Storage Reduction of Hyperledger Fabric with Fault Tolerance Control on Shard Channels

Anonymous

***Abstract*—Blockchain has gained popularity in software industry for its transparency, security and privacy guarantees. However it has scalability issues in terms of latency and throughput which is addressed by literature to some extent with better consensus and communication protocols. But the inherent problem of high storage requirement for append only blocks and high computing needs are also the barriers to large scale adoption of blockchain in a variety of applications. This issue has currently drawn attention for advances in Blockchain. In this work, we propose a sharding configuration for popular Hyperledger Fabric (HLF) blockchain and show a novel storage reduction with an alternative way of reducing fault tolerance. We have demonstrated partial ledger based participation in HLF network along with description of HLF ecosystem. Sharding in HLF shows feasible advantages such as running HLF on a small scale, reducing cost of running and adding more adoption in various use cases (e.g. IoT networks). Here we present numeric results on how much storage can be saved for trading off fault tolerance (FT) of HLF in sharded architecture compared with usual architecture. With our implementation of basic shard generation we get more than 18% storage reduction compared to non-sharded configuration. We compare the impact of shard size, transaction size and transaction per block (TPB) on storage consumption and latency to confirm the practical usability of sharding and the feasibility of aforementioned advantages. Finally, we show storage rate comparison with state-of-the-art.**

I. Introduction

Blockchain is like linked list of block. New blocks are linked to previous one using special hashing mechanism based on information in block. Blocks are built from transactions where each transaction is considered immutable since the blocks and chain of blocks are immutable for distributed consensus mechanism. With large number of blocks the required storage in the network is rising. This means each participating node incurs increased storage. For example, the Bitcoin network [23] has grown to 435GB+ and for Ethereum [31] its 1400GB+ as of October 2022 and December 2023 respectively. The new nodes wanting to join in these networks need to ensure far more capacity keeping future growth in mind. No wonder this size is increasing day by day. As pointed by Global Blockchain Sync (GBS) [1], the sync time without their specialized solution for Ethereum archival node is up to 14 days and with it up to 3 days. So, it is going to rise more with faster growth rate each coming year. Ethereum is working on their solution to scalability problem with Shard Chains [6] which was expected to ship sometime in 2022 and later revised for an 2023 release of Danksharding instead of shard chain [2].

It is evident that, the high storage problem should be solved not only for public blockchain but also for private ones. For addressing the storage problem two broad types of techniques are applied in current literature. One involves forming a cluster of nodes and trying to process transactions within a single cluster and thus minimizing inter-cluster communications [13], [17]. Some of these techniques create data clusters along with node clusters. In [8], [19] such a technique is applied. But for high percentage of cross-shard transactions these type of solution does not work well. The other type of solutions approach the problem with the idea of dividing the data in the blockchain [12], [33]. However, most of the solutions focus on performance in terms of latency and throughput and the storage problem is often overlooked.

Data compression is the go to solution when it comes to reducing the size of data which ensures lesser storage need. But compression comes with the cost of computing time [28]. Blockchain is inherently slow incorporating data compression for storage reduction is not desirable. A widely used technique is to partition data into fragments/shards for reducing storage requirements in individual nodes. Systems like [26] and [16] have applied such ideas. The fundamental inspiration for utilizing distributed data sources rather than a central one has always been to increase throughput and reduce latency. Researchers have improved these aspects of distributed database technology through the researches carried out over the last few decades; thus, a distributed database seems to be a natural candidate for processing data-heavy applications like training artificial intelligence video recognition models [27] in a distributed and decentralized way. Distributed data implementations in database systems have attracted many researchers to attempt to bring this procedure into blockchain based frameworks. Numerous approaches like weighted models [14], [18], [20], off-chain [15], [25], on-chain [10], blockchain sharding [21], [32], [34] are proposed in recent researches to improve the performance of the blockchain. Various methodologies attempt to ease the burden of individual nodes keeping up the decentralized system and increasing the performance. A few researchers have concentrated on forming clusters utilizing the nodes in the system and afterward convey the information among them to enhance performance and reduce latency; while other groups have concentrated on partitioning the data only, and not on making clusters of nodes. The study [13] is an example of the former approach. It proposes an efficient shard formation protocol to form clusters of nodes. Their effort is only to utilize the nodes in a cluster for validating a transaction and thus, minimizing inter-cluster communications. They ensure safety and liveness with their distributed protocol. [17] proposes a service-oriented

sharded blockchain to enhance scalability while [8] takes the same approach as in [13]. However, they allow intra-shard and cross-shard transactions. Intra-shard transactions access only a data shard of the respective cluster while cross-shard transactions access more than one data shards across clusters. This work [30] also proposes to partition the network into some asynchronous zones. Each group of nodes in a zone runs independently and in parallel; thus achieve linear scalability. As a case of the latter approach (dividing data only), the work [12] of Mingjun Dai et al. presents Network-Coded Distributed Storage (NC-DS) for blockchain platforms. They suggest to actually divide the blockchain into shards and then store these shards in nodes after encoding using linear transformations or bit-wise operations. While their approach reduces the storage requirement and bandwidth consumption, the overhead of encoding and decoding the shards is often significant. Another notable work is [33] where they dynamically divide the blockchain structures into segments. A node needs to show PoW (Proof of of work) [11], [29] to gain membership in the network and also to prove the claim of storing a segment. The system does not need to have a majority of the honest nodes. They ensure that each segment is stored in at least a reliable keeper by classifying the nodes using a hypothesis called the jury hypothesis [32]. However, their approach is not suitable for permissioned blockchain as PoW is a CPU-heavy protocol. The complexity in achieving a fully working solution is further validated by Ethereum's [31] progress of implementing shard-chains which they term danksharding [2] now which is still several years away according to them. This makes the topic more interesting for blockchains like HLF.

In this work, we adopt the latter approach Instead of forming clusters of nodes we focus on reducing the storage needed at each node. Our method does not need any encoding and thus like some methods described above which saves a lot of computational power. On top of that, we propose configurable fault tolerance and storage reduction based on the state of network consortium. Finally, we test sharded fabric network with Hyperledger Caliper benchmark tool and get valuable insights on storage reduction to compare with other work. Our proposal is to get rid of the need for whole chain by partitioning data over participating nodes while ensuring fault tolerance. We do so in the popular Hyperledger Fabric [9] blockchain which is a permissioned blockchain framework with modular architecture. Moreover, the permission boundary is a channel which represents a private ledger inside the network with defined and agreed membership of participants. We make use of the fact that, the joined members agreed on the specifics of the channel so there should not be any issue in spawning another channel with the same criteria. This way we create a new channel(shard) based on a number of block threshold (shard size) and enable transactions as usual while all shards are not available in a single node.

In next sections we have covered the HLF concepts with literature, showed the proposed architecture, workflow and algorithms to illustrate our solution. The organization based node and membership along with channel and SDK for development in Hyperledger Fabric ecosystem is described. We have explained our methodology and experimental configuration. The experimental evaluation for storage usage is also presented which shows a controllable reduction in storage with fault tolerance change and introduction of shard. The experiment is done for an example scenario which is very representative for any other scenario in private blockchain with three peers in an organization. We showed the usual setup and our proposed setup for the example and results show more than 18% reduction in storage consumed for transaction payload of 100KB. This percentage is for the practical deployment of sharding which gets closer to the basic mathematical formulae for our method as payload increase.

II. Background and Methodology

Hyperledger Fabric is the most popular blockchain for permissioned participation [24]. The root of this popularity lies in its modular architecture where the designers offered an organizational structure and collaboration framework among organizations. Fig. 1 shows only the part associated with this work. Here, the diverse interaction among organizations through channel is represented. Many organizations can join a channel and one organization may join in many channels based on real world scenario. We can present the transaction flow sequence diagram in Fig. 2 for easy understanding of the process. Note that in our case the client is Hyperledger Caliper which uses concept of HLF Gateway to initiate transaction as in the Fig. 2.

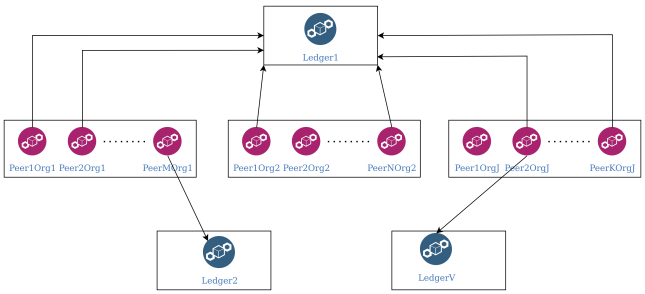


Fig. 1: Hyperledger Fabric General Channel Participation

The details of every components is available in Hyperledger documentation [5] and source codes are available in Github [3].

We take a novel approach for storage reduction in Hyperledger Fabric and verify the storage reduction measurements. As Figure 3 shows, the basic motivation to divide the ledger into shard is to enable dynamic membership of several peers in a shard and thus the whole chain. To show the storage reduction in HLF, we choose a sample network like HLF which itself provides test network in fabric samples. Figure 3(a) shows our sample network where there are three peers joining in the channel from one organization Org1. If the ledger size is S then the total space occupied would be $3 * S$.

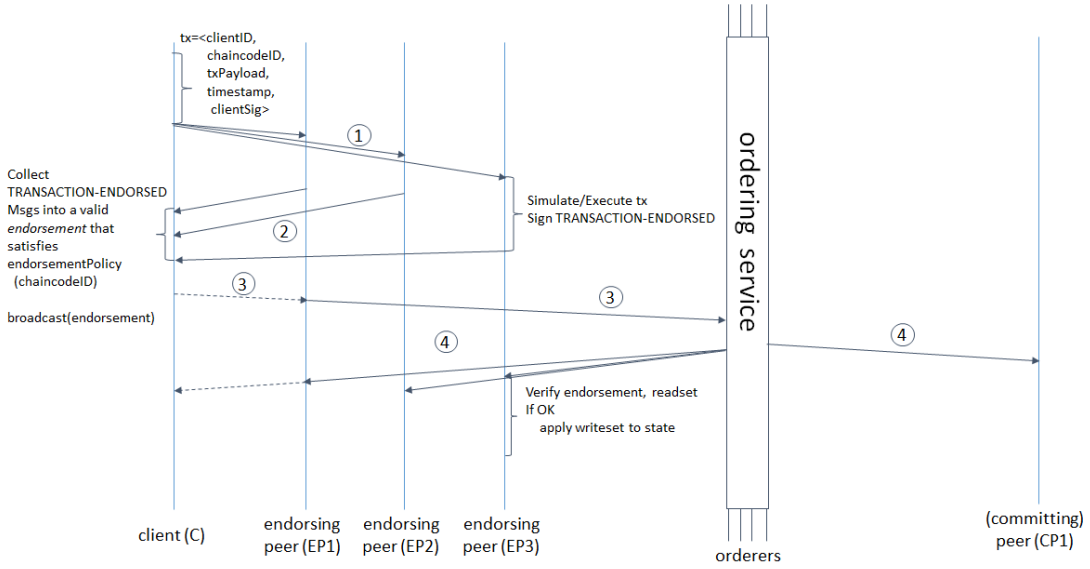


Fig. 2: Hyperledger Fabric Transaction Flow [4]

Now we present our change to the network in Fig. 3(c) where we divide the ledger in three shards and join 2 peers in each of the shard so that each shard has a copy of it in two of the joined peers. The FT can be reduced like Fig. 3(b) where there is no guarantee of ledger size and peers don't get options to join specific shard as needed unlike in Fig. 3(c). Now, we move to theoretical storage reduction calculation where the original ledger of size S is divided into 3 parts leaving the shard size T to be-

$$T = \frac{S}{3}$$

As we distribute each shard to at most two peers here, to ensure fault tolerance (FT) of 1 node as a minimal, the total storage requirement (T_n) for sharding becomes-

$$T_n = 2 * T * 3 = 2 * \frac{S}{3} * 3 = 2S$$

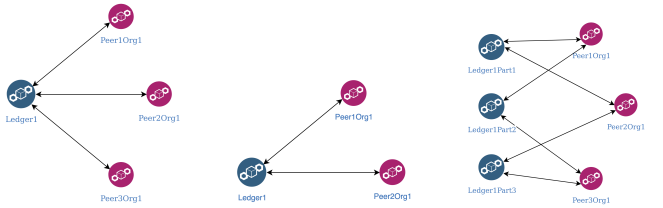
We can now calculate the storage reduction (R) with the following equation-

$$R = \frac{3S - 2S}{3S} * 100\% = 33.33\%$$

This is an optimistic calculation and a baseline of comparing results. Also this equations are specific to the idea shared in Figure 3. For any other scenario (of course there will be many based on organizations and their requirements) designers may devise their own mathematical form and find the baseline. In practice, R is much lower than the baseline calculation for block header and configuration related overhead of transactions, blocks and channels. With regards to the system reliability, the same baseline stands as in [22] for our work as well.

A. *Flowchart and Key Functions*

In Fig. 4, we have outlined basic steps for sharding keeping state of the blockchain configuration in mind. We can identify five major tasks in our experimental work as described below-



(a) One common ledger for all peers (b) Trade-off of FT by peer removal (c) Splitting the ledger in 3 shards

Fig. 3: Sharded vs. Non-sharded

- **Shard Creation:** Shard creation is performed when the limit is crossed for a shard and the participating peers need to rejoin with the same configurations in a new shard as shown in Fig. 5(a). In Fig. 5(b) a generic view of shard creation over time is presented. The channel creation is only supported by command line interface of Hyperledger Fabric peers or need to customize the peer source code available in Github which is the most challenging task. We have choosen the later approach since using CLI will not help achieving the automatic shard creation goal. We have used Hyperledger Fabric Java SDK by REST API calling in our custom implementation of Hyperledger Fabric peer source and achieved shard generation functionality. For shard initiation the sample configuration described in III-A1 is used in test network where three active shards at a time is expected which must be created on network startup. The number of shard may change based on change in network structure. Since our goal is to demonstrate storage reduction for this work, we are sticking to the sample test network.
- **Chaincode Deployment in New Shard Channel.** Once a new channel is created as a shard for the chain it

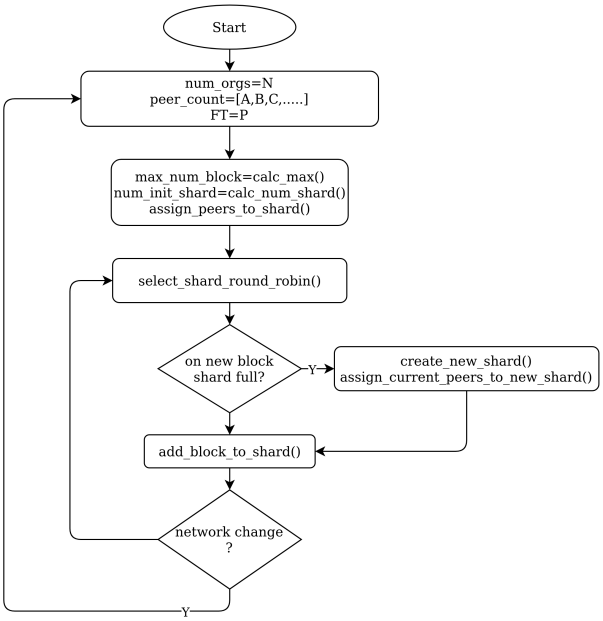


Fig. 4: Flowchart of Sharding

is initially empty. The preexisting chaincodes for its predecessor will not be inherited automatically. So, the chaincodes must be programmatically deployed to new shard channel to be able to execute transaction on it. Adding a chaincode to the channel is a multi-step process where the participating peers will interact to make it ready to execute. Moreover there are two options [Fig. 6] to deploy chaincode for channel starting from HLF V2.0-

1) **Embedded Chaincode** where the chaincode run attached with the peer machine. This is the default way. The steps need to be executed from CLI in default HLF. As it is the default way no additional overhead is there to the setup process except deployment steps. Since, only one instance of chaincode is launched, this way incurs low scalability.
2) **Chaincode as External Service** allows the chaincode to run detached from the related peer. It supports multiple such instance and using a proper load balancing approach the system can scale more compared to embedded setup. It comes with the additional burden of configuration steps compared to embedded approach.

For this experiment we used the embedded approach for simplicity and considering the fact that scalability is not in the scope of this work. For prospective work with scalability analysis both of the approaches can be experimented. Regardless of the deployment options the steps are same for the deployment itself as described below [7] -

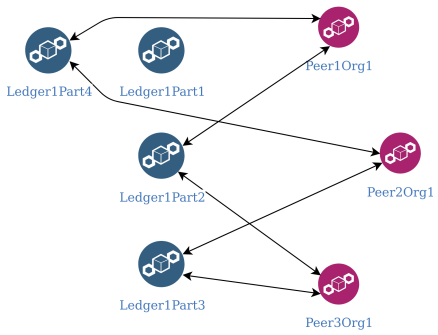
1) **Package the chaincode:** This step refers to compilation and packaging of chaincode source. Since chaincode can be written in Java, Go or Node.js, respective build processes will be followed to produce the chaincode package for installation in peer channel. Only one organization for all or each organization individually can complete this step.
2) **Install the chaincode on your peers:** All endorsing organization must install the packaged chaincode on its peer/s according to the network configuration and agreed endorsement policy.
3) **Approve a chaincode definition for your organization:** Any organization using the chaincode must complete this step. The chaincode definition must be approved by a sufficient number of organizations to satisfy the channel's LifecycleEndorsement policy (a majority, by default) before the chaincode can be started on the channel.
4) **Commit the chaincode definition to the channel:** If sufficient approval from participating organizations is attained only one organization needs to execute the commit transaction to start the chaincode in the channel for submitting transactions.

The final view of the deployed chaincode after committing chaincode definition to the channels will be like Fig. 7. For this experiment we have achieved automatic deployment of the chaincode by customizing the peer with the help of several shell scripts and SDK.

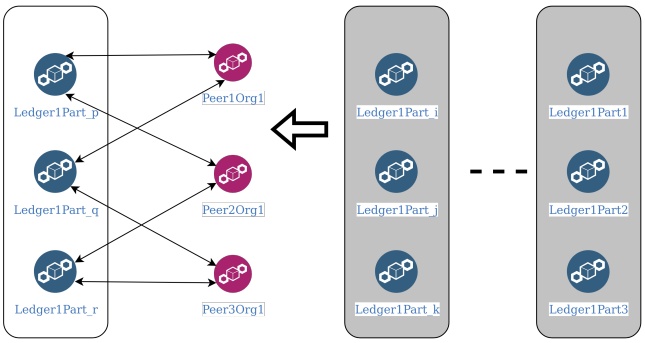
- **Meta-data Update on Transaction:** The transaction initiated by client is assigned to a shard in a round-robin fashion. The shard/s for a key's write/update can be stored in a MySQL table as a temporary solution. A good choice will be to utilize the state DB of Hyperledger Fabric channel to get optimal results and achieve less point of failure. We are skipping this as a future work.
- **Querying Result:** This part will be related to II-A. The easiest way is to find the channel/shard from query key from MySQL/State DB table and retrieve the value from proper channel.
- **Configuration change:** This experiment is done with a sample configuration. The configuration change parameters are listed and yet to be implemented. The realistic scenario is to experience network configuration changes like number of peers, organizations, chaincode, fault tolerance etc for a ledger along with shard size and transaction size. To maintain a balanced trade-off of system reliability and FT that provides desired storage reduction, the constraints on those numbers can be set. Keeping the configuration changes out of scope of this work serves as a starting point showing how to dynamically join peers in a channel for one organization. The same can be achieved for additional organizations.

B. Algorithm for Transaction, Shard Generation

As in II-A the full workflow is described for shard generation and configuration change. We limit our scope of work to

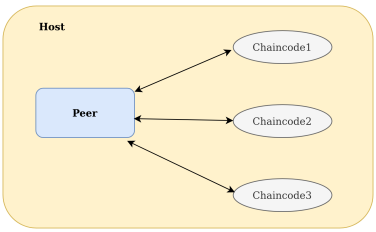


(a) Network change for new shard compared to 3(b)

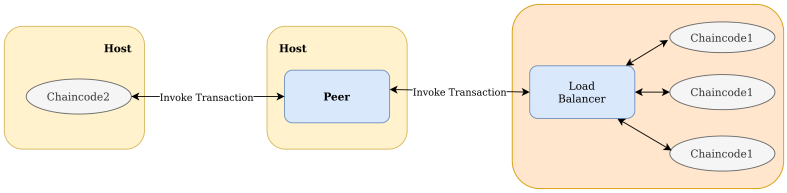


(b) History of shard generation

Fig. 5: Shard creation flow



(a) Embedded Chaincode



(b) Chaincode as External Service

Fig. 6: Chaincode Deployment Options in HLF

transaction Algorithm 1 only . The other functionalities will be implemented in subsequent work.

Algorithm 1 Transaction Algorithm

```
1: function TRANSACTION(rw_set, network_config, shard)
2:     max =read_max_num_of_block(network_config)
3:     num_block =read_num_of_block(shard)
4:     if num_block == max then
5:         shard = create_shard(network_config)
6:     end if
7:     do_transaction(rw_set, shard)
8:     update_metadata(rw_set, shard)
9: end function
```

III. Experimental Evaluation

In this section we will describe the experimental setup, network structure and chaincode description for this work.

A. Experiments

Firstly, we run chaincode for two different shard limit for each sharded channel. The limits we used are 500 and 1000 blocks per shard for respective run. We executed a varied number of transactions and took normalized storage measure of 10K transactions for each limit and compared result with no sharding scenario. For this experiment the number of transactions per block is set to 10. There is a configuration parameter of batch timeout in the fabric network which is set to 12 seconds. If the specified maximum transactions per block is not reached within this batch timeout the block will be produced with available number of transactions. For transaction size we use payload of 1KB, 5KB, 20KB and 40KB binary data as the value of key. The key is generated as random alphanumeric string.

Secondly, we initiated transactions for the combination of shard limit (10,50,100,200,500), transaction size (1KB,5KB,20KB,40KB) and transaction per block (1,5,10,15,20,50) to calculate and observe the variance in storage consumption and latency.

1) Network Structure: As shown in Figure 3, with three peers joining the non-sharded channel and three shard channels each with a combination of 2 peers out of 3 making up initial network configuration. When a shard is full a new shard is created with the same members as in the old shard thus maintaining the chronological structure over the lifetime of the network which is visualized in Fig. 5.

2) Experimental Setup: We have done this experiment in containerized environment using docker. The configuration of machine used here is presented in Table I. We have created nine peers, three for each three organization with pre-built docker image provided by Hyperledger. Similarly we have created one CAs and ordering service following Raft consensus protocol. We connect the nodes using a docker overlay network and execute script from inside our project source to complete handshake and join peers accordingly. We

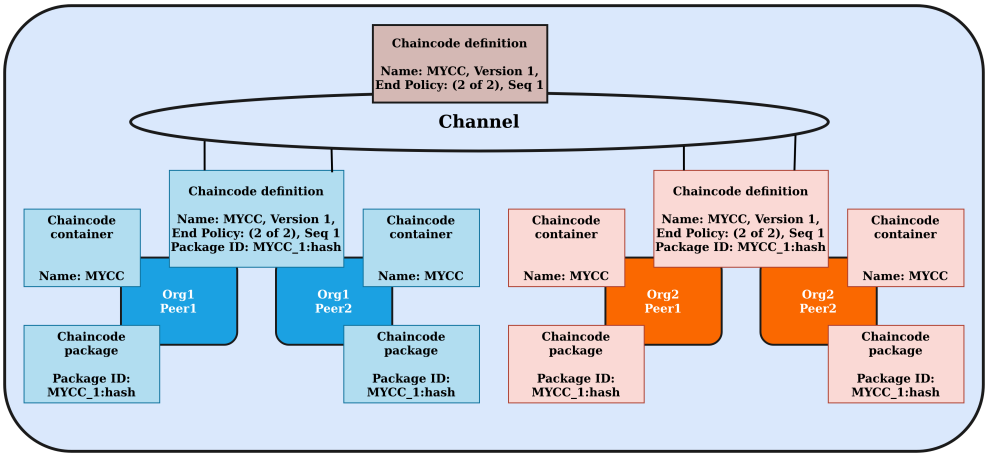


Fig. 7: Schema of a Deployed Chaincode MYCC in 2 Org 2 Peer Network [7]

simulate installation and endorsement of the asset chaincode (described later) in the network to start transactions. Here, the version of Hyperledger Fabric is 2.2 LTS (Long Term Support) as can be seen from the Table II. This is one of the recent LTS version for Hyperledger Fabric and used widely.

TABLE I: Machine used

Machine	CPU	RAM	OS
Dell Laptop	Intel core i-3, 4-core @2.4GHz	16GB	Ubuntu 18.04

TABLE II: Docker Images of Fabric for Experimental Setup

Image	Count	Version tag
CA	1	1.4.7
Orderer	1	2.2
Peer	3	2.2
MySQL	1	5.7

3) Chaincode Description: A chaincode is the synonym for a smart contract of Ethereum and other block-chains in HLF. Like smart contract chaincode embodies some if-else like rules to produce transaction for the blockchain. We have used the HLF samples provided implementation of simple asset chaincode with write functionality since the focus is on storage consumption reduction measurement. The chaincode is written in Java language and installed in all peers internally. The policy is to have approval from all participating peers. As in the example scenario two peers from one organization joins one shard and all of them must agree on asset update by chaincode. The method is as below-

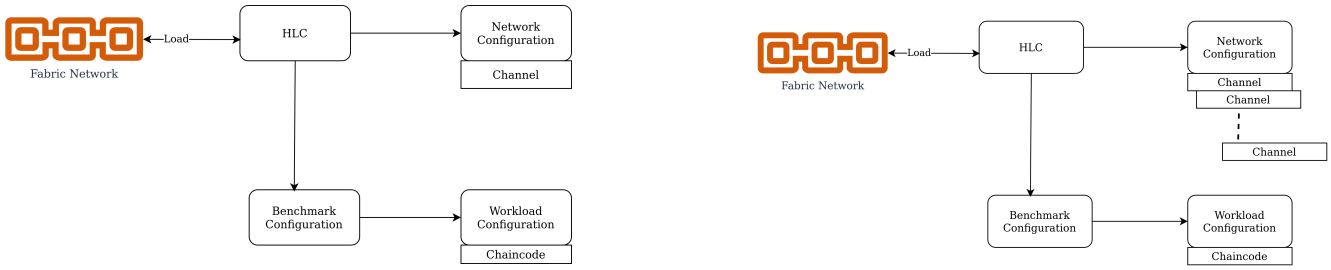
- WriteAsset(key, value)

4) State DB: Between Level DB and Couch DB we choose Level DB for its less overhead compared to Couch DB in configuration steps. Since, the storage measure is for the ledger data instead of state DB data and query functionality is skipped, choosing either is same in our context. In future works involving query functionality, performance for both state DB may be analysed.

5) Endorsement Policy: This policy specifies the peer nodes on a channel required to execute transactions related to a specific chaincode application, and the needed combination of responses (endorsements) from peers. Policy can be defined in terms of minimum number of endorsing peers, a minimum percentage of endorsing peers, or by all endorsing peers that are attached to a specific chaincode program. Policies can be tuned based on the application and the required level of protection against misbehavior by the endorsing peers. No transaction will be added to ledger by HLF ordering service until the endorsement policy is satisfied by getting valid status from required committing peers. For our experiment we used very basic endorsement policy where one of the peer must endorse the transaction for both sharded and non-sharded configuration.

6) Experimental Load: Hyperledger Fabric has its own benchmark tool named Hyperledger Caliper (HLC) to generate load and obtain results. HLC is limited to generate load for a single channel only. In our implementation the channels in question are always changing once the shard limit is reached. This makes HLC unable to generate load for sharded version. To make testing feasible using HLC we changed HLC to support our customized HLF. The version of HLC used in this experiment is 0.4. To generate load HLC need three configuration files-

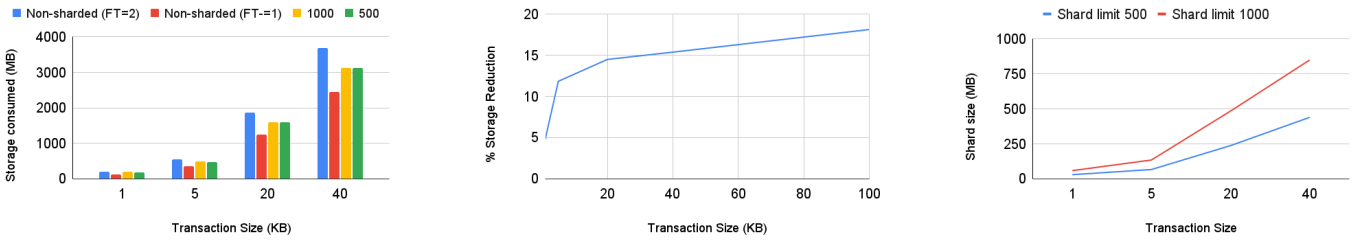
- **Network Configuration File:** This file contains the networking related information of the blockchain along woth required certificates to be able to execute transaction. The channel name where transaction will be executed is specified in this file.
- **Workload Configuration File:** In workload file the initiation of transaction and generation of test values are covered. The actual call to chaincode method is also defined here.
- **Benchmark Configuration File:** Benchmark configu-



(a) One common channel for load

(b) No fixed channel for sharded network

Fig. 8: HLC for Sharded and Non-sharded Network



(a) Storage consumed for 10K transactions

(b) % of storage reduction

(c) Shard size

Fig. 9: Storage Analysis

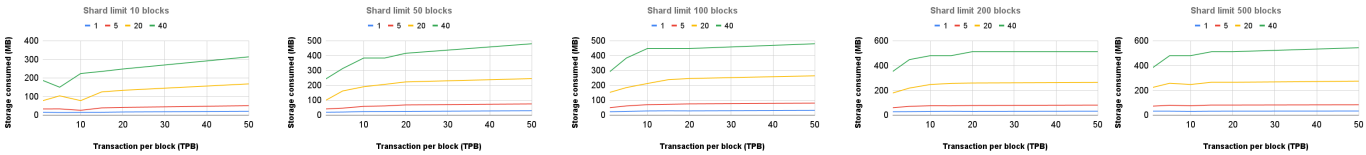
ration file defines the number of worker, load time and target workload along with chaincode name.

In Fig. 8 the difference of HLC for two types of setup is shown. In HLC source we made changes to accept endorsement responses from channels other than the one used in first place thus achieving our goal. Here if the invocation is with channel **X** but returned endorsements from peer are from channel **Y** instead of flagging an error we proceed to step 3 of Fig. 2 which subsequently executes step 4. In customized caliper we mark this as success to display consistent result. For verification of the validity of this approach we manually executed query for the specific channel **Y** from CLI to retrieve the value for the key which is working as expected.

B. Results

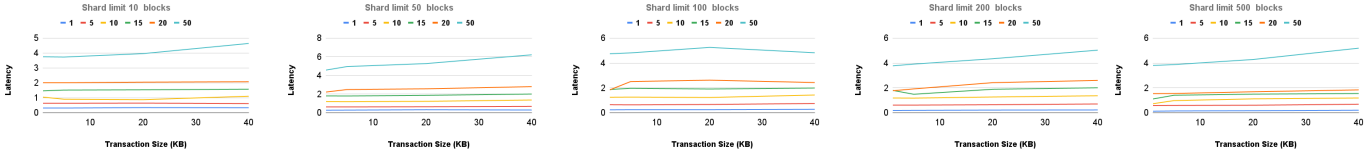
We present the results in Fig. 9 - 11. The amount of storage consumption compared to non sharded architecture is shown in Fig. 9(a). It can be noted that the storage consumed for a specific number of transactions for both 500 and 1000 shard limit is almost identical with one transaction per block, It is very trivial that, when the transaction size is higher the shard size is linearly higher. However having a control of the shard limit will give flexibility on other application specific aspects. The graph in Fig. 9(b) shows the reduced storage usage with respect to transaction size for sample chaincode. We further validate the storage measures and latency variance with respect to shard limit, transaction per block, transaction size in Fig. 10 and Fig. 11 respectively. For storage reduction we find the impact of these parameters to be negligible. The latency vary very similarly for different shard limit , transaction size and TPB. This implies while choosing shard limit for specific network and application the designer need not think much about those parameters. A fitting combination of shard limit can be chosen based on the needs as well as the transaction size, transaction per block can also be chosen for specific applications as needed. As the results suggest for smaller transaction size the storage consumption is close to non-sharded version. This is because the actual data in transaction is superseded by header requirements for transactions. As we increase the transaction size the overhead of header information becomes less visible. For 100KB transaction size it stands at around 18.114% compared to non-sharded configuration. This % of reduction should increase more with increasing transaction size. Still there will be some deviation form expected reduction of 33.33% due to exclusion of transaction header, block header and channel header in our calculation. The information in Fig. 9(c) is important in the sense that, even in non-sharded version we could keep 2 peers (Fig. 3 (b)) to reduce more storage consumption, so why we have done sharding. As can be seen from the result by changing transaction size and shard limit we are producing a range of shards with different sizes. This will enable us to use such a system for mobile and IoT devices. Another application would be to implement archiving/caching for Hyperledger Fabric blockchain. The older shards can be put into archive and active shards will be loaded in faster memory. By considering more parameters and configurations more interesting applications can be achieved.

In [22], a simulation is performed to show sharding and



(a) shard limit 10 (b) shard limit 50 (c) shard limit 100 (d) shard limit 200 (e) shard limit 500

Fig. 10: Storage Consumption w.r.t shard limit, TPB and TX size (Legends represent transaction size)



(a) shard limit 10 (b) shard limit 50 (c) shard limit 100 (d) shard limit 200 (e) shard limit 500

Fig. 11: Latency Measure w.r.t shard limit, TPB and TX size (Legends represent transaction per block)

distribution of ledger copies (s) in a cluster (m node per cluster among n nodes). The shard size(|B|) and world state size(|S|). The authors introduced storage rate (R_v) by below equation,

$$R_v = \frac{\frac{s}{m*n} * |B| - |S|}{|B| - |S|}$$

We compare practical storage values with values computed by this equation. We quantitatively get the numbers respective to the simulation results presented in their work. Since, the ratio is shown in their results, we stick with our simple example of 3 shards and extend the overlapping shard number s so that we have s copies as formulated in their work. So our formula of R_v becomes-

$$R_v = \frac{3 * s * S_c}{25 * S}$$

Where, S_c is the observed value in our experiment and S is the non-sharded size of the ledger for 10k transactions. We choose closest representative configurations from our experiment where transaction size is 40KB, shard limit is 10 blocks and 10 transactions per block. For this, $S_c = 480MB$ and $S = 1229MB$ then,

$$for\ s = 2, R_v = \frac{3 * 2 * 480}{25 * 1230} * 100 = 9.37\%$$

$$for\ s = 8, R_v = \frac{3 * 8 * 480}{25 * 1230} * 100 = 37.49\%$$

and so on. In this way, we present the comparison results for several transaction size in Fig. 12

The result shows the actual storage reduction will be less than the simulation due to some overheads. In addition, our calculation is the lower bound for R_v as the overhead added for blocks by more participating node is excluded for $s > 2$. This overhead can be ignored for higher transaction sizes. In accordance with our results in Fig. 9(b), this deviation will reduce with increasing transaction size. As presented in the result, with decreasing transaction size the storage rate is increasing.

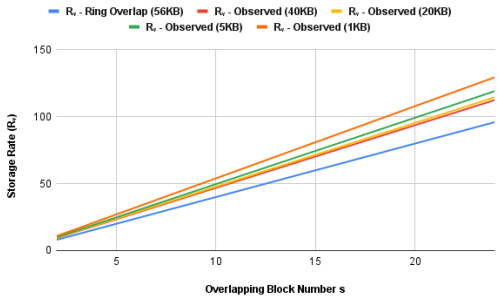


Fig. 12: Ring Overlap Simulation vs Our Experimental Storage Rate

IV. Conclusions

In this paper, we show significant reduction of storage for Hyperledger Fabric. By customizing HLF, we implemented basic shard generation with a configurable shard limit. In addition, HLC is also customized to work with the customized HLF. We showed 18% storage reduction with our proposed configuration change on sample network. We have presented a complete flow of the sharded network considering dynamic participants and the overview of metadata update and query method while implementing the transaction invocation only. We demonstrated the notion of shard limit agnostic characteristic of storage reduction using shard limit of 500 and 1000 as well as the linear relation of shard size to shard limit and transaction size. However, the performance in terms of throughput and latency is not measured in this work. Since the consensus on creating and joining new channel can be implemented with existing process it should not increase the order of latency much and we leave it as a future work. Also for throughput, the creation of shard will take some time but as it will not be a frequent operation and can be controlled by configuration parameters this should not degrade considerably.

REFERENCES

[1] https://bisontrails.co/global-blockchain-sync/.

[2] https://ethereum.org/en/roadmap/danksharding/.

[3] https://github.com/hyperledger/.

[4] https://hyperledger-fabric.readthedocs.io/en/latest/txflow.html.

[5] https://hyperledger-fabric.readthedocs.io/en/release-2.2/.

[6] https://learn.bybit.com/glossary/definition-shard-chain/.

[7] hyperledger-fabric.readthedocs.io/en/latest/chaincode_lifecycle.html.

[8] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. Sharper: Sharding permissioned blockchains over network clusters. In *Proceedings of the 2021 International Conference on Management of Data*, pages 76–88, 2021.

[9] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the thirteenth EuroSys conference*, pages 1–15, 2018.

[10] Shehar Bano, Mustafa Al-Bassam, and George Danezis. The road to scalable blockchain designs. *USENIX; login: magazine*, 42(4):31–36, 2017.

[11] Martijn Bastiaan. Preventing the 51%-attack: a stochastic analysis of two phase proof of work in bitcoin. In *Availab le at http://referaat. cs. utwente. nl/conference/22/paper/7473/preventingthe-51-attack-a-stochasticanalysis-oftwo-phase-proof-of-work-in-bitcoin. pdf*, 2015.

[12] Mingjun Dai, Shengli Zhang, Hui Wang, and Shi Jin. A low storage room requirement framework for distributed ledger in blockchain. *IEEE Access*, 6:22970–22975, 2018.

[13] Hung Dang, Tien Tuan Anh Dinh, Dumitrel Loghin, Ee-Chien Chang, Qian Lin, and Beng Chin Ooi. Towards scaling blockchain systems via sharding. In *Proceedings of the 2019 international conference on management of data*, pages 123–140, 2019.

[14] Ali Dorri, Salil S Kanhere, Raja Jurdak, and Praveen Gauravaram. Lsb: A lightweight scalable blockchain for iot security and anonymity. *Journal of Parallel and Distributed Computing*, 134:180–197, 2019.

[15] Jacob Eberhardt and Stefan Tai. On or off the blockchain? insights on off-chaining computation and data. In *European Conference on Service-Oriented and Cloud Computing*, pages 3–15. Springer, 2017.

[16] Yuan Yu Michael Isard Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, and Pradeep Kumar Gunda Jon Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. *Proc. LSDS-IR*, 8, 2009.

[17] Adem Efe Gencer, Robbert van Renesse, and Emin Gün Sirer. Short paper: Service-oriented sharding for blockchains. In *International Conference on Financial Cryptography and Data Security*, pages 393–401. Springer, 2017.

[18] Damian Gruber, Wenting Li, and Ghassan Karame. Unifying lightweight blockchain client implementations. In *Proc. NDSS Workshop Decentralized IoT Security Stand.*, pages 1–7, 2018.

[19] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. Sharper: Sharding permissioned blockchains over network clusters. *arXiv e-prints*, pages arXiv–1910, 2019.

[20] Sidra Khatoon and Nadeem Javaid. 'blockchain based decentralized scalable identity and access management system for internet of things. Technical report, Working Paper, 2019.

[21] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. Omniledger: A secure, scale-out, decentralized ledger via sharding. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 583–598. IEEE, 2018.

[22] Wenxuan Liu, Donghong Zhang, Chunxiao Mu, Xiangfu Zhao, and Jindong Zhao. Ring-overlap: A storage scaling mechanism for hyperledger fabric. *Applied Sciences*, 12(19):9568, 2022.

[23] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Decentralized Business Review*, page 21260, 2008.

[24] Julien Polge, Jérémy Robert, and Yves Le Traon. Permissioned blockchain frameworks in the industry: A comparison. *Ict Express*, 7(2):229–233, 2021.

[25] Joseph Poon and Thaddeus Dryja. The bitcoin lightning network: Scalable off-chain instant payments, 2016.

[26] Manuel Rodriguez-Martinez and Nick Roussopoulos. Mocha: A self-extensible database middleware system for distributed data sources. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pages 213–224, 2000.

[27] Nathanael Rota and Monique Thonnat. Activity recognition from video sequences using declarative models. In *ECAI*, pages 673–680. Citeseer, 2000.

[28] Khalid Sayood. *Introduction to data compression*. Morgan Kaufmann, 2017.

[29] Marko Vukolić. The quest for scalable blockchain fabric: Proof-of-work vs. bft replication. In *International workshop on open problems in network security*, pages 112–125. Springer, 2015.

[30] Jiaping Wang and Hao Wang. Monoxide: Scale out blockchains with asynchronous consensus zones. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pages 95–112, 2019.

[31] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

[32] Yibin Xu and Yangyu Huang. An n/2 byzantine node tolerate blockchain sharding approach. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, pages 349–352, 2020.

[33] Yibin Xu and Yangyu Huang. Segment blockchain: A size reduced storage mechanism for blockchain. *IEEE Access*, 8:17434–17441, 2020.

[34] Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova. Rapidchain: Scaling blockchain via full sharding. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 931–948, 2018.