated and committed all blocks, i.e., redundant work. Hence, we need a better approach to scale a network.*

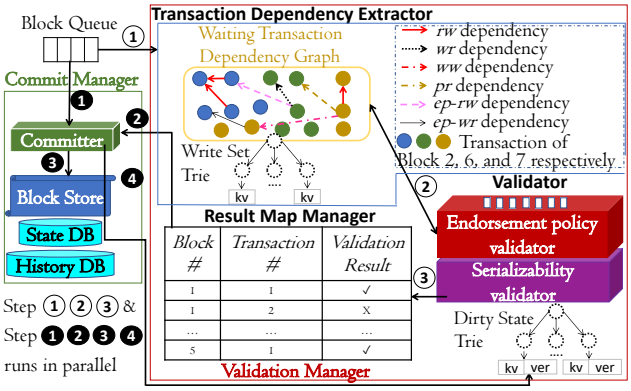

Figure 9: Proposed design to enable pipelined execution of validation and commit phases.

2.4 Problem Statement.

Re-architect Hyperledger Fabric to (1) increase the CPU utilization per peer and (2) reduce the redundant work within an organization such that the overall throughput achieved in a network is improved without using channels and `simple` endorsement policy. As the load might change over time, build a framework that can help to scale up or down the network quickly to reduce the overall operational cost.

3 DESIGN OF A SCALABLE FABRIC

First, in §3.1, we present a new block processing architecture that enables the pipelined execution of validation and commit phases and parallel validation of transactions, which belong to different blocks, to increase the CPU utilization and throughput without violating the serializability isolation. Next, in §3.2, we introduce a new peer type called *sparse peer* that can avoid redundant work within an organization.

3.1 Pipelined Execution of Phases

As mentioned in takeaway 1 in §2.2, we propose a new architecture (refer to Figure 9) that enables the pipelined execution of validation and commit phases while still ensuring the correctness, i.e., serializability isolation. Besides, it enables the *serializability validator* to validate transactions in parallel (even across blocks) as opposed to serial validation in vanilla Fabric. The two key ideas of the proposed architecture are (1) to exploit the *read-write* set and the *range query info* present in the block for each transaction. Using these two information, we can construct a dependency graph which can be used to validate transactions parallelly without violating the serializability isolation; (2) to maintain a dirty state in memory to keep the valid transaction's writes which are not yet committed to the state DB. This dirty state helps to avoid reading a stale state while pipelining the execution of both phases. Next, we describe the steps involved in each phase.

3.1.1 *VALIDATION PHASE.*

The proposed validation phase consists of three components: (1) transaction dependency extractor, (2) validator, and (3) a result map manager, as shown in Figure 9. In step ①, the extractor reads a block from the block queue and updates the dependency graph. In step ②, each free validator worker performs the following three operations: first, it retrieves a transaction from the graph that has no out-edges & validates it; second, if the validation passes, it applies the write-set to the dirty state; third, it updates the dependency graph. In step ③, it adds the validation result to the *result map* before moving to step ②. Next, we describe these steps in details.

(1) Transaction dependency extractor. The dependency extractor adds each transaction in a block to the *waiting-transactions dependency graph*. This graph contains a node per transaction with an identifier of form {*block number, transaction number within the block*}. We interchangeably use node and transaction. Let's assume T_i and T_j are two transactions, and T_i appeared in a block before T_j (where T_j can be in the same block or any next block). An edge from T_j to T_i denotes that the transaction T_i must be validated before validating T_j. The following are the six types of dependencies that can create an edge from T_j to T_i: (1) ***read-write*** (rw) dependency—T_i writes a version of some state, and T_j reads the previous version of that state; (2) ***write-read*** (wr) dependency—T_j writes a version of some state, and T_i reads the previous version of that state; (3) ***write-write*** (ww) dependency—Both T_i and T_j write the same state; (4) ***phantom-read*** (pr) dependency—T_j performed a range query, and T_i writes a new state that would match the range query performed by T_j; (5) ***endorsement-policy-read-write*** (ep-rw) dependency—T_i updates the endorsement policy of a smart-contract, and T_j invoked that smart-contract using the previous version of the endorsement policy; (6) ***endorsement-policy-write-read*** (ep-wr) dependency—T_j updates the endorsement policy of a smart-contract, and T_i invoked that smart-contract using the previous version of the endorsement policy; No cycle is possible in the graph as an edge is always from a new transaction to an old one. Table 2 presents the difference between our proposed *waiting-transaction dependency graph* and dependency graphs used in prior arts [19, 32, 34] related to a permissioned blockchain platform. It is evident that our proposed graph is novel in its own way due to the difference in goal.

Fate dependencies. While all six dependencies are used to decide between parallel and serial validation of transactions, only the *rw-dependency*, *pr-dependency*, and *ep-rw-dependency* also decide the validation results of dependent transactions—called *fate dependencies*. When there is a *fate dependency* from T_j to T_i and T_i is valid, the T_j would be marked invalid to ensure serializability. For any other dependencies, T_j can be validated irrespective of the validation results of T_i. The dependency extractor records all dependencies between every two transactions in the graph.

Table 2: Comparison of our dependency graph with dependency graphs used in prior arts in permissioned blockchain

	our approach	**ParBlockchain** [19]	**blockchain RDBMS** [32]	**Fabric++** [34]
goal	to validate transactions parallely (even across blocks)	to execute transactions parallely		to reorder transactions to reduce the abort rate
transaction flow	execute-order-validate	order-execute	order-execute & execute-order in parallel	execute-order-validate
dependencies tracked	rw, wr, ww, pr, ep-rw, ep-wr	rw, wr, ww		rw
dependency boundary	across blocks	single block		
cycles	impossible	possible		
constructed place	peer	orderer	peer	orderer
constructed time	during validation	during ordering	during execution	during ordering
ordering semantics	FIFO			order that reduces the transactions abort
serialization order	FIFO		partial FIFO	
supported queries	get, put, range	get, put	SQL	get, put

Detecting phantom dependency. Except for *pr-dependency*, all other dependencies are easy to identify using read-write set. To identify *pr-dependency*, we maintain a trie data structure with states which are present in the write set of all waiting transactions in the graph. Before adding a transaction to the graph, we run the range query, which is present in *range query info* of the transaction, on the trie data structure. If the query finds any matching state, then the new transaction has a *pr-dependency* to the transaction that holds the matching state in its write set.

The extractor exposes the following two operations to enable other components to access the dependency graph. (1) `GetNextTransaction()` T_i—Finds a list of transactions from the graph which has no out-edge; From the list, returns a transaction T_i that has the least {*block number, transaction number*} as the identifier. In other words, it returns the oldest transaction in the graph which has no dependency on other transactions (but other transaction can depend on it). (2) `UpdateDependencyGraphAndValidationResults(`T_i`, isValid)`—updates the dependency graph by removing the node T_i. If `isValid` is *true*, i.e., T_i is valid, it removes all transactions that have an out-edge to T_i due to a *fate dependency*, and adds them to the *result map* as invalid.

(2) Validator. The logic of *endorsement policy validator* and *serializability validator* remains the same except the following two modifications: (1) both validators use dirty state & state DB (to avoid reading a state data); (2) the *serializability validator* uses multiple workers to validate transactions parallelly instead of a single worker & serial OCC execution.

Each free worker calls `GetNextTransaction()` to get the next transaction to be processed. First, the *endorsement policy validator* checks whether the policy is satisfied. On a success, the same worker executes the *serializability* check. If the transaction passes both validations, the worker applies the write-set to the dirty state and calls `UpdateDependencyGraphAndValidationResults()` to update the dependency graph. Finally, the worker adds the validation result to the *result map*. To ensure that the validator does not read stale state, a read request (such as a read of the policy of a smart-contract or the version of a state) would first go to the dirty state. Only on a miss, the read request would reach the state DB. For storing the dirty state, we use a trie data structure with leaf as a {key-value pair (i.e., a state), version}. This trie structure enables the *serializability validator* to validate range queries present in *range query info* for a phantom reads. Every range query is executed on both the trie and state DB to detect a phantom read.

(3) Result map manager. It manages a map of {block number, transaction number} to validation result. It exposes the following two operations on the map: (1) `AddValidationResult` (*block number*, *transaction number*, *isValid*) adds the validation result of a transaction to map; (2) `GetAndDeleteValidationResult` (*block number*, *transaction number*) returns the validation result associated with a given transaction after deleting it. Both *extractor* and *validator* call (1) whereas only the committer calls (2).

3.1.2 *COMMIT PHASE.*

As compared to the validation phase, the logic of the commit phase does not change significantly. As shown in Figure 9, in step ❶, whenever the *committer* becomes free, it reads a block from the queue and retrieves the list of transactions. In step ❷, the committer fetches the validation results by calling `GetAndDeleteValidationResult()` for each transaction. If the validation result is not available for a transaction, the call would be blocked until the result is available. Once the committer collects validation result of all transactions, in step ❸, it stores the block in the block store and applies the valid write-sets to state & history DB as in vanilla Fabric. In step ❹, it calls the validation manager to remove the dirty state associated with the just committed block number as the validator can read those states from the state DB itself.

3.2 Sparse Peer to Avoid Redundant Work

From the takeaway 2 in §2.2, we know that multiple peers within an organization do redundant work which limits the

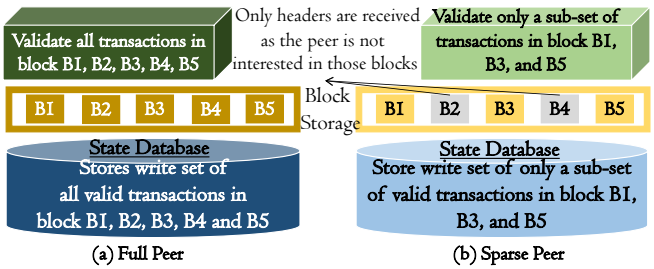

Figure 10: Difference between a full peer and a sparse peer.

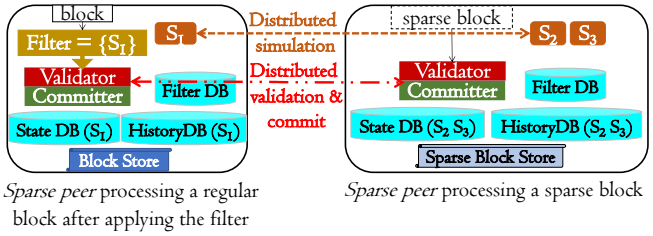

Figure 11: Two variants of sparse peer.

efficiency of horizontal scaling. To avoid redundancy, in Figure 10, we propose a new peer type called a *sparse peer*. As compared to a full peer in vanilla Fabric, a *sparse peer* may not validate and commit all transactions within a block. The concept of a *sparse peer* is inspired by the sharding concept in a distributed database. First, in §3.2.1, we present our design of a sparse peer. Second, in §3.2.2, we present a method to execute the simulation phase in a distributed manner, since a sparse peer does not hold all states of a blockchain. Finally, in §3.2.3, we present distributed validation and commit of transactions, which is needed as a result of distributed simulation.

3.2.1 *SPARSENESS IN VALIDATION AND COMMIT.*

The key idea behind a *sparse peer* is that it can selectively validate and commit transactions. If all *sparse peers* within an organization select a non-overlapping set of transactions, we can avoid the redundant work. Towards achieving this, first, we define a deterministic selection logic such that each *sparse peer* selects a different set of transactions. Second, we make appropriate changes at the validator and committer to apply the selection logic on the received block. Third, as an optional feature, we make the peer pass the selection logic to the orderer such that the orderer itself can apply the filter and send only required transactions in a *sparse block*. As a result, both network bandwidth utilization and disk IO would reduce. Figure 11 shows two variants of *sparse peer*.

(1) Transaction selection filter: Each sparse peer owns a *filter* and applies it on a received block to identify which transactions to consider. The *filter* is simply a list of smart-contracts. For each peer, the admin assigns/updates the *filter* by issuing a request via gRPC to the peer process. The *sparse peer* only validates and commits transaction in a block that invoked a smart-contract specified in the *filter*. When a transaction had invoked multiple smart-contracts, even if the filter contains only one of those smart-contracts, the transaction would be considered by the *sparse peer*. The reason for chosing smart-contracts in a filter is that most application such as TradeLens [14], WeTrade [17], Food Trust [7] are built using multiple smart-contracts each performing a specific task as similar to micro-services. Such an architecture enables each team of developers to build and manage a smart-contract independently. Further, the application uses the feature of one smart-contract invoking another when a transaction needed to touch multiple smart-contracts (which was of lower percentage among the total number of transactions submitted). Note that each contract maintains its states.

(2) Validation and commit based on a filter. When the transaction dependency extractor reads a block from the queue, it only adds transactions, which invoked a smart-contract present in the *filter*, to the waiting-transactions dependency graph. When the committer reads the block, it marks transactions, which did not invoke any smart-contract present in the *filter*, as not validated but stores the whole block in the block store. The rest of the validator and committer logic remains the same. The advantage of storing the whole block is that when the filter is updated, the sparse peer can fetch all transactions associated with the newly added smart-contract from the local block store itself. Then, it needs to fetch only the validation results from other peer's block store to avoid revalidation. However, the disadvantage is that it would not reduce network bandwidth utilization and disk IO. As an optional feature, next, we allow sparse peers to pass its filter directly to the orderer.

(3) Block dissemination based on filters. If orderers themselves apply the filter and send only appropriate transactions via a *sparse block* to each *sparse peer*, we can save both network bandwidth utilization and disk IO. Hence, each *sparse peer* sends its filter to an orderer to which it has connected. For each block, the orderer applies the filter and send only the required transactions to the peer. However, this creates a problem with the hash chain and its verification. In vanilla Fabric, at the time of block creation, the orderer computes a *block hash* and stores it in the block. The *block hash* is computed using all transactions' bytes within that block and the hash present in the previous block. When a peer receives a block, it can check its integrity by verifying the hash chain. Further, this hash chain is the source of truth of a blockchain network. If we make the orderer to send only a sub-set of transactions in a block, the peer would not be able to verify and maintain the hash chain integrity.

Sparse block. To fix this problem, we propose a *sparse block* which includes (1) a Merkle tree to represent the *block hash* (as shown in Figure 12); (2) only a sub-set of transactions after applying the filter; (3) applied filter; (4) transactions

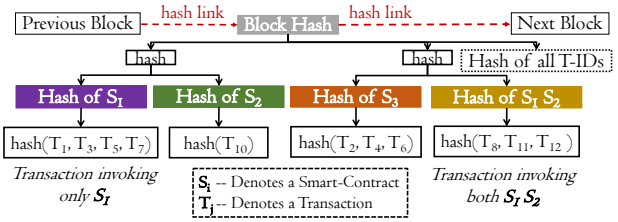

Figure 12: Merkle tree based block hash computation.

identifiers (T-IDs) of all transactions. In our Raft based consensus service, the leader node constructs the merkle tree while other nodes applies filter before sending the block to its connected peers. When a *sparse peer* receives a *sparse block*, it can verify the hash chain integrity only by using a sub-set of transactions. Due to the verification of Merkle tree hash, as compared to a single hash verification, this approach would increase the CPU utilization but reduces network and disk IO. A list of transaction identifiers are sent with a *sparse block* to enable the validator to check for duplicates and mark them invalid appropriately. In vanilla Fabric, orderer does not peek into the transaction. However, we break that as the orderer needs to find transactions associated with each smart-contract and also find transactions which invoke multiple smart-contracts. Since the orderers already have access to the entire transaction even in a vanilla Fabric, a motivated party could always peak into the transactions. Hence, our approach does not weakens the trust model. Moreover, a *sparse block* is an optimization, not a necessity.

3.2.2 DISTRIBUTED SIMULATION.

During a simulation phase, when a smart-contract invokes another, the peer expects both contracts instances to reside on it. With an introduction of *sparse peers*, the peer expectation would fail if filters are not designed carefully by considering the dependencies between smart-contracts. Assuming that administrators define filters carefully taking dependencies into account, we would lose the benefits of *sparse peers* in the following two cases. (1) when the number of transactions involving multiple smart-contracts is relatively low; (2) when a smart contract invokes all other smart contracts, but not all of them together. Therefore it is inevitable to get rid of this expectation and use a distributed simulation.

In vanilla Fabric, when a smart-contract named *supplier* invokes another contract named *carrier*, the *supplier* contract sends a message to the peer process to invoke the *carrier* contract. The peer process forwards the request to *carrier* contract and returns its results to *supplier* contract. Note that states read/written by *carrier* contract are also added to the transaction's read-write set. There is no fundamental reason for both smart-contracts to reside on the same peer. Hence, with *sparse peer*, we allow these smart-contracts to be placed on different peers and enable distributed simulation to allow *supplier* contract hosted on sparse peer P_1 to invoke the *carrier* contract hosted on P_2. We achieve this by making P_1 send a message to P_2 which invokes the *carrier* contract. The response, which includes the states that are read/written by the *carrier* contract, is relayed back to P_1. The *filter DB*, as shown in Figure 11, holds the filters of each *sparse peer* and is used to decide which peer to contact for a given smart-contract. As the distributed simulation happens over a network and the *endorser* holds a read lock on the whole state DB [35], the commit operation would get delayed if there are substantial number of distributed simulations. Hence, we adopt the technique proposed by Meir et al. [30] to get rid of the read-write lock on the state DB.

3.2.3 DISTRIUBTED VALIDATION AND COMMIT.

Due to the distributed simulation, now we need distributed validation and commit. Consider a transaction T invoking smart-contracts S_1, S_2, $\ldots S_n$. Consider sparse peers P_1, P_2, $\ldots$, P_n where each peer P_i has filter F_i with a single smart-contract S_i. The transaction invoking all n smart-contracts is considered valid when the transaction satisfies both the policy and serialization checks of each contract invocation. Hence, to commit this transaction, we would require an agreement from all n *sparse peers* during the validation phase.

Strawman protocol. Each peer P_i validates parts of the transaction that involved smart contracts $S_i \epsilon F_i$, and then broadcast the results to every other peer. Once a peer has recevied *valid* as a result for all smart-contracts $S_1 \ldots S_k$, it can consider the transaction valid and commit it. If an *invalid* result is received, the peer does not need to wait for any more results, and can proceed by invalidating the transaction. As the peers within an organization are trusted, there are no security issues. While the approach is simple, there is a significant drawback. Since the peers could be at different block height due to different block commit rates (as a result of heterogenous hardware, or heterogenous workloads), a transaction requiring distributed commit could block other transactions in the block for a considerable amount of time. As a result, the *committer* could get blocked at `GetAndDeleteValidationResult()` call.

Improved protocol using priorities. To reduce the committer's blocking time, we prioritize the validation of distributed transactions by enhancing the *waiting-transactions dependency graph* introduced in §3.1 as follows: each node in the graph has two flags—`committerBlocked` & `priority`. The `committerBlocked` flag is marked when the *committer* call to `GetAndDeleteValidationResult()` is blocked. The `priority` flag is marked when the transaction requires distributed validation. Further, we enhance the logic of `GetNextTransaction()` described in §3.1 as explained next. The `GetNextTransaction()` first executes the following three steps in the order until it finds a list of nodes: (1) find a list of nodes where each node is marked with `committerBlocked`

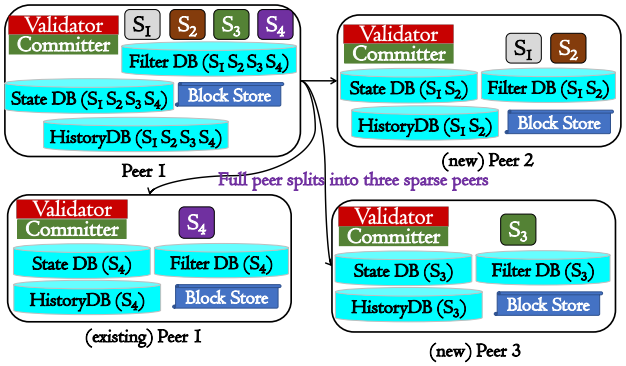

Figure 13: A full peer is splitting into 3 sparse peers.

and has no out-edge; return the list if non-empty. (2) find a list of nodes where each node is marked with *priority* and has no out-edge; return the list if non-empty. (3) find a list of nodes which have no out-edge; return the list if non-empty. From the list of nodes, the `GetNextTransaction()` return the transaction T_i associated with a node that has the least {*block number, transaction number*} as the identifier.

Deferred Transactions. Even with the improved protocol, the committer might get blocked due to network delays. Hence, we introduce deferred transactions in which the *committer* can proceed to commit all local transactions without waiting for the distributed transactions instead marking them as deferred during the commit (as it is needed for recovery after a peer failure). Whenever the result is available, the deferred transaction is committed and removed from the graph. Note that even a local transaction that has a dependency on a deferred transaction must be deferred.

4 AN AUTO-SCALING FRAMEWORK

From takeaway 6 in §2.2, we know that dynamically scaling a Fabric network by adding more peers took significant time. Hence, in this section, we propose a method to quickly scale a network using *sparse peer*, i.e., by splitting a full peer into multiple *sparse peers* or a *sparse peer* into multiple *sparse peers*. Further, we also propose a method to scale down a network by merging multiple *sparse peers* into a smaller set of *sparse peer* or a full peer (as described in §4.2).

4.1 Splitting a full peer into sparse peers

To split a full peer into multiple sparse peers, we make the full peer in the vanilla Fabric also to use the filter. A full peer contains all deployed smart-contracts (denoted by a set S) in its filter as opposed to only a subset of smart-contracts. The process of splitting a full peer involves three steps: (1) add a new sparse peer with a filter containing a sub-set of smart-contracts S_A; (2) copy the blockchain states of smart-contracts in set S_A, to the new sparse peer from the full peer; (3) remove smart-contracts in set S_A from the filter in full peer to make it a sparse peer. While step 1 and 3 are trivial, step 2 is complex. Figure 13 shows an example of such a split.

Copy of a smart-contract's states. To a newly added sparse peer, we need to copy data, which are associated with smart-contracts in set S_A, from the block store, state DB, and history DB in the full peer. In vanilla Fabric, each smart-contract's state is prefixed with a smart-contract name. Hence, it is trivial to identify the states associated with a given smart-contract. We can copy the states in two approaches: (1) copy blocks from the block store along with the validation results to the new sparse peer to build all three stores; (2) copy the required states directly from the state DB. As the former approach is costlier in terms of both network and disk IO, we go with the later. Once the state DB is copied along with the last committed block, we can start to allow endorsement and regular block commit on the new sparse peer. Note that the last committed block is required such that the new peer can validate the hash chain when a new block is received. In addition to this, we also need to copy all transactions' identifiers from an index DB maintained at the block store. These transactions' identifiers are used by the validation manager to detect duplicate transaction identifier.

Interference from on-going block commits. To copy the state DB data, first, the newly added sparse peer uses the *gossip* component to identify the last committed block number in the full peer. Second, it asks the full peer to send all active states as of a given block number. Third, it receives the data and builds the state DB. At the full peer, it is not trivial to send all active states as of a given block number as the full peer continues to commit blocks and changes the active states, which interfer with state transfer. For example, in Figure 14, the new sparse peer is requesting all states associated with the smart-contract S_1 as of block number 3. In other words, it asks for the active value of keys *K1* to *K5* as of block number 3. However, the existing peer continues to commit blocks and changes the active states. At block number 5, the key *K1* does not even exist in the state DB.

Interference Mitigation. To tackle the above mentioned interference problem, the following three steps are executed:

(1) The full peer adds new entries in history DB of form {*smart-contract, block number, transaction number*} $\mapsto$ a list of {*key, isDelete, isDeferred*} for every block commit. Note that the `isDelete` and `isDeferred` are used to denote a deleted key and a deferred transaction, respectively.
(2) When the full peer receives a copy request with a block number, it performs a range query over the newly added index with the start key as {*smart-contract, 0*} and the end key as {*smart-contract, the requested block number*} to find all needed keys. The full peer can then read the values and versions from the state DB. However, the state DB's copy either might be at a version that falls outside the requested block range due to new blocks' commit or

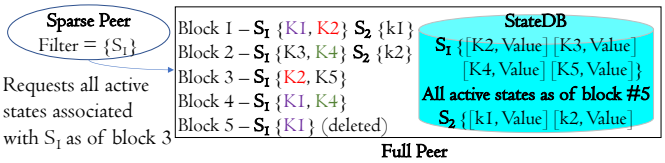

Figure 14: A sparse peer issues a copy request to a full peer.

might not even exist due to a delete operation. In such a case, the full peer would fetch the respective value and version from the write-set present in the block store.

(3) Once the newly added sparse peer copies all needed data including the data associated with the deferred transactions, the full peer updates its filter to become a sparse peer. The newly added sparse peer starts to receive blocks (including the one that got committed in the network during the data copy phase) from the orderer and proceed with the regular block validation and commit.

Thus, a full peer got split into two sparse peers. The approach is same for splitting a peer into more than two sparse peers.

Parallel copy of a smart-contract's states. When either there are two full peers within an organization or two replicas of a sparse peer to load balance the endorsement requests, we copy the required data to a new sparse peer parallelly. The following steps describes the flow:

(1) Once the new sparse peer gets the last committed block number of other peers, it creates multiple requests with non-overlapping block ranges and send each request to a different peer while considering its last block number.

(2) When a peer receives a copy request with a range of blocks denoted by {*start block number*, *end block number*}, it performs a range query over the newly added index in the history DB with the start key as {*smart-contract, start block number*} and the end key as {*smart-contract, end block number*} to find all needed keys. The peer can then read the values and versions from the stateDB. It might happen that the stateDB's copy might be at a version that falls outside the requested block range. In that case, the full peer marks the key with a sentinel value indicating the same. We call this entry as a **`hole`**.

(3) If the new sparse peer receives a `hole` in a response, it waits for responses of other requests sent to other peers as they might fill the hole automatically.

(4) Once all responses have been received, there might still be a few `holes`. The new peer then requests for the value of the key as of the last committed block number. The other peer retrieves the value from the transaction's write-set present in the block store.

Further to enable a new peer to fetch states from other newly added peers which do not have the full block store, we always copy the complete history DB. As a result, every peer can be used to copy states to a new peer.

4.2 Merging sparse peers into a full peer

To scale down a network, we need to merge multiple sparse peers into a few sparse peers or to a full peer. The logic of merge operation is almost same as splitting a full peer into multiple sparse peers, in terms of changing the filter and copying the data. In the split operation, we remove smart-contracts from the filter in existing peers, whereas in the merge operation, we add smart-contracts to filter in a few of existing peers while emptying the filter of other peers. The peer that has the largest volume of the data are retained while the peer with least volume of data are removed.

5 IMPLEMENTATION

The overall implementation added 15k lines of golang code to Fabric v1.4 excluding test and generated code. The dependency graph was implemented using two queues: one for distributed transactions and another for a normal transactions. Each entry in the queue pointed to transactions blocking it. These pointers were used to track dependencies. The dirty state component was implemented using a trie in order to handle validation of range queries. The implementation of sparse peer itself was straightforward as it mostly involved making the code ignore transactions outside of the a peer's filter. For modifying filters of a peer on the fly by the administrators, we implemented a system smart-contract that gave an interface to modifying the peer's filters. For splitting and merging of peers, we added a controller between the peers that handled requesting and sending "diffs" for stateDB and historyDB in a given *start block number* and *end block number* For responding to such requests, a controller would acquire a snapshot on the underlying levelDB instances.

6 EVALUATION OF PROPOSED DESIGN

In this section, we evaluate the performance of our proposed design against vanilla Hyperledger Fabric. The setup used for the study is same as the one shown in Figure 3. For all experiments, peers were assigned with 16 vCPUs and only one channel was created in a network. Unless specified otherwise, we use one peer per organization, and eight smart-contracts each hosting the smallbank workload.

Pipelined Execution of Validation and Commit Phases. Figure 15(a) plots the throughput achieved with the pipelined execution against vanilla Fabric. As expected, the throughput increased by 1.36× while increasing the CPU utilization from 50% to 70%. The validation manager was so efficient that the committer never got blocked. The size of the *result-map* was always greater than 500. This is because the time taken by the committer (≈31 ms at 2500 endorsement requests per second—eps) was always higher than the time taken by validators. Further, the end-to-end commit

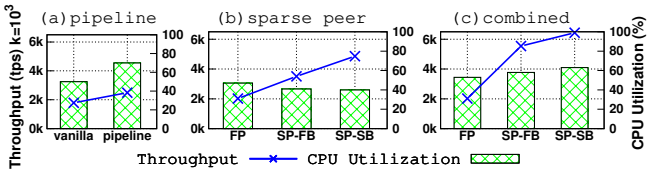

Figure 15: Performance comparison of (a) pipelined execution of validation and commit phases, (b) sparse peer (SP) with full blocks (FB) and sparse blocks (SB), (c) combination of (a) and (b) against vanilla full peer (FP).

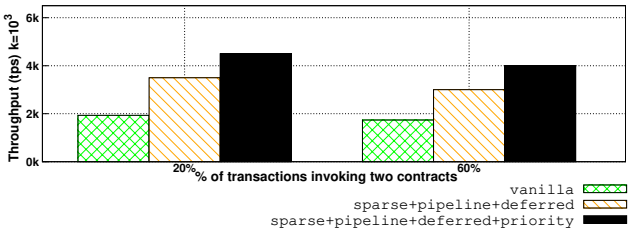

Figure 16: Performance with distributed transactions.

latency (block verification + validation + commit) for a block reduced from 50 *ms* (at 1800 eps) to 38ms (at 2500 eps).

Sparse Peer with Full and Sparse Blocks. We evaluate the performance of two variants of sparse peer proposed in §3.2. Each organization hosted 4 sparse peers where the filter of each sparse peer contained only 2 non-overlapping smart-contracts. Figure 15(b) plots the throughput achieved with both variants of sparse peers against a network where each organization hosted 4 vanilla peers. As expected, the throughput increased significantly by 2.4× with sparse peers. At the same time, the CPU utilization per node reduced as compared to the vanilla full peer because of the avoidance of redundant CPU intensive tasks. This suggests that we can use smaller servers for higher throughput with sparse peer. Compared to the sparse peer processing full blocks, the sparse peer processing sparse block achieved higher throughput due to the reduced IO operation, and minor improvement accounted by reduction in block deserialization.

Sparse Peer with Pipelined Execution. Figure 15(c) plots the throughput achieved with the combination of sparse peer and pipelined execution against vanilla Fabric. The throughput increased significantly to 6400 tps, i.e., by 3.16×. Further, the CPU utilization also increased due to the pipelined execution of validation and commit phases.

Distributed Simulation and Validation. When transactions invoke multiple smart-contracts, sparse peer employs distributed simulation and validation. To study the performance of our proposed system in the presence of distributed transactions, we submitted transactions that invoked multiple smallbank contracts. Figure 16 plots the throughput with vanilla Fabric, sparse peer with pipeline and deferred transactions, and sparse peer with pipeline, deferred transactions, and priority. When 60% of transactions invoked multiple

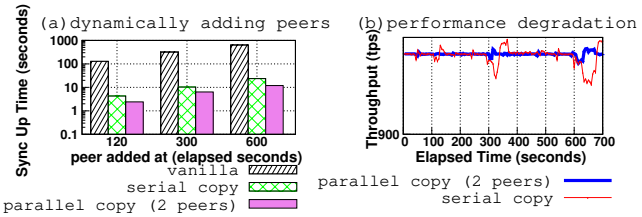

Figure 17: Time taken to a add a new peer (split).

contracts, the performance achieved was lower as excepted compared to the run where only 20% of transactions invoked multiple contracts, irrespective of the employed optimization. This is because, with 60% transactions invoking multiple contracts, the amount of work to be done at each peer increased. Futher, our proposed techniques resulted in a significant performance improvement as compared to vanilla Fabric.

Scale Up. We evaluate the dynamic scaling approch discussed in §4. Figure 17(a) plots the time taken to add a new peer by copying states from a single other peer as well as from two other peers. To perform a fair comparison with vanilla, we added a new sparse peer with all smart-contracts (which is an equavalent of a full peer) and generated the same load on existing peers. Compared to the time taken to add a vannila peer at various minutes, our approach provided a multifold reduction. This is because, our approach copied a much smaller amount of data as compared to vanilla approach. In general, the size of block store is 60× higher than the size of state DB. As we directly copy required states from the state DB, the time taken to add a new peer reduced significantly. Further, when the new peer pulled states from two peers parallelly, the time taken reduced by half (the time in Figure 17(a) is represented in log-scale). Figure 17(b) plots the impact of state transfer on the performance of existing peers. Whenever a new peer is added, existing peers observed a throughput drop due to additional disk IO contention and state DB synchronization. When the new peer pulled data from multiple sources, the performance degradation reduced significantly. This shows that our approach can help in scaling a Fabric network very quickly.

Scale Down. When peers are under-utilized, it is necessary to scale down to reduce the operational cost. In vanilla Fabric network, the scale down operation is very easy as we need to just stop a few peers within an organization. However, with sparse peers, we need to merge peers by copying one sparse peer's states to another sparse peer. To measure the time taken for merging multiple sparse peers, we ran an experiment where each organization hosted four sparse peers. Each sparse peer had two smart-contracts in its filter and they were non-overlapping with other sparse peers within an organization. We merged all sparse peers into a single full peer after running the network at 1800 tps for sometime. The time taken to merge these sparse peers after 2 minutes, 5 minutes, and 10 minutes were 4.29 *secs*, 10.17

secs, 24.59 *secs*, respectively. While the sparse peer helped to scale up a network quickly, it increased the time taken to scale down a network as compared to vanilla Fabric.

7 RELATED WORK

In this section, we cover the existing work that improved the performance of Hyperledger Fabric. None of them have studied the performance using different scaling techniques.

Thakkar *et al.* [35] conducted a comprehensive performance study and found bottlenecks in Fabric v1.0 and provided guidelines to design applications and operate the network to attain a higher throughput. Further they implemented a few optimizations on the peer process. These optimizations have already been included in Fabric v1.4, and hence our work builds upon this work.

Dang *et al.* [23] applied sharding at the consensus layer in order to scale a blockchain network. Our work is orthogonal to theirs as we optimize the block validation and commit phases. Further, we identified that the performance of a peer is a bottleneck in Fabric, not the ordering service.

Sharma *et al.* [34] studied the fundamental differences and similarities between blockchain systems and traditional databases. They then used ideas from the database literature, namely transaction reordering during the ordering phase and early transaction abort during the simulation phase. They used these techniques to reduce the failure rate of the transactions due to serialization conflicts. These techniques are orthogonal to our work as we focus on pipelined execution of different phases and to avoid redundant work.

Istvan *et al.* [27] proposed to shift from the block processing paradigm to a stream processing paradigm and only used blocks as a means to amortize disk access costs. This helped in decreasing the latency. This work is partially orthogonal to ours in that the benefits of sparseness and pipelined execution can be transfered to their design trivially.

Gorenflo *et al.* [26] proposed various optimizations such as replacing the state DB with a hash table, storing blocks in a separate server, separating the committer and endorser into different servers, parallelly validating the transactions headers, and caching the unmarshed blocks to reach a throughput of 20000 tps. However, we believe that many of these optimizations are not practical. For e.g., a state DB is must to support range queries and persist all active states (which would help to recover a peer quickly after a failure). In their measurement, there is no disk IO which is not suitable for a production setup and the workload had no read-write conflicts. Moreover, their work is orthogonal to ours as (1) they do not execute the validation and commit phases in a pipelined manner; (2) they not parallelly validate transaction during the serializability check; (3) they do not have a concept of sparseness in a peer; and (4) they do not provide a framework to scale up a Fabric network quickly. Note, we have adopted the block cache proposed in this work as mentioned in § 2.2.

Meir *et al.* [30] proposed to remove the read-write lock on the state DB and instead used an optimistic concurrency technique during simulation phase. We have adopted this technique to improve the performance achieved with the distributed simulation as described in § 3.2.2.

Goel *et al.* [25] argued that the default first-in-first-out ordering of transactions is unfair and inefficient. They proposed a weighted fair queuing strategy for ordering transactions which can support different quality of service for different transactions. Further, they implemented this in a decentralized manner. Our work is orthogonal to this work as we focus on the scalability of a Fabric network.

8 CONCLUSION

In this paper, we studied the performance of Hyperledger Fabric using various scaling techniques and identified two major bottlenecks: (1) serial execution of code in the critical path; (2) duplication of CPU and IO intensive tasks. Hence, we re-architected Fabric to remove these two bottlenecks. Toward this, we introduced a pipelined execution of important phases and a new peer type called *sparse peer*. Overall, the performance of Fabric improved by 3×. Fabric displays increased failed transaction rates when it is overloaded. As existing approaches took significant time to scale a network, we used *sparse peer* and provided a auto-scaling framework that can scale a network up/down very quickly.

REFERENCES

[1] Alipay press release. https://www.alibabagroup.com/en/news/press_pdf/p180201.pdf.
[2] Chain-m for airline industry. https://www.niit-tech.com/news-events/news/niit-technologies-introduces-chain-m-blockchain-powered-solution-airlines-and-its.
[3] Clsnet for bilateral payment netting. https://www.cls-group.com/products/processing/clsnet/.
[4] Electrum decentralized marketplace. https://electrumdark.co/.
[5] Everledger for mining industry. https://www.everledger.io/.
[6] grpc. https://grpc.io/.
[7] Ibm food trust for food industry. https://www.ibm.com/in-en/blockchain/solutions/food-trust.
[8] The linux foundation. https://www.linuxfoundation.org/.
[9] Open bazaar decentralized marketplace. https://openbazaar.org/.
[10] openidl for insurance industry. https://aaisonline.com/openidl.
[11] Origami network decentralized marketplace. https://ori.network/.
[12] particl decentralized marketplace. https://particl.io/.
[13] Securekey verified.me. https://securekey.com/.
[14] Tradelens for global trade. https://www.tradelens.com/.
[15] Understanding digital tokens: Market overviews and proposed guidelines for policymakers and practitioners by token alliance, chamber of digital commerce. https://morningconsult.com/wp-content/uploads/2018/07/token-alliance-whitepaper-web-final.pdf.
[16] Visa annual report. https://s1.q4cdn.com/050606653/files/doc_financials/annual/2018/visa-2018-annual-report-final.pdf.
[17] Wetrade for trade finance. https://we-trade.com/.

[18] M. Alomari, M. Cahill, A. Fekete, and U. Rohm. The cost of serializability on platforms that use snapshot isolation. In *2008 IEEE 24th International Conference on Data Engineering*, pages 576–585, April 2008.

[19] M. J. Amiri, D. Agrawal, and A. E. Abbadi. Parblockchain: Leveraging rransaction parallelism in permissioned blockchain systems. 2019.

[20] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De 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. Vukolić, S. W. Cocco, and J. Yellick. Hyperledger fabric: A distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pages 30:1–30:15, New York, NY, USA, 2018. ACM.

[21] V. Buterin. Ethereum: A next-generation smart contract and decentralized application platform, 2014. Accessed: July 31, 2019.

[22] Y. Chen. Blockchain tokens and the potential democratization of entrepreneurship and innovation. *Business Horizons*, 61(4):567 – 575, 2018.

[23] H. Dang, T. T. A. Dinh, D. Loghin, E.-C. Chang, Q. Lin, and B. C. Ooi. Towards scaling blockchain systems via sharding. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, pages 123–140, New York, NY, USA, 2019. ACM.

[24] 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.

[25] S. Goel, A. Singh, R. Garg, M. Verma, and P. Jayachandran. Resource fairness and prioritization of transactions in permissioned blockchain systems (industry track). In *Proceedings of the 19th International Middleware Conference Industry*, Middleware '18, pages 46–53, New York, NY, USA, 2018. ACM.

[26] C. Gorenflo, S. Lee, L. Golab, and S. Keshav. Fastfabric: Scaling hyperledger fabric to 20,000 transactions per second. In *2019 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pages 455–463, May 2019.

[27] Z. István, A. Sorniotti, and M. Vukolić. Streamchain: Do blockchains need blocks? In *Proceedings of the 2Nd Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers*, SERIAL'18, pages 1–6, New York, NY, USA, 2018. ACM.

[28] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, June 1981.

[29] Y. Manevich, A. Barger, and Y. Tock. Endorsement in hyperledger fabric via service discovery. *IBM Journal of Research and Development*, 63(2/3):2:1–2:9, March 2019.

[30] H. Meir, A. Barger, and Y. Manevich. Increasing concurrency in hyperledger fabric. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, SYSTOR '19, pages 179–179, New York, NY, USA, 2019. ACM.

[31] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system, Dec 2008. Accessed: July 31, 2019.

[32] S. Nathan, C. Govindarajan, A. Saraf, M. Sethi, and P. Jayachandran. Blockchain meets database: Design and implementation of a blockchain relational database. *Proc. VLDB Endow.*, 12(11):1539–1552, July 2019.

[33] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 305–320, Berkeley, CA, USA, 2014. USENIX Association.

[34] A. Sharma, F. M. Schuhknecht, D. Agrawal, and J. Dittrich. Blurring the lines between blockchains and database systems: The case of hyperledger fabric. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, pages 105–122, New York, NY, USA, 2019. ACM.

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