SharPer: Sharding Permissioned Blockchains Over Network Clusters

Mohammad Javad Amiri
University of Pennsylvania
mjamiri@seas.upenn.edu

Divyakant Agrawal
University of California Santa Barbara
agrawal@cs.ucsb.edu

Amr El Abbadi
University of California Santa Barbara
amr@cs.ucsb.edu

Abstract

Scalability is one of the main roadblocks to business adoption of blockchain systems. Despite recent intensive research on using sharding techniques to enhance the scalability of blockchain systems, existing solutions do not efficiently address cross-shard transactions. In this paper, we introduce *SharPer*, a scalable permissioned blockchain system. In SharPer, nodes are clustered and each data shard is replicated on the nodes of a cluster. SharPer supports networks consisting of either crash-only or Byzantine nodes. In SharPer, the blockchain ledger is formed as a directed acyclic graph and each cluster maintains *only* a view of the ledger. SharPer incorporates *decentralized flattened* protocols to establish cross-shard consensus. The decentralized nature of the cross-shard consensus in SharPer enables parallel processing of transactions with non-overlapping clusters. Furthermore, SharPer provides deterministic safety guarantees. The experimental results reveal the efficiency of SharPer in terms of performance and scalability especially in workloads with a low percentage of cross-shard transactions.

CCS Concepts

• **Networks** → **Network protocol design**; • **Information systems** → **Distributed database transactions**; • **Computer systems organization** → **Dependable and fault-tolerant systems and networks**.

Keywords

Blockchain, Scalability, Sharding, Consensus, Permissioned

ACM Reference Format:
Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. 2021. SharPer: Sharding Permissioned Blockchains Over Network Clusters. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 18–27, 2021, Virtual Event , China.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3448016.3452807

1 Introduction

A blockchain is a distributed data structure for recording transactions maintained by nodes without a central authority [10]. Blockchain systems are classified into two categories: *permissionless* and *permissioned* systems. While in a permissionless blockchain system, e.g., Bitcoin [34], the network is public, and anyone can participate without a specific identity, a *permissioned* blockchain system, e.g., Hyperledger Fabric [6], consists of a set of known, identified but possibly untrusted nodes which might be placed in data centers, public clouds, or local infrastructures.

Scalability is the ability of a blockchain system to process an increasing number of transactions by adding nodes to the system. Partitioning the data into multiple shards that are maintained by different subsets (i.e., clusters) of non-malicious nodes is a proven approach to improve the scalability of distributed databases, e.g., Spanner [14]. In such an approach, the performance of the system scales linearly with the number of clusters. Recently, sharding has been utilized in both permissionless and permissioned blockchain systems in the presence of Byzantine nodes. Sharded permissionless blockchains, e.g., Elastico [32], OmniLedger [27], and Rapidchain [45], ensure *probabilistic* safety by randomly assigning nodes to *committees* resulting in a uniform distribution of faulty nodes to the different committees. OmniLedger and Rapidchain also support cross-shard transactions using Byzantine consensus protocols.

Sharding techniques have also been used by different permissioned blockchain systems, e.g., Fabric [6], Cosmos [21], RSCoin [22], and AHL [16]. AHL[16], similar to OmniLedger, provides a probabilistic safety. AHL, however, employs a trusted hardware (the technique presented in [13][43][42]) to reduce the size of each committee from ~600 in OmniLedger to 80 nodes. In AHL [16], consensus on the order of cross-shard transactions not only requires an extra set of nodes (called a *reference committee*) but also results in a large number of inter- and intra-committee communications. Furthermore, since a single reference committee processes cross-shard transactions, AHL is not able to process cross-shard transactions in parallel.

In general, large-scale sharded systems, such as Spanner [14], typically partition data into shards and replicate each shard on the nodes of a *pre-determined fault-tolerant* cluster, e.g., based on physical constraints such as a data center with a majority of non-faulty nodes, to guarantee *deterministic* safety. Maintaining data on pre-determined fault-tolerant clusters for the purpose of scalability has also been studied in permissioned blockchains ResilientDB [24] and Blockplane [35]. However, most sharded blockchain systems, e.g., Elastico, OmniLedger, and AHL, operate on a flat homogeneous network of peers and hence *configure* fault-tolerant units by randomly assigning nodes to clusters and provide a *probabilistic* safety guarantee. To guarantee safety with a high probability, such systems need to uniformly distribute faulty nodes across all clusters, resulting in large-size clusters, e.g., ~600 nodes in OmniLedger.

In our previous work [3], we presented a model including a blockchain ledger for sharded permissioned blockchains. In this paper, we expand this model and develop a sharded permissioned blockchain system, *SharPer*, to improve scalability with deterministic safety guarantees. In the presence of pre-determined fault-tolerant clusters, SharPer, similar to large-scale sharded systems, provides deterministic safety guarantees when more than a half (if nodes are crash-only) or two-thirds (if nodes are Byzantine) of the nodes of each cluster are non-faulty. Without such pre-determined clusters, however, and in order to guarantee *deterministic* safety, SharPer assumes that the number of available nodes is much larger than the number of faulty nodes and assigns nodes to clusters so that ensure more than a half (if nodes are crash-only) or two-thirds (if nodes are Byzantine) of the nodes of each cluster are non-faulty. This assumption is reasonable in sharded systems that strive for high scalability, as such systems typically use more reliable infrastructure.

SharPer assigns data shards to the clusters where each cluster processes the transactions that access its shard. If a transaction accesses only a single shard, i.e., an *intra-shard transaction*, the corresponding cluster orders and executes the transaction locally. As a result, intra-shard transactions of different clusters are independent of each other and are processed in parallel. However, for a *cross-shard transaction*, agreement among *all* and *only* involved clusters is required. Nevertheless, if two cross-shard transactions have no overlapping clusters, they still are processed in parallel. Since the ordering of different transactions might be performed in parallel and due to the existence of cross-shard transactions, the blockchain ledger of SharPer is represented as a *directed acyclic graph* including all intra- and cross-shard transactions. Nonetheless, for the sake of performance, the entire blockchain ledger is *not maintained* by any nodes, and nodes of each cluster maintain a *view* of the ledger including the intra-shard transactions of the cluster and only the cross-shard transactions involving this particular cluster. Unlike traditional single-primary consensus protocols, e.g., PBFT [11], in SharPer, multiple clusters each with its own primary compete with each other to order cross-shard transactions. We believe this setting has been encountered neither in traditional consensus protocols nor in coordinator-based sharded systems, leading us to resolve challenges such as conflicting transactions, deadlock situations as well as the failure of primary nodes across different replicated domains.

The main contributions of this paper are:

- SharPer, a permissioned blockchain system that supports the concurrent processing of transactions by clustering nodes into clusters and sharding both data and the ledger. SharPer supports intra-shard as well as cross-shard transactions.
- Two *decentralized flattened* consensus protocols for ordering cross-shard transactions among all and only the involved clusters in networks consisting of either crash-only or Byzantine nodes. The protocols order cross-shard transactions with no overlapping clusters in parallel.

The rest of this paper is organized as follows. The SharPer model is introduced in Section 2. Sections 3 and 4 present consensus in SharPer. Section 5 evaluates the performance of SharPer. Section 6 discusses related work, and Section 7 concludes the paper.

2 The SHARPER Model

In SharPer, the network consists of a set of clusters. The data is partitioned into data shards and a data shard that represents the *blockchain state* and a view of the blockchain ledger are replicated on nodes of each cluster to provide fault tolerance. This section presents the SharPer infrastructure, cluster formation, and the blockchain ledger.

2.1 SharPer Infrastructure

SharPer consists of a set of nodes in an asynchronous distributed system where nodes might be placed in data centers, public clouds, or local infrastructures. Nodes in SharPer either follow the crash or Byzantine failure model. Crash fault-tolerant protocols, e.g., Paxos [31], guarantee deterministic safety in an asynchronous network using $2f+1$ crash-only nodes to overcome the simultaneous crash failure of any f nodes while in Byzantine fault-tolerant protocols, e.g., PBFT [11], $3f+1$ nodes are needed to provide deterministic safety in the presence of f malicious nodes [8].

SharPer uses point-to-point bi-directional communication channels to connect nodes. Network channels are pairwise authenticated, which guarantees that a malicious node cannot forge a message from a correct node. Furthermore, messages might contain public-key signatures and message digests [11]. We denote a message m signed by replica r as $\langle m \rangle_{\sigma_r}$ and the digest of a message m by $D(m)$. For signature verification, we assume that all nodes have access to the public keys of all other nodes. We assume that a strong adversary can coordinate malicious nodes and delay communication to compromise the replicated service. However, the adversary cannot subvert standard cryptographic assumptions.

2.2 Cluster and Shard Formation

In sharded database systems, data shards are assigned to pre-determined fault-tolerant clusters, e.g., cloud environments, to guarantee *deterministic* safety. In particular, if the system has $|P| = \{p_1, p_2, ...\}$ fault-tolerant clusters and each cluster p_i includes $3f_i+1$ Byzantine nodes, the network size would be $3f + |P|$ where $f = \sum_{i=1}^{|P|} f_i$ is the total number of faulty nodes in the system.

Some Sharded blockchain systems, e.g., OmniLedger [27] and AHL [16], on the other hand, *configure* fault-tolerant clusters (called *committees*) themselves and provide *probabilistic* safety guarantees. Given the lack of well-defined fault-tolerant clusters, such systems, assign nodes randomly to the clusters in order to uniformly distribute faulty nodes. In particular, clusters are formed such that for every cluster p_i, with a high probability, $|p_i| \geq 3f_i + 1$ where f_i is the number of faulty nodes within cluster p_i. To achieve a high probability, e.g., $1{-}2^{-20}$, however, the clusters need to be large-sized, e.g., 80 nodes in AHL. Moreover, to prevent security attacks, clusters are reconfigured periodically.

SharPer in the presence of pre-determined fault-tolerant clusters, i.e., similar to large-scale sharded databases provides deterministic safety guarantees with $2f + |P|$ crash-only or $3f + |P|$ Byzantine nodes where f is the total number of faulty nodes in the system and $|P|$ is the number of clusters. In SharPer, the goal is to provide *deterministic* safety guarantees, hence, without such pre-determined clusters, the number of nodes, N, is assumed to be much larger than f, thus, nodes are partitioned into clusters each large enough to tolerate f failures. The trusted hardware technique can also be utilized in SharPer resulting in enhanced performance.

Nodes are assigned to clusters based on their geographical distribution, i.e., nodes that are in close proximity are assigned to the same cluster to reduce the latency of intra-cluster communication. We denote the set of clusters by $P = \{p_1, p_2, ..., p_{|P|}\}$. Since there are $|P|$ clusters, the data is also sharded into $|P|$ shards, i.e., d_1, ..., $d_{|P|}$, shard d_i is replicated on the nodes of cluster p_i.

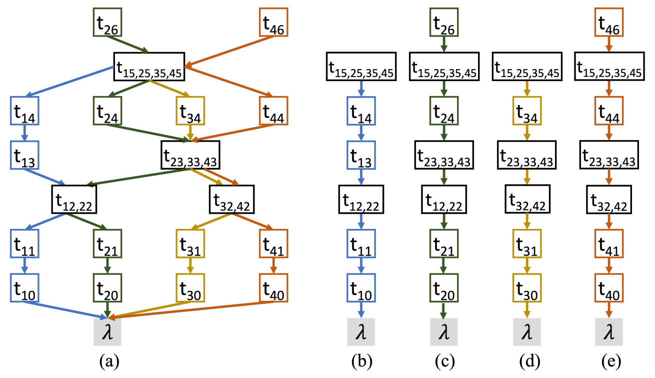


Figure 1: (a): A ledger consisting of four shards, (b), (c), (d), and (e): The views of the ledger from different shards

To ensure high performance, an appropriate sharding needs to be *workload-aware*, i.e., have prior knowledge of the data and how it is accessed by transactions. Workload-aware sharding increases the probability of transactions accessing records in a single shard [15]. If sharding is not workload-aware, transactions will be processed by multiple, possibly far apart, clusters. Establishing consensus among all those involved clusters, although correct, will severely impact the overall performance. Different approaches have been proposed to minimize the number of cross-shard transactions [37], nevertheless, there might still be a portion of transactions that accesses records across multiple shards. As a result, SharPer supports both *intra-shard* and *cross-shard* transactions.

2.3 Blockchain Ledger

Blockchain systems record transactions in the form of a hash chain in an append-only data structure, called the *blockchain ledger*. In SharPer, each data shard is replicated on all nodes of a cluster. As a result, to ensure data consistency, a total order among transactions (both intra- and cross-shard) that access the same data shard is needed. Note that the total order imposed by blockchain is less flexible than serializability, the common correctness criterion in databases, that allows transactions to be executed in a different order. The total order of transactions in the blockchain ledger is captured by *chaining* transaction blocks, i.e., each block includes a sequence number or the cryptographic hash of the previous transaction block. For simplicity and without loss of generality, we assume each block consists of a single transaction[1]. Since more than one cluster is involved in each cross-shard transaction, similar to Caper [2], the ledger is formed as a *directed acyclic graph*. The ledger also includes a unique initialization block, called the *genesis* block.

Fig. 1(a) shows a blockchain ledger created in the SharPer model consisting of four clusters p_1 to p_4 (data shards d_1 to d_4). In this figure, λ is the genesis block. Intra- and cross-shard transactions are also specified. For example, t_{10}, t_{11}, t_{13}, and t_{14} are the intra-shard transactions of cluster p_1. Each cross-shard transaction is labeled with $t_{o_1,..,o_k}$ where k is the number of involved clusters and o_i indicates the order of the transaction among the transactions of the i^{th} involved cluster. For example, $t_{12,22}$ is a cross-shard transaction that accesses data shards d_1 and d_2. A cross-shard transaction requires a sequence number from every involved cluster to ensure that the transactions are ordered correctly with respect to the intra-shard transactions of all involved clusters. Transactions that access a data shard form a total order e.g., t_{10}, t_{11}, $t_{12,22}$, t_{13}, t_{14}, and $t_{15,25,35,45}$ are chained. Intra-shard transactions of different clusters, e.g., t_{11}, t_{21}, as well as non-overlapping cross-shard transactions, e.g., $t_{12,22}$ and $t_{32,42}$, can be appended to the ledger in parallel.

In SharPer, the entire blockchain ledger is *not maintained* by any cluster and each cluster maintains only its *view* of the ledger. The ledger is indeed the union of all these physical views. Fig. 1(b)-(e) show the views of the ledger for clusters p_1, p_2, p_3, and p_4 respectively. As can be seen, each cluster p_i maintains only a (linear) view of the ledger consisting of the intra-shard transactions of p_i and the cross-shard transactions that access d_i.

3 Consensus with Crash-Only Nodes

In a permissioned blockchain system, nodes establish consensus on a unique order in which entries are appended to the blockchain ledger. To establish consensus, asynchronous fault-tolerant protocols have been used. Fault-tolerant protocols use the State Machine Replication (SMR) algorithm [29] where nodes agree on an ordering of incoming requests. The algorithm has to satisfy four main properties [9]: (1) *agreement:* every correct node must agree on the same value, (2) *Validity (integrity):* if a correct node commits a value, then the value must have been proposed by some correct node, (3) *Consistency (total order):* all correct nodes commit the same value in the same order, and (4) *termination:* eventually every node commits some value. The first three properties are known as *safety* and the termination property is known as *liveness.* Consistency is a trivial property in consensus protocols with a single ordering routine, however, since multiple clusters with different ordering routines are involved in SharPer, consistency between different instances of the consensus algorithm needs to be guaranteed. As shown by Fischer et al. [20], in an asynchronous system, where nodes can fail, consensus has no solution that is both safe and live. Based on that impossibility (FLP) result, in SharPer, safety is guaranteed in an asynchronous network, however, a synchrony assumption is needed to ensure liveness. Due to the trust assumptions of blockchains, most existing blockchain systems employ Byzantine fault-tolerant protocols. Studying crash fault-tolerant protocols, however, is beneficial for two main reasons. First, it can be used in permissioned blockchain systems with more federated settings, e.g., Hyperleger Fabric [6] uses crash fault-tolerant protocol Raft [36]. Second, from a development point of view, it is pedagogically easier to introduce the complex concepts used in Byzantine consensus protocols. In this section, we first show how consensus is established in SharPer for intra-shard and cross-shard transactions in the presence of crash-only nodes. Then, the primary failure handling routine of SharPer is presented and finally, the correctness of SharPer is proven.

3.1 Intra-shard consensus

Crash fault-tolerant protocols, e.g., Paxos [31], guarantee safety in an asynchronous network using $2f+1$ nodes to overcome the simultaneous crash failure of any f nodes. SharPer uses multi-Paxos, a variation of Paxos, where *the primary* (a pre-elected node that initiates consensus) is relatively stable, to establish consensus on

[1] Each block could include multiple *consecutive* intra-shard or *consecutive* cross-shard transactions (but no combination of both). It is indeed a performance trade-off, while in highly loaded geo-distributed settings, batching transactions into blocks is beneficial, in lightly loaded workloads where nodes are placed in close proximity, as demonstrated in StreamChain [26], batching transactions into blocks reduces performance.

Algorithm 1 Cross-shard Consensus with Crash-Only Nodes

```
1: init():
2:     $r$ := node_id
3:     $p_i$ := the cluster that initiates the consensus (initiator cluster)
4:     $\pi(p)$ := the primary node of cluster $p$
5:     $P$ := set of involved clusters
6: upon receiving valid request $m$ and ($r == \pi(p_i)$)
7:     multicast ⟨PROPOSE, $h_i$, $d$, $m$⟩ to $P$
8: upon receiving valid ⟨PROPOSE, $h_i$, $d$, $m$⟩ from primary $\pi(p_i)$
9:     if $r$ is not waiting for commit message of request $m'$ where $m$ and $m'$ intersect
       in some other cluster $p_k$
10:        send ⟨ACCEPT, $h_i$, $h_j$, $d$, $r$⟩ to primary $\pi(p_i)$
11: upon receiving $f$+1 valid matching ⟨ACCEPT, $h_i$, $h_j$, $d$, $r$⟩ from every cluster $p_j$ in
    $P$ and node is $\pi(p_i)$
12:    multicast ⟨COMMIT, [$h_i$, $h_j$, ..., $h_k$], $d$⟩$_{\sigma_{\pi(p_i)}}$ to $P$
13:    append the transaction and commit message to the ledger
14: upon receiving ⟨COMMIT, [$h_i$, $h_j$, ..., $h_k$], $d$⟩$_{\sigma_{\pi(p_i)}}$ from $\pi(p_i)$
15:    append the transaction and commit message to the ledger
```

the order of intra-shard transactions. In SharPer, upon receiving a signed request (i.e., transaction) from a client, the primary assigns a sequence number to the request (to provide a total order among requests) and multicasts a propose message (called accept in Paxos) including the transaction to every node within the cluster. Instead of a sequence number, the primary can also include the cryptographic hash of the previous transaction block, $H(b)$, in the message where $H(.)$ denotes the hash function and b is the previous block that is ordered by the cluster. Upon receiving a valid propose message from the primary, each node sends an accept (i.e., accepted) message to the primary. The primary waits for f accept messages from different nodes (plus itself becomes $f + 1$), multicasts a *signed* commit message to every node within the cluster, appends the transaction block including the transaction and the signed commit message (as evidence of the transaction's validity) to the blockchain ledger, executes the transaction, updates the blockchain state (data shard), and sends a reply to the client. We assume that all transactions are executed deterministically in the system. Upon receiving a commit message from the primary, each node appends the transaction block (i.e., the transaction and the received commit message) to its blockchain ledger, executes the transaction and updates the state. The client also waits for a valid reply from the primary to accept the result. Since commit messages include the digest (cryptographic hash) of the corresponding transactions, appending valid signed commit messages to the blockchain ledger in addition to the transactions, provides the same level of *immutability* guarantee as including the cryptographic hash of the previous transaction in the transaction block, i.e., any attempt to alter the block data can easily be detected.

3.2 Cross-Shard Consensus

Cross-shard transactions access records from data shards which are maintained by different clusters. This section presents how SharPer processes cross-shard transactions on crash-only nodes.

Algorithm 1 presents the normal case operation for SharPer to process a cross-shard transaction in the presence of crash-only nodes. Although not explicitly mentioned, every sent and received message is logged by the nodes. As indicated in lines 1-5 of the algorithm, p_i is the *initiator* cluster, i.e., the cluster that initiates the transaction, $\pi(p)$ is the primary node of cluster p, and P is the set of involved clusters in the transaction.

A cross-shard transaction is sent by a client to the (pre-elected) primary node of a cluster (i.e., one of the clusters that store data records accessed by the transaction). Note that once a primary node of a cluster is elected, it initiates all intra-shard transactions of the cluster as well as cross-shard transactions that are sent to the cluster by clients. As shown in lines 6-7, upon receiving a valid signed cross-shard transaction $m = \langle \text{REQUEST}, op, t_c, c\rangle_{\sigma_c}$ from an authorized client c (with timestamp t_c) to execute operation op, the primary node $\pi(p_i)$ of cluster p_i (called *initiator primary*) assigns sequence number h_i to the request and multicasts a propose message $\langle \text{PROPOSE}, h_i, d, m\rangle$ to *all* nodes of all *involved* clusters, i.e., clusters that store data records accessed by the transaction, where m is the client's request message and $d = D(m)$ is digest of m. Timestamp t_c is used to ensure exactly-once semantics for the execution of requests and prevent replay attacks. The timestamps for each client's requests are totally ordered. The sequence number h_i represents the correct order of the transaction in cluster p_i. Since all nodes are crash-only, there is no need to sign messages.

Upon receiving a propose message, as indicated in lines 8-10, each node r of an involved cluster p_j validates the message and its sequence number. If node r of cluster p_j is currently waiting for a commit message of some cross-shard request m' where the involved clusters of two requests m and m' intersect in p_j as well as some other cluster p_k, the node does not process the new request m (only buffers m) before the earlier request m' gets committed. This ensures that cross-shard requests are committed in the same order on overlapping clusters (consistency), e.g., m and m' are committed in the same order on both p_j and p_k. Otherwise, the node sends an accept message $\langle \text{ACCEPT}, h_i, h_j, d, r\rangle$ to the initiator primary node $\pi(p_i)$ where h_j is the sequence number assigned by r, which represents the correct order of request m in cluster p_j.

Once initiator primary $\pi(p_i)$ receives valid matching accept messages from f+1 nodes (out of $2f$+1 nodes) of *every* involved cluster p_j with matching h_j and also h_i and d that match its sent propose message, as presented in lines 11-13, it collects all valid sequence numbers (e.g., h_i, h_j, ..., h_k) from the accept messages of all involved clusters (e.g., p_i, p_j, ..., p_k) and multicasts a commit message $\langle \text{COMMIT}, [h_i, h_j, ..., h_k], d\rangle_{\sigma_{\pi(p_i)}}$ to the nodes of all involved clusters. The order of sequence numbers h_i, h_j, ..., h_k in the message is in ascending order determined by their cluster ids. In fact, the sequence number consists of multiple sub-sequence numbers where each sub-sequence number presents the local order of the transaction in one of the involved clusters. The initiator primary signs its commit messages because they might be used later by nodes to prove the correctness of the transaction block.

Finally, as shown in lines 14-15, once a node of some cluster p_j receives a valid signed commit message from the initiator primary $\pi(p_i)$, the node considers the transaction as committed (even if the node has not sent an accept message for that request). If all transactions with lower sequence numbers than h_j have been committed, the node appends the transaction and the corresponding commit message to the ledger, executes it, and updates the state. This ensures that all replicas execute requests in the same order as required to ensure safety. The primary also sends a reply $\langle \text{REPLY}, t_c, c, o\rangle_{\sigma_{\pi(p_i)}}$ to client c where t_c is the timestamp of the corresponding request and o is the execution result. If the client does not receive reply soon enough, it multicasts the request to all nodes within the cluster. If the request has already been processed, the nodes simply send the execution result back to the client. Otherwise, if the node is

Algorithm 2 Dealing with Conflicting ACCEPT Messages

```
****** The configuration is the same as Algorithm 1 ******
1: if accept messages of cluster $p_j$ not matching and $r == \pi(p_i)$
2:     multicast ⟨SUPER-PROPOSE, $h_i, d, m$⟩ to $\pi(p_j)$
3: upon receiving ⟨SUPER-PROPOSE, $h_i, d, m$⟩ from $\pi(p_i)$ and $r == \pi(p_j)$
4:     multicast ⟨SUPER-ACCEPT, $h_i$, $h_j$, $d$, $r$⟩ to $\pi(p_i)$ and all nodes of $p_j$
5: upon receiving ⟨SUPER-ACCEPT, $h_i, h_j, d, \pi(p_j)$⟩ from $\pi(p_j)$ and $r \in p_j$
6:     send ⟨SUPER-ACCEPT, $h_i$, $h_j$, $d$, $r$⟩ to $\pi(p_i)$
```

not the primary, it relays the request to the primary. If the primary does not multicast the request to the nodes of the cluster, it will eventually be suspected to be faulty by the nodes.

3.3 Dealing with Conflicting Messages

In the presented consensus protocol and after multicasting a propose message, the primary might not receive a quorum of *matching* accept messages from $f+1$ nodes of every involved cluster after a predefined time τ_a because the primary nodes of different clusters might multicast their propose messages in parallel, hence, different nodes in an overlapping cluster might receive these *conflicting messages* in different orders and assign them inconsistent sequence numbers in their corresponding accept messages. A special case of conflicting messages is when there is more than one cluster in the intersection of conflicting propose messages, hence, to ensure consistency, as explained earlier, nodes of overlapping clusters do not send accept messages for later transactions before committing the earlier ones, thus, the system might face a *deadlock* situation. We propose two techniques, the first for the general case of conflicting messages and the second to deal specifically with deadlocks.

Conflicting Messages. Algorithm 2 demonstrates an optimization to deal with conflicting messages. In case of non-matching accept messages, as indicated in lines 1-2 of Algorithm 2, the initiator primary $\pi(p_i)$ needs to re-initiate the request in *only* the *conflicting clusters*, i.e., clusters that have not sent $f + 1$ matching accept messages to the initiator primary. However, to preventing any further conflicts, the initiator primary $\pi(p_i)$ multicasts a super-propose message with the same structure as propose messages to *only* the primary nodes of the conflicting clusters. Once the initiator primary $\pi(p_i)$ sends a super-propose message for transaction m to the primary node of a cluster, $\pi(p_i)$ does not accept any further accept messages for transaction m from that cluster. As shown in lines 3-4, the primary node of each conflicting cluster then assigns a sequence number and multicasts a super-accept message (with the same structure as accept messages) to the nodes of its cluster and also the initiator primary $\pi(p_i)$. Upon receiving a super-accept message from the primary of its cluster, as presented in lines 5-6, each node logs the message and sends a super-accept message with the same sequence number to $\pi(p_i)$. Nodes also remove the previous sent accept messages for m from their logs. Once $\pi(p_i)$ receives matching super-accept messages from $f+1$ nodes of *every* conflicting cluster, it returns to its normal operation, as presented in lines 11-13 of Algorithm 1, and multicasts commit messages.

Well-designed sharded systems attempt to reduce cross-shard transactions, distribute the load on geographically distributed workloads, and balance heavy and light workloads. Nevertheless, SharPer might still incur heavy workloads with a high percentage of cross-shard transactions where the probability of receiving conflicting accept messages is high. In such circumstances, instead of multicasting propose messages, waiting for probably conflicting accept messages and then re-initiating the transaction by multicasting super-propose messages, the initiator primary can initially multicast super-propose messages to the primary nodes of the involved clusters. In this way, since the primary of each cluster assigns all sequence numbers for both intra-shard and cross-shard transactions, no conflicts will occur. This solution, however, comes with an extra intra-cluster message passing. Depending on the type of workload and percentage of cross-shard transactions, SharPer can dynamically switch between these two techniques to deal efficiently with conflicting messages.

Deadlock Situations. If different overlapping clusters receive propose messages for concurrent cross-shard transactions in conflicting orders, the system might face a deadlock situation. In particular, if two clusters p_1 and p_2 receive propose messages for cross-shard transactions m and m' in conflicting orders, e.g., p_1 receives m before m' and p_2 receives m' before m, to ensure the consistency property (as explained in Algorithm 1, line 9), clusters do not process the second transaction before committing the first one, i.e., p_1 waits for the commit message of m and p_2 waits for the commit message of m'. However, since committing a cross-shard transaction requires $f + 1$ accept messages from every involved cluster, neither of m and m' can be committed (i.e., deadlock situation). In such a situation, similar to conflicting messages, the initiator primary nodes of deadlocked transactions multicast super-propose messages to the primary node of clusters that are involved in the deadlocked transactions. All involved clusters must then reach a unique order between deadlocked transactions and based on that undo their sent accept messages if needed. Note that at that point, primary nodes do not add any new transaction m'' into the deadlock situation before all existing transactions get committed to preventing any possible starvation. We explain the technique in two cases. First, if both m and m' have been initiated by the same cluster, the primary nodes of other clusters, which are involved in both m and m', can detect the correct order by comparing the sequence numbers of m and m' and in case a node has already sent an accept message for the request with the higher sequence number, it needs to undo its sent accept message by sending a super-accept message with a different sequence number. The primary node multicasts the super-accept message to the nodes of its cluster, hence, they also send the super-accept message to the initiator primary (i.e., to prevent any further conflict the primary assign sequence numbers to deadlocked transactions). Nodes as well as the initiator primary also remove the previous sent accept messages from their logs. Second, when m and m' have been initiated by different clusters, e.g., m is initiated by p_3 where p_1, p_2, and p_3 are involved in m and m' is initiated by p_4 where p_1, p_2, and p_4 are involved in m'. In such a situation and to determine a unique order, transactions m and m' are ordered based on the id of their initiator clusters. As a result, if a node has already sent an accept message for the request with the higher initiator cluster id, it sends an super-accept message to the initiator primary with a different sequence number. Both the nodes and the initiator primary also remove the previous sent accept messages from their logs. Note that this deadlock resolution technique can easily be generalized for situations with more overlapping clusters and more conflicting messages. In particular, while in deadlocks of length greater than two, clusters have no global knowledge of all deadlocked transactions, the partial knowledge of each cluster does not violate the global ordering of the deadlocked transactions, i.e., transactions will be processed in the same order, although some transactions might incur more waiting time.

3.4 Primary Failure Handling

The goal of the primary failure handling routine is to improve liveness by allowing the system to make progress when a primary node fails. The routine is triggered by timeout. When node r of some cluster p_j receives a valid propose message from a primary for either an intra-shard or a cross-shard transaction, it starts a timer that expires after some predefined time τ_f. Time τ_f for cross-shard transactions is longer than τ_f of intra-shard transactions because processing cross-shard transactions usually takes more time. Moreover, time τ_f for cross-shard transactions is much longer than τ_a (i.e., the timeout for resolving conflicting messages) to allow primary nodes to resolve conflicts and deal with deadlock situations. If the timer has expired and node r has not committed the request, the node suspects that the primary might be faulty. We need to address three cases. First, a cross-shard transaction where node r (of cluster p_j) and the initiator primary $\pi(p_i)$ which is suspected to be faulty, i.e., it has not sent super-propose (if accept messages are conflicting) or commit messages, are in different clusters (i.e., $i \neq j$). Second, a cross-shard transaction where node r is not in the initiator cluster (i.e., $i \neq j$), however, $\pi(p_j)$ is suspected to be faulty, i.e., it has not sent super-accept messages (if accept messages are conflicting or if the system uses the optimization discussed for heavy workloads), and third, an intra- or a cross-shard transaction where node r and the initiator primary which is suspected to be faulty, i.e., it has not sent propose, super-propose, super-accept, or commit messages, are in the same cluster (i.e., $i = j$).

In the first case, node r multicasts a $\langle$COMMIT-QUERY, $h_i, h_j, d, r\rangle$ message to every node of the initiator cluster p_i where h_i and h_j are the sequence numbers assigned to the transaction by clusters p_i and p_j (in the corresponding propose and accept (or super-accept) messages). There are indeed three possible situations: (1) The request has already been committed, thus, the corresponding commit message will be sent back to node r by the initiator primary, (2) The initiator primary is still waiting for super-accept messages of some involved cluster, and (3) The initiator primary itself has failed, hence, the nodes of p_i need to elect a new primary. The nodes of p_i can easily distinguish between cases 2 and 3 (waited or failed initiator primary) by exchanging messages and electing a new primary only if the primary has failed. The primary failure handling routine is performed by the nodes of the same cluster as the faulty primary.

In the second case, when $\pi(p_j)$ has failed, if node r has not detected that $\pi(p_j)$ is failed, similar to the first case, r multicasts a commit-query message to every node of the initiator cluster p_i (assuming that the initiator primary has failed). Upon receiving a commit-query message, the initiator primary multicasts super-propose messages to every node of p_j, hence, nodes of p_j suspect that $\pi(p_j)$ is faulty. Note that, this case is very unlikely to happen because, on one hand, node r usually is able to detect that the $\pi(p_j)$ is faulty (from intra-shard messages) and on the other hand, the timer τ_a of $\pi(p_i)$ will expire much earlier than the timer τ_f of node r, hence, $\pi(p_i)$ will send super-propose messages to nodes of cluster p_j earlier (the first time τ_a expires, $\pi(p_i)$ multicasts super-propose messages to the primary nodes of the conflicting clusters, the second time, it multicasts super-propose to every node. For heavy workloads, however, it multicasts super-propose to every node from the beginning).

Third, when node r and the initiator primary are in the same cluster, node r initiates the leader election phase of Paxos [31] to elect the new primary, and the new primary handles all the uncommitted intra- and cross-shard transactions, and takes care of new client requests. Due to space limitation, the detailed explanation is omitted and is provided in the extended version of the paper [5].

3.5 Correctness Arguments

Consensus protocols have to satisfy *safety* and *liveness*. Safety means all correct nodes receive the same requests in the same order whereas liveness means all correct requests are eventually ordered. In this section, the safety (agreement, validity, and consistency) and liveness (termination) properties of SharPer in the presence of crash-only nodes are demonstrated. Since intra-shard transactions follow Paxos, we mainly focus on cross-shard transactions.

LEMMA 3.1. (*Agreement*) If node r commits request m with sequence number h, no other correct node commits request m' ($m \neq m'$) with the same sequence number h.

PROOF. Let m and m' ($m \neq m'$) be two committed requests with sequence numbers $h = [h_i, h_j, h_k, ...]$ and $h' = [h'_k, h'_l, h'_m, ..]$ respectively. Committing a request requires matching accept (or super-accept) messages from $f + 1$ different nodes of *every* involved cluster. Therefore, if the involved clusters of m and m' intersect in cluster p_k, at least $f+1$ nodes of cluster p_k have sent matching accept (or super-accept) messages for m, and similarly, at least $f + 1$ nodes of cluster p_k have sent matching accept (or super-accept) messages for m'. Since each cluster includes $2f + 1$ nodes and nodes are non-malicious, $h_k \neq h'_k$. Note that the same proof logic applies in special cases where m or m' is an intra-shard transaction (i.e., $h = h_k$ or $h' = h'_k$).

If the primary fails, since each committed request has been replicated on a quorum Q_1 of $f + 1$ nodes and to be elected primary, agreement from a quorum Q_2 of $f + 1$ nodes is needed, Q_1 and Q_2 must intersect in at least one node that is aware of the latest committed request. Hence, SharPer guarantees the agreement property for both intra-shard as well as cross-shard transactions. □

LEMMA 3.2. (*Validity*) If a correct node r commits m, then m must have been proposed by some correct node π.

PROOF. Since crash-only nodes do not send fictitious messages, validity is ensured. □

LEMMA 3.3. (*Consistency*) Let P_μ denote the set of involved clusters for a request μ. For any two committed requests m and m' and any two nodes r_1 and r_2 such that $r_1 \in p_i$, $r_2 \in p_j$, and $\{p_i, p_j\} \in P_m \cap P_{m'}$, if m is committed before m' in r_1, then m is committed before m' in r_2.

PROOF. As shown in Section 3.2, once node r_1 of some cluster p_i receives a propose message for some cross-shard transaction m, if the node is involved in another uncommitted cross-shard transaction m' where $|P_m \cap P_{m'}| > 1$, i.e., some other cluster p_j is also involved in both transactions, node r_1 does not send an accept message for transaction m before m' gets committed. Since committing request m requires $f + 1$ accept messages from *every* involved cluster, m

cannot be committed until m' is committed. As a result, the order of committing messages is the same in all involved nodes. □

Note that ensuring consistency might result in deadlock situations which can be resolved as explained in Section 3.3.

PROPERTY 3.4. (*Termination*) A request m issued by a correct client eventually completes.

SharPer, as mentioned earlier and due to the FLP impossibility result [20], guarantees liveness *only* during periods of synchrony. To show that a request issued by a correct client eventually completes, we need to address three scenarios. First, if the primary is non-faulty and accept messages are non-conflicting. As shown in Algorithm 1, the protocol ensures that a correct client receives a reply from the primary. Second, if a non-faulty primary has multicast a propose message but not received matching accept messages from $f + 1$ nodes of every involved cluster. As explained in Sections 3.3, the initiator primary re-initiates the transaction by multicasting super-propose messages to only the primary nodes of the involved clusters. Since the primary node of each cluster assigns the sequence number (in its super-accept message), super-accept messages that are received from each cluster must match, thus increasing the chances of termination. In case of a deadlock situation, i.e., different clusters receive transactions in conflicting orders, upon receiving a super-propose message from the initiator primary, a unique order is determined and nodes within different clusters might need to send a new super-accept message. Third, if the (initiator) primary fails, as explained in Sections 3.4, nodes involved in an uncommitted transaction (initiated by the faulty primary) detect its failure (using timeouts) resulting in triggering the failure handling routine.

4 Consensus with Byzantine Nodes

In this section, intra- and cross-shard consensus in the presence of Byzantine nodes are presented followed by the primary failure handling routine. Then, the correctness of SharPer is proven.

4.1 Intra-shard consensus

Most Byzantine fault-tolerant protocols, e.g., PBFT [11], require $3f+1$ nodes to guarantee safety in the presence of at most f *malicious* nodes. In PBFT, the replicas move through a succession of configurations called *views* [18][19] where in each view, one replica, called *the primary*, initiates the protocol and the others are *backups*. SharPer uses PBFT to establish consensus on the order of intra-shard transactions. During normal case execution, a client c requests an intra-shard transaction by sending message $m = \langle \text{REQUEST}, op, t_c, c\rangle_{\sigma_c}$ to the primary where op is the requested intra-shard transaction, and t_c is a timestamp used to ensure exactly-once semantics (prevent replay attacks). When the primary receives a valid request from an authorized client, it initiates the consensus protocol by assigning a sequence number and multicasting a *signed* propose (called pre-prepare in PBFT) message including the requested transaction to all nodes within the cluster. Note that in the presence of Byzantine nodes and to provide validity, all messages sent by all nodes are signed. Once a node receives a valid propose message from the primary, it multicasts an accept (prepare) message to every node within the cluster. Each node then waits for $2f$ valid accept messages from different nodes (including itself) that match the propose message and then multicasts a commit message to all nodes of the cluster. Once a node receives $2f$ valid commit messages from different nodes that match its own commit message, it appends the transaction as well as all $2f + 1$ commit messages to the ledger (to ensure immutability), executes the transaction, updates the state, and sends a reply to the client. Finally, the client waits for $f + 1$ valid matching responses from different replicas to ensure at least one correct replica executed its request.

Algorithm 3 Cross-shard Consensus with Byzantine Nodes

```
1:  init():
2:      $r$ := $node\_id$
3:      $p_i$ := the cluster that initiates the consensus
4:      $\pi(p)$ := the primary node of cluster $p$
5:      $P$ := set of involved clusters
6:  upon receiving valid transaction $m$ and $(r == \pi(p_i))$
7:      multicast $\langle\langle \text{PROPOSE}, h_i, d\rangle_{\sigma_{\pi(p_i)}}, m\rangle$ to $P$
8:  upon receiving valid $\langle\langle \text{PROPOSE}, h_i, d\rangle_{\sigma_{\pi(p_i)}}, m\rangle$ from $\pi(p_i)$
9:      if $r$ is not involved in any uncommitted request $m'$ where $m$ and $m'$ intersect
    in some other cluster $p_k$
10:         multicast $\langle \text{ACCEPT}, h_i, h_j, d, r\rangle_{\sigma_r}$ to $P$
11: upon receiving valid matching $\langle \text{ACCEPT}, h_i, h_j, d, r\rangle_{\sigma_r}$ from $2f+1$ different nodes
    of every cluster $p_j$ in $P$
12:     multicast $\langle \text{COMMIT}, [h_i, h_j, ..., h_k], d, r\rangle_{\sigma_r}$ to $P$
13: upon receiving valid $\langle \text{COMMIT}, [h_i, h_j, ..., h_k], d, r\rangle_{\sigma_r}$ from $2f + 1$ nodes of every
    cluster in $P$
14:     append the transaction and commit messages to the ledger
```

4.2 Byzantine Cross-shard Consensus

In the presence of malicious nodes, a Byzantine fault-tolerant protocol is needed where for each cross-shard transaction, similar to the crash-only case, agreement from all involved clusters is needed. Unlike in the case of crash failure where the quorum size is $f + 1$, in consensus with Byzantine nodes, the quorum size is $2f + 1$. In addition and due to the potential malicious behavior of the primary node, all non-faulty nodes of every involved cluster multicast both accept and commit messages to each other.

The normal case operation (i.e., when the primary is non-faulty) for SharPer to process a cross-shard transaction in the presence of Byzantine nodes is presented in Algorithm 3. Similar to Algorithm 1 and as shown in lines 1-5, p_i is the initiator cluster, P is the set of involved clusters, and $\pi(p)$ indicates the primary node of cluster p.

Once the initiator primary $\pi(p_i)$ receives a valid signed cross-shard request from an authorized client, as presented in lines 6-7, $\pi(p_i)$ assigns sequence number h_i to the request and multicasts a propose message including sequence number h_i and digest d of the request to all nodes of every involved cluster. Requests are piggybacked in propose messages to keep propose messages small.

Upon receiving a propose message for a request m, node r of an involved cluster p_j, as indicated in lines 8-10, validates the request, signature and message digest. If the node belongs to the initiator cluster $(i = j)$, it also checks h_i to be *valid*, i.e., within a predefined range to prevent a malicious primary from exhausting the space of sequence numbers by choosing a very large value [11]. Furthermore, if the node is currently involved in an uncommitted cross-shard request m' where the involved clusters of two requests m and m' overlap in some other cluster as well, as explained in the crash-only case, the node does not process the new request m (only buffers m) before the earlier request m' is processed. This is needed to ensure concurrent requests are committed in the same order on overlapping clusters (consistency property). The node then multicasts an accept message including the corresponding sequence number h_j (that represents the order of m in cluster p_j) as well as the digest $d = D(m)$ to *every* node of *all* involved clusters.

Each node waits for valid accept messages with matching sequence numbers from $2f+1$ nodes of *every* involved cluster with h_i, and d that match the propose message which is sent by initiator primary $\pi(p_i)$. If a node receives accept messages without receiving a propose message, the node contacts the primary node (or its neighbors to reduce the load on the primary) to get the propose message. We define the predicate accepted-local$_{p_j}(m, h_i, h_j, r)$ to be true if and only if node r has received request m, a propose for m with sequence number h_i from the initiator cluster p_i and $2f + 1$ accept messages from different nodes of an involved cluster p_j that match the propose message. The predicate accepted(m, h, r) where $h = [h_i, h_j, ..., h_k]$ is then true on node r if and only if accepted-local$_{p_j}$ is true for *every* involved cluster p_j in cross-shard transaction m. The order of sequence numbers in the predicate is an ascending order determined by their cluster ids. Here, since nodes might behave maliciously, each cluster includes $3f + 1$ nodes and $2f + 1$ matching messages from *every* involved cluster for each step of the protocol are needed. The propose and accept phases of the algorithm basically guarantee that non-faulty nodes agree on a total order for the transactions. When accepted(m, h, r) becomes true, as presented in lines 11-12, node r multicasts a commit message $\langle \text{COMMIT}, h, d, r\rangle_{\sigma_r}$ to every node of all involved clusters.

Finally, as shown in lines 13-14, each node waits for valid matching commit messages from $2f + 1$ nodes of *every* involved clusters that match its commit message. Predicate committed-local$_{p_j}(m, h, r)$ is defined to be true on node r if and only if accepted (m, h, r) is true and node r has accepted $2f + 1$ valid matching commit messages from different nodes of cluster p_j that match the propose message for cross-shard request m. Predicate committed(m, h, r) is then true on node r if and only if committed-local$_{p_j}$ is true for *every* involved cluster p_j in request m. The committed predicate indeed shows that at least $f + 1$ nodes of each involved cluster have multicast valid commit messages. When the committed predicate becomes true, the node considers the transaction as committed. If the node has executed all transactions with lower sequence numbers than h_j, it appends the transaction and $2f + 1$ commit messages to the ledger, executes the transaction, updates the state, and sends a $\langle \text{REPLY}, t_c, c, o, r\rangle_{\sigma_r}$ message to client c where t_c is the timestamp of the corresponding request and r is the execution result. The client waits for $f + 1$ valid matching responses from different replicas to ensure at least one correct replica executed its request. If the client does not receive reply messages soon enough, it multicasts the request to all nodes within the cluster. If the request has already been processed, the nodes simply re-send the reply message to the client (nodes remember the last reply message they sent to each client). Otherwise, if the node is not the primary, it relays the request to the primary. If the primary does not multicast the request to the nodes, it will eventually be suspected to be faulty by nodes to cause a primary failure handling routine.

4.3 Dealing with Conflicting Messages

In the consensus protocol with Byzantine nodes, similar to the crash-only case, a quorum of $2f + 1$ *matching* accept messages from every cluster might not be received due to conflicting propose messages coming from different primary nodes in parallel. We first address conflicting messages and then discuss deadlock situation, a special case of conflicting messages where there is more than one cluster in the intersection of conflicting propose messages. The $\langle \text{ACCEPT}, h_i, h_j, d, r\rangle_{\sigma_{\pi(p_i)}}$ messages might be non-matching for two reasons. First, the initiator primary $\pi(p_i)$ is malicious and sends inconsistent messages, i.e., assigns inconsistent sequence numbers, to different nodes, hence, there is no quorum of $2f + 1$ nodes from a cluster with matching sequence number h_i for the same request. Note that a malicious initiator primary might also assign invalid digest d or sign its message incorrectly, however, it will be easily detected by all nodes as an *invalid* message. The only malicious behavior that is not detected by nodes alone and requires communication among them (i.e., sending accept messages) is when the initiator primary assigns inconsistent sequence numbers to the same request. Second, when different nodes of the same cluster, similar to the crash-only case, assign inconsistent sequence number h_j. We address the first case, in the primary failure handling routine. In the second case, as presented in lines 1-2 of Algorithm 4, the primary node of each conflicting cluster p_j, i.e., a cluster where at least $2f + 1$ of accept messages have matching h_i and d (to ensure that the initiator primary is non-faulty) but less than $2f+1$ of accept messages have matching h_j, multicasts a super-accept message (with the same structure as accept messages) to the nodes of its own cluster after a predefined time τ_a. Note that this is in contrast to the crash-only case where only the initiator primary multicasts super-accept messages. Once a node receives a super-accept message for some cross-shard transaction m from the primary node of its cluster, as shown in lines 3-5, it first validates the message, the digest, and its sequence number to be within a predefined range. Node also checks the received accept messages from nodes of its cluster to ensure that the initiator primary is non-malicious, i.e., less than f accept messages have non-matching h_i. If (1) the node has received less than $2f + 1$ matching accept messages, i.e., messages with matching h_j, for transaction m from the nodes of its cluster or (2) the transaction is in a deadlock situation, hence, there is a need to undo accept messages by sending super-accept messages, the node (including the primary node) multicasts a super-accept message to all nodes of every involved cluster.

Algorithm 4 Dealing with Conflicting ACCEPT Messages

```
   ******The configuration is the same as Algorithm 3******
1: if accept messages of cluster $p_j$ not matching and ($r == \pi(p_j)$)
2:     multicast $\langle$SUPER-ACCEPT, $h_i$, $h_j$, $d$, $r\rangle_{\sigma_r}$ to nodes of $p_j$
3: upon receiving $\langle$SUPER-ACCEPT, $h_i$, $h_j$, $d$, $\pi(p_j)\rangle_{\sigma_{\pi(p_j)}}$ and $r \in p_j$
4:     if (less than $f$ accept messages have non-matching $h_i$) and ( (less than $2f$+1
       matching accept from $p_j$ for $m$ are logged) or (the transaction is deadlocked))
5:         multicast $\langle$SUPER-ACCEPT, $h_i$, $h_j$, $d$, $r\rangle_{\sigma_r}$ to $P$
```

In heavy workloads with a high percentage of cross-shard transactions where the probability of receiving conflicting accept messages is high, similar to the crash-only case, the initiator primary initially multicasts super-propose messages to all nodes of other involved clusters. The primary of each involved cluster then multicasts a super-accept message (with piggybacked super-propose message received from the initiator primary) to the nodes of its cluster.

To address deadlock situations, i.e., where overlapping clusters receive propose messages in conflicting orders, similar to the crash-only case, the initiator primary multicasts super-propose messages to the primary nodes of the overlapping clusters. The overlapping clusters then reach a unique order among concurrent transactions using either the sequence number of transactions (if concurrent transactions are initiated by the same cluster) or the id of the initiator clusters (if transactions are initiated by different clusters).

4.4 Primary Failure Handling

The primary failure handling routine, similar to the crash-only case, is triggered by timeout. If the timer of some node r of cluster p_j expires (after some predefined time τ_f) node r suspects that the primary might be faulty. There are three cases. First, a cross-shard transaction where node r and the initiator primary $\pi(p_i)$, which is suspected to be faulty, i.e., it has not sent valid propose or super-propose messages, are in *different* clusters (i.e., $i \neq j$). Second, a cross-shard transaction where node r is not in initiator cluster p_i, however, $\pi(p_j)$ is suspected to be faulty, i.e., it has not sent valid super-accept messages. Third, an intra- or a cross-shard transaction where node r and the initiator primary, which is suspected to be faulty, i.e., it has not sent valid propose, super-propose, or super-accept messages, are in the *same* cluster (i.e., $i = j$).

In the first case, if the propose message has an incorrect digest or signature, node r discards it. However, if propose messages are valid but more than f of accept messages that node r receives from nodes of an involved cluster have non-matching h_i, then initiator primary $\pi(p_i)$ is malicious. Therefore, node r multicasts an $\langle \text{ACCEPT-QUERY}, h_i, d, r\rangle_{\sigma_{\pi(r)}}$ messages to every node of initiator cluster p_i. Note that, node r still processes all intra-shard transactions as well as all transactions coming from all other clusters. If a node receives accept-query messages from $2f + 1$ different nodes of another cluster with matching d, the node suspects that the primary of its cluster is faulty and initiates the primary failure handling routine (explained later). Second, when primary $\pi(p_j)$ is malicious and multicasts super-accept messages with (consistent sequence number h_i but) inconsistent sequence numbers h_j to the nodes of its cluster. In this case, node r will receive inconsistent super-accept messages from different nodes of p_j, suspects that $\pi(p_j)$ is faulty and initiates the primary failure handling routine. Note that, this case happens when either accept messages are conflicting or the optimization presented for heavy workloads is used. Third, when node r and the faulty primary are in the same cluster, node r initiates the primary failure handling routine by multicasting a failure-query message including all received valid accept, accept-query, and commit messages for all intra-shard as well as cross-shard transactions to every node of the cluster. To decrease the size of failure-query messages, SharPer uses checkpoints as PBFT [11], i.e., each node sends the last stable checkpoint that it knows, proof of its correctness, and messages with a sequence number higher than the checkpoint sequence number. An accept-query message is valid if it is received from at least $2f + 1$ different nodes of a cluster. Upon receiving $2f$ failure-query messages, the next primary (determined in a round-robin manner based on node ids) handles the uncommitted transactions by multicasting a new-primary message including $2f + 1$ failure-query messages and a propose message for each uncommitted request (either intra-shard or cross-shard) to every node within the cluster. For uncommitted cross-shard requests where the cluster has initiated the requests, the new primary multicasts a new-primary message including $2f + 1$ failure-query messages and the related propose messages to every node of all involved clusters. If other clusters have already accepted the request, they simply send back their accept (or super-accept) messages. Once node r multicasts a failure-query message, it starts a timer that expires after some time τ_v. If the timer expires before it receives a valid new-primary message, it starts the routine again. In the worst case, the system might incur f consecutive faulty primary nodes.

4.5 Correctness Arguments

We demonstrate how SharPer satisfies the safety and liveness properties in the presence of Byzantine nodes.

LEMMA 4.1. (*Agreement*) If node r commits request m with sequence number h, no other correct node commits request m' ($m \neq m'$) with the same sequence number h.

PROOF. The propose and accept phases of the Byzantine cross-shard consensus protocol guarantee that correct nodes agree on a total order of all requests. Indeed, if the accepted(m, h, r) predicate where $h = [h_i, h_j, ..., h_k]$ is true, then accepted(m', h, q) is false for any non-faulty node q (including $r = q$) and any m' such that $m \neq m'$. This is true because (m, h, r) implies that accepted-local$_{p_j}(m, h_i, h_j)$ is true for each involved cluster p_j and since each cluster include $3f + 1$ nodes, at least $2f + 1$ nodes within the cluster (from which at least $f + 1$ nodes are non-faulty) have sent accept (or propose) messages for request m with sequence number h_j. As a result, for accepted(m', h, q) to be true, at least one of those non-faulty nodes needs to have sent two conflicting accept messages with the same sequence number but different message digest. This condition guarantees that first, a malicious primary cannot violate safety and second, at most one of the concurrent *conflicting* transactions can collect the required number of messages ($2f + 1$) from each overlapping cluster.

The primary failure handling routine of SharPer guarantees that the non-faulty nodes of any cluster p_j agree on the sequence number of requests that are committed-local at different nodes. The committed-local$_{p_j}$ predicate becomes correct on node r if r has received a quorum Q_1 of matching commit messages from $2f + 1$ nodes of cluster p_j. To change the primary node of cluster p_j, a quorum Q_2 of $2f + 1$ valid failure-query messages is needed. Since there are $3f + 1$ nodes in each cluster, Q_1 and Q_2 intersect in at least one correct replica, thus if a request is accepted by the previous primary node, it is propagated to subsequent primary nodes. □

LEMMA 4.2. (*Validity*) If a correct node r commits m, then m must have been proposed by some correct node π.

PROOF. In the presence of Byzantine nodes, validity is guaranteed mainly based on standard cryptographic assumptions about collision-resistant hashes, encryption, and signatures which the adversary cannot subvert (as explained in Section 2). Since the request as well as all messages are signed and either the request or its digest is included in each message (to prevent changes and alterations to any part of the message), and in each step, $2f + 1$ matching messages (from each cluster) are required, if a request is committed, the same request must have been proposed earlier. □

LEMMA 4.3. (*Consistency*) Let P_μ denote the set of involved clusters for a request μ. For any two committed requests m and m' and any two nodes r_1 and r_2 such that $r_1 \in p_i$, $r_2 \in p_j$, and $\{p_i, p_j\} \in P_m \cap P_{m'}$, if m is committed before m' in r_1, then m is committed before m' in r_2.

Proof. Consistency is guaranteed similar to crash-only nodes (lemma 3.3) except that committing request m requires $2f + 1$ matching commit messages (out of $3f + 1$) from each cluster. □

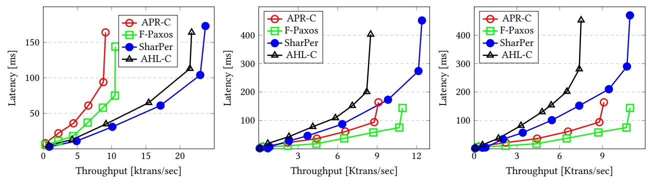
Property 4.4. (*Termination*) A request m issued by a correct client eventually completes.

To provide termination during periods of synchrony, similar to the crash-only case, several scenarios need to be addressed. If the primary is non-faulty and accept messages are non-conflicting, following Algorithm 3, request m completes. Next, if the primary is non-faulty, but more than f accept messages of some involved cluster p_j have inconsistent sequence number h_j, as explained in Section 4.3, $\pi(p_j)$ multicasts a super-accept message including a h_j to the nodes of its cluster. In case of a deadlock situation, i.e., where different clusters receive transactions in different orders, a unique order is determined by the primary of each cluster and clusters might need to send new super-accept messages. Finally, the primary failure handling routine (Section 4.4) handles primary failures in several cases where (1) an initiator primary multicasts incorrect propose or super-propose messages to other clusters (2) the primary of an involved cluster multicasts incorrect super-accept messages for a cross-shard transaction to nodes of its cluster, and (3) a primary node multicasts incorrect propose, super-propose, or super-accept messages to the nodes of its cluster.

Note that in the optimization explained in Section 4.3 for heavy workload where the initiator primary multicasts super-propose messages to the nodes of all other involved clusters, assigning an inconsistent sequence number, h_j will be detected in the accept phase. Furthermore, if either the initiator primary does not multicast the super-propose message to an involved primary (or cluster) or an involved primary does not multicast the super-accept message to the nodes of its cluster, since all nodes of all involved clusters multicast super-accept messages to each other, as long as nodes of one involved cluster multicast super-accept messages, other involved clusters will be informed (i.e. no liveness issue will happen). If a node receives valid super-accept messages from nodes of other clusters without receiving super-accept message from the primary of its cluster (and super-propose message from the initiator primary), it multicasts a query message to all nodes of the initiator cluster. The primary of an involved cluster multicasts the query message to the nodes of its cluster as well in case they received the message (since nodes of a cluster are in proximity, it reduce the latency of processing a request). If nodes of the initiator cluster receive such queries from $2f + 1$ nodes of an involved cluster for a request, they suspect that the initiator primary is faulty. If nodes of an involved cluster do not receive the super-accept message from their primary after some predefined time, they suspect that their primary is faulty. In the worst case, a faulty initiator primary node might continue to operate maliciously by not sending super-propose messages to the primary nodes of the involved clusters. However, in this case, the primary node of each involved cluster can obtain the actual request probably from a node in its cluster, since these nodes are in closer proximity, hence the safety and liveness are not affected.

5 Experimental Evaluations

In this section, we conduct several experiments to evaluate SharPer. In our implementation of SharPer, Algorithms 1 and 3 are followed in normal workloads (and Algorithms 2 and 4 in case of conflicts) in the presence of crash-only and Byzantine nodes. In heavy workloads, however, the optimization explained at the end of sections 3.3 and 4.3 has been used. SharPer is able to dynamically switch between these two different techniques depending on the workload. We have also deployed an accounting application on SharPer where clients initiate transactions to transfer assets between accounts in the same or different shards. The experiments were conducted on the Amazon EC2 platform. Each VM is a c4.2xlarge instance with 8 vCPUs and 15GB RAM, Intel Xeon E5-2666 v3 processor clocked at 3.50 GHz. When reporting throughput measurements, we use an increasing number of clients running on a single VM, until the end-to-end throughput is saturated, and state the throughput (x axis) and latency (y axis) just below saturation.



(a) 20% Cross-shard (b) 80% Cross-shard (c) 100% cross-shard

Figure 2: Cross-Shard Transactions with Crash-Only Nodes

5.1 Cross-Shard Transactions with Crash-Only Nodes

In the first set of experiments, we measure the performance of SharPer for workloads with different percentages of cross-shard transactions where nodes are crash-only. We compare SharPer with the two main approaches for exploiting the availability of extra resources: the active/passive replication technique and Fast Paxos [30]. In the active/passive replication technique, the protocol relies only on $2f+1$ active nodes to establish consensus and updates the passive replicas asynchronously whereas Fast Paxos use $3f+1$ replicas instead of $2f+1$ to reduce one phase of communication. We implemented two permissioned blockchain systems referred to as *APR-C* and *FPaxos* where their consensus protocols follow the active/passive replication and Fast Paxos designs respectively. In addition to SharPer and these two systems, we also implemented a modified version of the sharded permissioned blockchain system AHL [16]. AHL has two novel aspects: first, its intra-shard consensus protocol that uses trusted hardware to restrict the malicious behavior of nodes, and second, its cross-shard consensus protocol where a reference committee uses 2PC to order the transactions. Since the emphasis of the experiments is on cross-shard transactions, we implemented a modified version of AHL, called AHL-C where the intra-shard transactions are processed similar to SharPer, however, the cross-shard transactions are performed similar to AHL [16] where the classic two-phase commit (2PC) runs in two communication phases (prepare and commit) between the reference committee and involved clusters.

We consider a network with 12 nodes (15 nodes in AHL-C). In SharPer and AHL-C, the nodes are divided into four clusters where each cluster consists of 3 nodes and uses Paxos with $f = 1$ to establish consensus. AHL-C includes a reference committee of 3 crash-only nodes as well. Each cluster further maintains a data shard of 10000 records (clients). In APR-C, 3 nodes are used as the active replicas and the execution results are sent to the remaining 9 nodes whereas FPaxos uses 4 nodes $(3f + 1)$ to establish consensus.

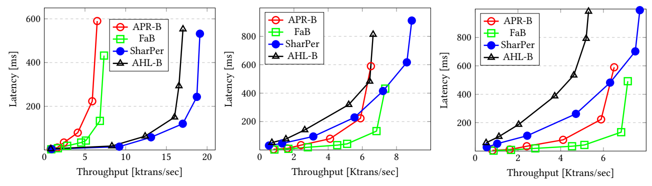
We consider four different workloads with (1) no cross-shard, (2) 20% cross-shard, (3) 80% cross-shard, and (4) 100% cross-shard transactions. We also assume that two (randomly chosen) shards are involved in each cross-shard transaction. Note that since APR-C and FPaxos do not use sharding, the percentage of cross-shard transactions does not affect their performance. The load is also equally distributed among all the nodes.

When there are no cross-shard transactions, SharPer is able to process 35230 transactions with 91 ms latency before the end-to-end throughput is saturated where every 5 ms, ∼ 45 requests from different clients are sent to each cluster. Note that in this setting, since there are no cross-shard transactions, each cluster orders and executes its transactions independently, thus the throughput of the entire system will increase linearly with the increasing number of clusters. Since for intra-shard transactions, AHL-C uses the same technique as SharPer, its results are identical to SharPer. APR-C and FPaxos are also able to process 8800 and 10700 transactions with 95 ms and 75 ms latency respectively (as can be seen in Figure 2(a)). Since FPaxos establishes consensus in less number of phases, it has better performance than APR-C. However, they both have much lower throughput in comparison to SharPer (25% and 33% of SharPer at 60 ms latency). The results mainly demonstrate the effectiveness of employing the sharding technique in blockchains.

By increasing the percentage of cross-shard transactions to 20% (Figure 2(a)), the throughput is reduced due to the overhead of cross-shard transactions. In this setting, SharPer is still able to process 23000 transactions with 100 ms latency whereas AHL-C processes 21000 transactions at the same latency. This is expected because first, SharPer, in contrast to AHL-C, is able to process non-overlapping cross-shard transactions in parallel, and second, the cross-shard protocol of SharPer involves less number of communication phases. As mentioned before, since the sharding technique is not utilized by APR-C and FPaxos, the percentage of cross-shard transactions does not affect their performance.

Similarly, increasing the percentage of cross-shard transactions to 80% (Figure 2(b)) and finally, 100% (Figure 2(c)) reduces the peak throughput of SharPer to 12300 and 10500, respectively. Note that by increasing the percentage of cross-shard transactions, SharPer still shows much better performance compared to AHL-C (44% better in their peak throughput with 100% cross-shard transactions) because SharPer is still able to process non-overlapping cross-transactions in parallel and also needs less number of communication phases. In these two scenarios, since APR-C and FPaxos order the transactions using only three ($2f+1$) and four ($3f+1$) nodes, their latency is lower than SharPer. Specially FPaxos processes transactions with significantly lower latency due to its fast consensus routine. However, since a large percentage of transactions is cross-shard, SharPer needs the participation of all involved clusters to order transactions and using sharding has no significant advantage. In fact, Figures 2(c) and 2(d) demonstrate that if sharding is not workload-aware the performance will be severely impacted.

To evaluate the impact of primary failure, we terminate the process of a primary node in the first two scenarios (0% and 20% cross-shard transactions). This failure and the failure handling routine reduce the throughput to 26000 (73.8%) and 17100 (74.3%) and the cluster was temporarily out of service for 18 and 23 ms respectively.



(a) 20% Cross-shard (b) 80% Cross-shard (c) 100% cross-shard

Figure 3: Cross-Shard Transactions with Byzantine Nodes

5.2 Cross-Shard Transactions with Byzantine Nodes

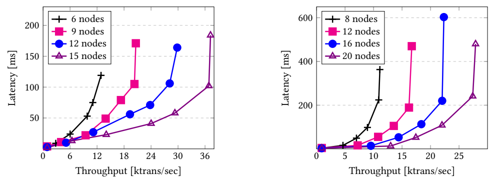
In the second set of experiments, we repeat the previous scenarios on networks with Byzantine nodes. Similar to the previous section, we implement four permissioned blockchain systems: (1) SharPer, (2) APR-B where its consensus protocol follows the active/passive replication technique on Byzantine nodes, (3) FaB where its consensus protocol follows Fast Byzantine consensus protocol [33] and uses $5f+1$ nodes (instead of $3f+1$) to establish consensus in two phases (instead of three as in PBFT), and (4) AHL-B where its intra-shard transactions are processed using PBFT (similar to SharPer) and its cross-shard transactions follow AHL [16].

We consider a network with 16 nodes. In SharPer and AHL-B, the nodes are partitioned into 4 clusters where each cluster consists of 4 nodes and uses PBFT protocol with $f=1$ to establish consensus on its transactions. In addition to these 16 nodes, in AHL-B, a reference committee of 4 Byzantine nodes is also considered. In APR-B, 4 nodes are used as the active replicas and finally, FaB uses 6 nodes ($5f+1$) to establish consensus. Similar to the previous case, since APR-B and FaB do not use sharding, the percentage of cross-shard transactions does not affect their performance.

With no cross-shard transactions, SharPer is able to process more than 25000 transactions with 200 ms latency. As before, since for intra-shard transactions, AHL-B uses the same technique as SharPer, the results of SharPer and AHL-B are identical. APR-B and FaB also process 5900 and 6800 transactions (23% and 27% of SharPer) with 220 ms and 130 ms latency respectively (as shown in Figure 3(a)). Note that since transactions are processed in two phases (instead of 3), FaB has lower latency in comparison to APR-B.

Increasing the percentage of cross-shard transactions to 20%, as shown in Figure 3(a), reduces the peak throughput of SharPer to 18700 (with 240 ms latency). In this scenario compared to AHL-B, SharPer processes 15% more transactions (at their respective peak throughput) because of the parallel ordering of cross-shard transactions and establishing cross-shard consensus in less number of phases. The peak throughput of SharPer is also 320% and 270% of the peak throughput of APR-B and FaB respectively.

With 80% cross-shard transactions, as can be seen in Figure 3(b), the peak throughput of SharPer reduces to 8600 which is still 34% higher than the peak throughput of AHL-B (6400) due to parallel processing of non-overlapping cross-shard transactions. Finally, when all transactions are cross-shard, as shown in Figure 3(c), SharPer is able to process 7500 transactions with 700 ms latency whereas AHL-B processes 5000 transactions (67% of SharPer) with the same latency. In the last two scenarios (80% and 100% cross-shard transactions), because of the high percentage of cross-shard transactions, using sharding techniques has no significant advantage (which again demonstrates the advantages of workload-aware sharding)



(a) Crash-Only Nodes (b) Byzantine Nodes

Figure 4: Increasing the Number of Nodes

and since APR-B and FaB rely on only four ($3f+1$) and six ($5f+1$) nodes to order transactions respectively, their latency is lower than SharPer. However, in SharPer, simultaneous processing of non-overlapping transactions results in improved throughput.

To evaluate the impact of primary failure, we terminate the process of a primary node in the first two scenarios (0% and 20% cross-shard transactions). This failure and the failure handling routine reduce the throughput to 18900 (75.6%) and 14200 (75.9%) and the cluster was temporarily out of service for 30 and 42 ms respectively.

5.3 Increasing the Number of Nodes

In the last set of experiments, we measure the performance of SharPer in networks with a different number of nodes. We evaluate SharPer in networks including 6, 9, 12, and 15 crash-only nodes as well as 8, 12, 16 and 20 Byzantine nodes (2, 3, 4 and 5 clusters). The workloads include 90% intra- and 10% cross-shard transactions (typical settings in partitioned databases [41] [40]).

As can be seen in Figure 4(a), when nodes follow the crash failure model, by increasing the number of nodes (clusters) the throughput of the system increases almost linearly. This is expected because 90% of transactions are intra-shard transactions and, as shown earlier, for intra-shard transactions, the throughput of the entire system will increase linearly with the increasing number of clusters. In addition, since cross-shard transactions access two clusters, by increasing the number of clusters, the chance of parallel processing of such transactions increases. As shown in Figure 4(a), in the settings with five clusters, SharPer processes 37000 transactions with 100 ms latency. When nodes follow the Byzantine failure model, as shown in Figure 4(b), SharPer demonstrates the similar behavior and processes more than 27000 transactions with 240 ms latency on a network with 5 clusters. These experiments demonstrate the scalability of SharPer as the number of clusters increases.

6 Related Work

A permissioned blockchain system, e.g., Tendermint [28], Quorum [12], Parblockchain [4], Fast Fabric [23], Fabric++ [39], FabricSharp [38] ResilientDB [24], and Caper [2], consists of a set of known, identified nodes that might not fully trust each other. Scalability is the ability of a blockchain system to process an increasing number of transactions by adding nodes to the system. Data sharding techniques are commonly used in globally distributed databases such as Amazon Dynamo [17] to improve scalability. In such systems, nodes are assumed to be crash-only and a centralized approach is used to process crash-shard transactions. SharPer, on the other hand, supports both crash-only and Byzantine nodes and introduces a decentralized approach to process crash-shard transactions.

Sharding techniques have been used in both permissionless, e.g., Elastico [32], OmniLedger [27], Monoxide [44], Ethereum 2 [1] and Rapidchain [45], and permissioned blockchain systems, e.g., multi-channel Fabric [7], AHL [16], Cosmos [21], and RSCoin [22] to improve scalability. Ethereum 2 [1], which as a permissionless blockchain is supposed to be used for the development of permissioned blockchain applications, consists of different shards (currently planned for 64 shards) where every shard block is processed by a randomly chosen set of validators. In multi-channel Fabric [6][7], channels (i.e., disjoint partitioned states of the full system) are introduced to shard the system [7]. Using channels, Fabric processes intra-shard transactions efficiently. However, processing cross-shard transactions, in contrast to SharPer, requires either the existence of a trusted channel among the participants or an atomic commit protocol (inspired by two-phase commit) [7]. Similarly, in Cosmos [21], interacting chains in any Inter-Blockchain Communication must be aware of the state of each other which requires establishing a bidirectional trusted channel between two blockchains. AHL [16] employs a trusted hardware (the technique that is presented in [13, 42, 43]) to restrict the malicious behavior of nodes which results in committees of $2f + 1$ nodes (instead of $3f + 1$). The system also relies on an extra set of nodes, called a reference committee, to process cross-shard transactions in a centralized manner using the classic two-phase commit (2PC) and two-phase locking (2PL) protocols. SharPer, in contrast to AHL, processes cross-application transactions in a decentralized manner. In addition, cross-shard transactions are ordered in only three communication phases. Furthermore, cross-shard transactions with non-overlapping committees can be processed simultaneously. Note that since the intra-shard consensus is pluggable, the trusted hardware technique can be employed to reduce the cluster size. Finally, Cerberus [25] eliminates the reference committee of AHL by adding one extra phase of communication across the involved clusters. Cerberus includes three protocols of which OCerberus is the most similar to SharPer. OCerberus focuses on malicious failures while SharPer supports both crash and malicious failures. Furthermore, OCerberus detects all faulty behavior unlike SharPer where, as discussed in Section 4.5, a faulty node might continue to operate maliciously in a restrictive manner. However, this malicious behavior has no ramifications on the correct execution and termination of transactions, i.e., safety and liveness.

7 Conclusion

In this paper, we proposed SharPer, a permissioned blockchain system to improve scalability. SharPer uses the sharing technique and provides deterministic safety guarantees in networks where more than a half (if nodes are crash-only) or two-thirds (if nodes are Byzantine) of the nodes of each cluster are non-faulty. Two decentralized flattened consensus protocols are introduced to order cross-shard transactions without relying on centralized entities or trusted participants. Furthermore, SharPer is able to process cross-shard transactions with non-overlapping clusters in parallel. Base on our experiments, in workloads with a low percentage of cross-shard transactions (typical settings), SharPer demonstrates better performance with both crash-only and Byzantine nodes in comparison to other approaches and the throughput of SharPer improves semi-linearly with the increasing number of clusters.

Acknowledgments

This work is funded by NSF grants CNS-1703560 and CNS-1815733.

References

[1] [n. d.]. The Beacon Chain Ethereum 2.0 explainer you need to read first. https://ethos.dev/beacon-chain/. ([n. d.]).

[2] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. 2019. CAPER: a cross-application permissioned blockchain. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1385–1398.

[3] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. 2019. On Sharding Permissioned Blockchains. In *Int. Conf. on Blockchain*. IEEE, 282–285.

[4] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. 2019. ParBlockchain: Leveraging Transaction Parallelism in Permissioned Blockchain Systems. In *Int. Conf. on Distributed Computing Systems (ICDCS)*. IEEE, 1337–1347.

[5] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. 2019. SharPer: Sharding Permissioned Blockchains Over Network Clusters. *arXiv preprint arXiv:1910.00765* (2019).

[6] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, et al. 2018. Hyperledger Fabric: a distributed operating system for permissioned blockchains. In *European Conf. on Computer Systems (EuroSys)*. ACM, 30.

[7] Elli Androulaki, Christian Cachin, Angelo De Caro, and Eleftherios Kokoris-Kogias. 2018. Channels: Horizontal scaling and confidentiality on permissioned blockchains. In *European Symposium on Research in Computer Security (ESORICS)*. Springer, 111–131.

[8] Gabriel Bracha and Sam Toueg. 1985. Asynchronous consensus and broadcast protocols. *Journal of the ACM (JACM)* 32, 4 (1985), 824–840.

[9] Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. 2011. *Introduction to reliable and secure distributed programming.* Springer Science & Business Media.

[10] Christian Cachin and Marko Vukolić. 2017. Blockchain Consensus Protocols in the Wild. In *Int. Symposium on Distributed Computing (DISC)*. 1–16.

[11] Miguel Castro, Barbara Liskov, et al. 1999. Practical Byzantine fault tolerance. In *Symposium on Operating systems design and implementation (OSDI)*, Vol. 99. USENIX Association, 173–186.

[12] JP Morgan Chase. 2016. Quorum white paper. (2016).

[13] Byung-Gon Chun, Petros Maniatis, Scott Shenker, and John Kubiatowicz. 2007. Attested append-only memory: Making adversaries stick to their word. In *Operating Systems Review (OSR)*, Vol. 41-6. ACM SIGOPS, 189–204.

[14] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, et al. 2013. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)* 31, 3 (2013), 8.

[15] Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. 2010. Schism: a workload-driven approach to database replication and partitioning. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 48–57.

[16] Hung Dang, Tien Tuan Anh Dinh, Dumitrel Loghin, Ee-Chien Chang, Qian Lin, and Beng Chin Ooi. 2019. Towards Scaling Blockchain Systems via Sharding. In *SIGMOD Int. Conf. on Management of Data*. ACM.

[17] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: amazon's highly available key-value store. In *Operating Systems Review (OSR)*, Vol. 41. ACM SIGOPS, 205–220.

[18] Amr El Abbadi, Dale Skeen, and Flaviu Cristian. 1985. An efficient, fault-tolerant protocol for replicated data management. In *SIGACT-SIGMOD symposium on Principles of database systems*. ACM, 215–229.

[19] Amr El Abbadi and Sam Toueg. 1985. Availability in partitioned replicated databases. In *SIGACT-SIGMOD symposium on Principles of database systems*. ACM, 240–251.

[20] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. 1985. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)* 32, 2 (1985), 374–382.

[21] Ethan Frey and Christopher Goes. [n. d.]. Cosmos Inter-Blockchain Communication (IBC) Protocol. https://cosmos.network. ([n. d.]). 2018.

[22] Danezis George and Sarah Meiklejohn. 2016. Centrally Banked Cryptocurrencies. In *Network and Distributed System Security Symposium (NDSS)*.

[23] Christian Gorenflo, Stephen Lee, Lukasz Golab, and Srinivasan Keshav. 2019. Fastfabric: Scaling hyperledger fabric to 20,000 transactions per second. In *Int. Conf. on Blockchain and Cryptocurrency (ICBC)*. IEEE, 455–463.

[24] Suyash Gupta, Sajjad Rahnama, Jelle Hellings, and Mohammad Sadoghi. 2020. ResilientDB: Global Scale Resilient Blockchain Fabric. *arXiv preprint arXiv:2002.00160* (2020).

[25] Jelle Hellings, Daniel P Hughes, Joshua Primero, and Mohammad Sadoghi. 2020. Cerberus: Minimalistic Multi-shard Byzantine-resilient Transaction Processing. *arXiv preprint arXiv:2008.04450* (2020).

[26] Zsolt István, Alessandro Sorniotti, and Marko Vukolić. 2018. StreamChain: Do Blockchains Need Blocks?. In *Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers (SERIAL)*. ACM, 1–6.

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

[28] Jae Kwon. 2014. Tendermint: Consensus without mining. *Draft v. 0.6, fall* (2014).

[29] Leslie Lamport. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* 21, 7 (1978), 558–565.

[30] Leslie Lamport. 2006. Fast paxos. *Distributed Computing* 19, 2 (2006), 79–103.

[31] Leslie Lamport et al. 2001. Paxos made simple. *ACM Sigact News* 32, 4 (2001), 18–25.

[32] Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. 2016. A secure sharding protocol for open blockchains. In *SIGSAC Conf. on Computer and Communications Security (CCS)*. ACM, 17–30.

[33] J-P Martin and Lorenzo Alvisi. 2006. Fast byzantine consensus. *Transactions on Dependable and Secure Computing* 3, 3 (2006), 202–215.

[34] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system. (2008).

[35] Faisal Nawab and Mohammad Sadoghi. 2019. Blockplane: A global-scale byzantizing middleware. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 124–135.

[36] Diego Ongaro and John K Ousterhout. 2014. In search of an understandable consensus algorithm.. In *Annual Technical Conference (ATC)*. USENIX Association, 305–319.

[37] Andrew Pavlo, Carlo Curino, and Stanley Zdonik. 2012. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In *SIGMOD Int. Conf. on Management of Data*. ACM, 61–72.

[38] Pingcheng Ruan, Dumitrel Loghin, Quang-Trung Ta, Meihui Zhang, Gang Chen, and Beng Chin Ooi. 2020. A Transactional Perspective on Execute-order-validate Blockchains. In *SIGMOD International Conference on Management of Data*. ACM, 543–557.

[39] Ankur Sharma, Felix Martin Schuhknecht, Divya Agrawal, and Jens Dittrich. 2019. Blurring the lines between blockchains and database systems: the case of hyperledger fabric. In *SIGMOD International Conference on Management of Data*. ACM, 105–122.

[40] Rebecca Taft, Essam Mansour, Marco Serafini, Jennie Duggan, Aaron J Elmore, Ashraf Aboulnaga, Andrew Pavlo, and Michael Stonebraker. 2014. E-store: Fine-grained elastic partitioning for distributed transaction processing systems. *Proceedings of the VLDB Endowment* 8, 3 (2014), 245–256.

[41] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J Abadi. 2012. Calvin: fast distributed transactions for partitioned database systems. In *SIGMOD Int. Conf. on Management of Data*. ACM, 1–12.

[42] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, and Lau Cheuk Lung. 2010. EBAWA: Efficient Byzantine agreement for wide-area networks. In *Int. Symposium on High Assurance Systems Engineering (HASE)*. IEEE, 10–19.

[43] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, Lau Cheuk Lung, and Paulo Verissimo. 2013. Efficient byzantine fault-tolerance. *IEEE Trans. Comput.* 62, 1 (2013), 16–30.

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

[45] Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova. 2018. RapidChain: Scaling blockchain via full sharding. In *SIGSAC Conf. on Computer and Communications Security*. ACM, 931–948.