Towards Scaling Blockchain Systems via Sharding

Hung Dang, Tien Tuan Anh Dinh, Dumitrel Loghin
Ee-Chien Chang, Qian Lin, Beng Chin Ooi
National University of Singapore
{hungdang,dinhtta,dumitrel,changec,linqian,ooibc}@comp.nus.edu.sg

ABSTRACT

Existing blockchain systems scale poorly because of their distributed consensus protocols. Current attempts at improving blockchain scalability are limited to cryptocurrency. Scaling blockchain systems under general workloads (i.e., non-cryptocurrency applications) remains an open question.

This work takes a principled approach to apply sharding to blockchain systems in order to improve their transaction throughput at scale. This is challenging, however, due to the fundamental difference in failure models between databases and blockchain. To achieve our goal, we first enhance the performance of Byzantine consensus protocols, improving individual shards' throughput. Next, we design an efficient shard formation protocol that securely assigns nodes into shards. We rely on trusted hardware, namely Intel SGX, to achieve high performance for both consensus and shard formation protocol. Third, we design a general distributed transaction protocol that ensures safety and liveness even when transaction coordinators are malicious. Finally, we conduct an extensive evaluation of our design both on a local cluster and on Google Cloud Platform. The results show that our consensus and shard formation protocols outperform state-of-the-art solutions at scale. More importantly, our sharded blockchain reaches a high throughput that can handle Visa-level workloads, and is the largest ever reported in a realistic environment.

CCS CONCEPTS

• **Information systems** → *Distributed database transactions*; • **Security and privacy** → *Hardware-based security protocols.*

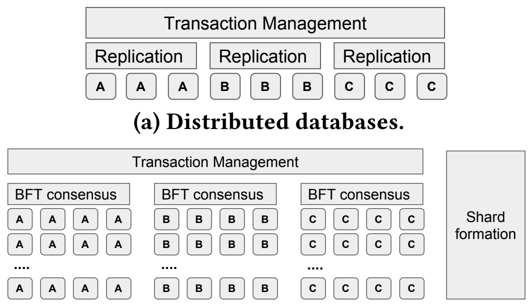
ACM Reference Format:
Hung Dang, Tien Tuan Anh Dinh, Dumitrel Loghin and Ee-Chien Chang, Qian Lin, Beng Chin Ooi. 2019. Towards Scaling Blockchain Systems via Sharding. In *2019 International Conference on Management of Data (SIGMOD '19), June 30–July 5, 2019, Amsterdam, Netherlands.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3299869.3319889

1 INTRODUCTION

Blockchain systems offer data transparency, integrity and immutability in a decentralized and potentially hostile environment. These strong security guarantees come at a dear cost to scalability, for blockchain systems have to rely on distributed consensus protocols which have been shown to scale poorly, both in terms of number of transactions per second (tps) and number of nodes [21].

A number of works have attempted to scale consensus protocols, ultimately aiming to handle average workloads of centralized systems such as Visa. One scaling approach is to exploit trusted hardware [2, 4, 10]. However, its effectiveness has not been demonstrated on data-intensive blockchain workloads. The second approach is to use sharding, a well-studied and proven technique to scale out databases, to divide the blockchain network into smaller committees so as to reduce the overhead of consensus protocols. Examples of sharded blockchains include Elastico [33], OmniLedger [27] and RapidChain [47]. These systems, however, are limited to cryptocurrency applications in an open (or permissionless) setting. Since they focus on a simple data model, namely the unspent transaction output (UTXO) model, these approaches do not generalize to applications beyond Bitcoin.

Our work takes a principled approach to extend *sharding* to permissioned blockchain systems. Existing works on sharded blockchains target permissionless systems and focus on security. Here, our focus is on performance. In particular, our goal is to design a blockchain system that can support a large network size equivalent to that of major cryptocurrencies like Bitcoin [40] and Ethereum [13]. At the same time, it achieves high transaction throughput that can handle the average workloads of centralized systems such as Visa, which is around $2,000 - 4,000$ transactions per second [11]. Finally, the system supports any blockchain application from domains such as finance [3], supply chain management [23] and healthcare [36], not being limited to cryptocurrencies.



(a) Distributed databases.

(b) Sharded blockchains.

Figure 1: Comparison of sharding protocols.

Sharding protocols have been extensively studied in distributed database systems. A sharding protocol must ensure both atomicity and isolation of transaction execution. State-of-the-art protocols [22, 39, 46, 48] aim to improve performance for distributed transactions in geo-replicated settings. However, they cannot be directly extended to blockchain systems, due to a fundamental difference in the failure models that databases and blockchains consider. Traditional databases assume the crash-failure model, in which a faulty node simply stops sending and responding to requests. On the other hand, blockchain systems operate in a more hostile environment, therefore they assume a stronger failure model, namely Byzantine failure, to account for malicious attackers. Figure 1 highlights the differences between distributed databases and sharded blockchains.

The distinction in failure models leads to three challenges when applying database sharding techniques to blockchains. First, high-performance consensus protocols used in distributed databases [29, 41], cannot be applied to blockchains. Instead, blockchains rely on Byzantine Fault Tolerance (BFT) consensus protocols which have been shown to be a scalability bottleneck [21]. Thus, the first challenge is to scale BFT consensus protocols. Second, in a distributed database any node can belong to any shard, but a blockchain must assign nodes to shards in a secure manner to ensure that no shard can be compromised by the attacker. The second challenge, therefore, is to achieve secure and efficient shard formation. Third, the distributed database relies on highly available transaction coordinators to ensure atomicity and isolation, but coordinators in the blockchain may be malicious. Consequently, the third challenge is to enable secure distributed transactions even when the coordinator is malicious.

We tackle the first challenge, that of improving BFT consensus protocols, by leveraging trusted execution environment (TEE) (e.g., Intel SGX [35]) to eliminate equivocation in the Byzantine failure model. Without equivocation, existing BFT protocols can achieve higher fault tolerance with the same number of nodes (i.e., a committee of n nodes can tolerate up to $\frac{n-1}{2}$ non-equivocating Byzantine failures, as opposed to $\frac{n-1}{3}$ failures in the original threat model [10, 17, 30]). We introduce two major optimizations to the TEE-assisted BFT consensus protocol to lower its communications overhead, which further improves the system throughput at scale.

We leverage the TEE to design an efficient and secure shard formation protocol, addressing the second challenge. More specifically, we implement a trusted randomness beacon inside the TEE to generate unbiased random values in a distributed setting. Furthermore, we exploit the increased fault tolerance of our TEE-assisted consensus protocol to reduce the shard size. For example, to tolerate a 25% adversary, our sharding protocol requires 80-node committees as opposed to 600-node committees used in related works [27, 33].

We tackle the final challenge, that of enabling distributed transactions for general blockchain applications, by using two-phase locking (2PL) and two-phase commit (2PC) protocols. They ensure isolation and atomicity for cross-shard transactions. Furthermore, they support any data model, as opposed to UTXO-optimized protocols that do not generalize [47] beyond Bitcoin. To prevent malicious transaction coordinators from causing infinite blocking as in Omniledger [27], we design a protocol that runs the 2PC's coordination logic as a BFT replicated state machine.

In summary, this paper makes the following contributions:

- To our knowledge, we are the first to present a sharded blockchain that supports workloads beyond cryptocurrency and scales scale to few thousands of transactions per second.
- Our sharded blockchain design consists of three key novelties: (i) optimizations that improve the performance of the consensus protocol running within each individual shard, (ii) an efficient shard formation protocol, and (iii) a secure distributed transaction protocol that handles cross-shard, distributed transactions.
- We conduct extensive, large-scale experiments to evaluate the performance of our design. We run our experiments on a local cluster with 100 nodes and on a realistic setup consisting of over 1400 Google Cloud Platform (GPC) nodes distributed across 8 regions. On GPC setup, we achieve a throughput of over 3, 000 transactions per second which, to the best of our knowledge, is the largest ever reported in a realistic environment.

The remaining of this paper is structured as follows. Section 2 provides background on database sharding techniques, blockchain consensus protocols, and Intel SGX. Section 3 describes the key challenges in extending database sharding to blockchain. Section 4 discusses how we improve the underlying consensus protocol running in each individual shard. Section 5 discusses the committee formation protocol. Section 6 presents our distributed transaction protocol. Section 7 reports the performance of our design. In Section 8 we discuss the related work, before concluding in Section 9.

2 PRELIMINARIES

2.1 Sharding in Databases

Traditional database systems achieve scalability by dividing the database states into independent shards (or partitions). By distributing the workload over multiple shards, the overall capacity of the system increases. Sharding requires coordination to ensure ACID properties for transactions that access multiple shards. Two important coordination protocols are distributed commit such as two-phase commit (2PC) which ensures atomicity, and concurrency control such as two-phase locking (2PL) which achieves isolation.

In this paper, we use the term *sharding* to refer to the combination of replication and partitioning as shown in Figure 1a. This architecture is adopted by recent distributed database systems to achieve fault tolerance and scalability [22]. Each partition is replicated over multiple replicas, and its content is kept consistent by consensus protocols [29, 41]. Transaction management and consensus protocols can be combined more judiciously, as opposed to layering one on top of another, to achieve better performance [39, 46, 48].

Sharding in database systems assumes crash-failure model, in which a faulty node stops sending and responding to requests. There are three important implications of this assumption. First, efficient consensus protocols catered for crash-failure model can be used to achieve high performance. Second, creating a shard is simple. For example, a node can be assigned to a shard based on its location. Third, the coordinators that drive coordination protocols are fully trusted.

2.2 Blockchains Consensus Protocols

Blockchain is essentially a distributed, append-only ledger that stores a sequence of transactions. The blocks are chained together by cryptographic hash pointers. The blockchain is maintained by a set of mutually distrusting nodes (a.k.a. replicas or validators). These nodes run a consensus protocol to ensure the blockchain's consistency under Byzantine (or arbitrary) failures. This is in contrast to distributed databases whose threat model does not account for Byzantine failures or malicious users [20].

Blockchain consensus protocols should achieve both *safety* and *liveness* despite Byzantine failures. Safety means that honest (non-Byzantine) nodes agree on the same value, whereas liveness means that these nodes eventually agree on a value. Two major classes of blockchain consensus protocols are Byzantine Fault Tolerance and Nakamoto consensus.

Byzantine Fault Tolerant (BFT) protocols. Practical Byzantine Fault Tolerance (PBFT) [15], the most well-known BFT protocol, consists of three phases: a *pre-prepare* phase in which the leader broadcasts requests as `pre-prepare` messages, the *prepare* phase in which replicas agree on the ordering of the requests via `prepare` messages, and the *commit phase* in which replicas commit to the requests and their order via `commit` messages. Each node collects a quorum of `prepare` messages before moving to the commit phase, and executes the requests after receiving a quorum of `commit` messages. A faulty leader is replaced via the *view change* protocol. The protocol uses $O(N^2)$ messages for N replicas. For $N \geq 3f + 1$, it requires a quorum size of $2f + 1$ to tolerate f failures. It achieves safety in asynchronous networks, and liveness in partially synchronous networks wherein messages are delivered within an unknown but finite bound. More recent BFT protocols [32] extend PBFT to optimize for the normal case (without view change).

Nakamoto consensus protocols. Proof-of-Work (PoW) [40], as used in Bitcoin, is the most well-known instance of Nakamoto consensus. The protocol randomly selects a leader to propose the next block. Leader selection is a probabilistic process in which a node must solve a computational puzzle to claim leadership. The probability of solving the puzzle is proportional to the amount of computational power the node possesses over the total power of the network. The protocol suffers from forks which arise when multiple nodes proposes blocks roughly at the same time. It has low throughput, but can scale to a large number of nodes.

Nakamoto consensus protocols quantify Byzantine tolerance in terms of the cumulative resources belonging to the Byzantine nodes (e.g., fraction of the total computational power). Their safety depends not only on the Byzantine threshold, but also on network latency. Under a fully synchronous network, safety is guaranteed against 50% Byzantine attackers [40]. However, this threshold drops quickly in a partially synchronous network, going below 33% when the latency is equal to the block time [42].

2.3 Trusted Execution Environment (TEE)

One approach to improve BFT protocols is to assume a hybrid failure model in which some components are trusted and only fail by crashing, while others behave in a Byzantine manner. This model is realized by running the trusted components inside a TEE. One important security guarantees of TEE is that it ensures the integrity of the protected components, so that the attackers cannot tamper with their execution and cause them to deviate from the prescribed protocols. Our work uses Intel Software Guard Extensions (SGX) [35] to provision the TEE. But we note that our design can work with other TEE instantiations (e.g., TrustZone [8], Sanctum [18]).

Enclave protection. Intel SGX provides TEE support in the form of *hardware enclave*. An enclave is a CPU-protected address space which is accessible *only* by the code within the enclave (i.e., the trusted component). Multiple enclaves can be instantiated by non-privileged user processes. The enclaves are isolated from each other, from the operating system (OS) and from other user processes. The enclave code, however, can invoke OS services such as IO and memory management.

Attestation. A user can verify if a specific TEE is correctly instantiated and running at a remote host via a remote attestation protocol [9]. Once the enclave in question has been initiated, the CPU computes a *measurement* of this enclave represented by the hash of its initial state. The CPU then signs the measurement with its private key. The user can verify the signed message, and then compare the measurement against a known value.

Data sealing. The TEE can persist its state to non-volatile memory via the data sealing mechanism, which allows for recovery after crash failures. An enclave *seals* its data by first requesting the CPU for a unique key bound to its measurement, then encrypting the data before storing it on persistent storage. This mechanism ensures the data can only be decrypted by the enclave that sealed it. However, enclave recovery is subject to rollback attacks wherein an attacker (e.g., the malicious OS) provides properly sealed but stale data to the enclave [34].

Other cryptographic primitives. SGX provides `sgx_read_rand` and `sgx_get_trusted_time` functions to the enclave processes. The former generates unbiased random numbers, the latter returns the elapsed time relative to a reference point.

3 OVERVIEW

In this section, we discuss our goals and illustrate them with a running example. We detail the challenges in realizing these goals and present our sharding approach. Finally, we describe the system model and security assumptions.

3.1 Goals

The design of a highly scalable blockchain must meet the following three goals: (i) support a large network size equivalent to that of major cryptocurrencies like Bitcoin and Ethereum, (ii) achieve high transaction throughput that can handle the average workloads of centralized systems such as Visa, and (iii) support general workloads and applications beyond cryptocurrencies. The resulting blockchain will enable scale-out applications that offer the security benefits of a decentralized blockchain with a performance similar to that of a centralized system. To better motivate and illustrate our design, we use the following example throughout the paper.

Running example. Consider a consortium of financial institutions that offer cross-border financial services to their customers. They implement a blockchain solution that provides a shared ledger for recording transactions which can be payments, asset transfers or settlements. Unlike Bitcoin or Ripple [3], there is no native currency involved. The ledger is distributed and its content is agreed upon by consortium members via distributed consensus. Given the amount of money at stake, the blockchain solution must be tolerant to Byzantine failures, so that group members are protected against attacks that compromise the ledger in order to double-spend or to revoke transactions. As the consortium can admit new members, the system should not assume any upper bound on the consortium size. Finally, the blockchain must be able to support high request rates and deliver high transaction throughput.

3.2 Challenges and Approach

Building a blockchain system that achieves all three goals above at the same time is challenging. To have high transaction throughput (second goal), it is necessary to build on top of a permissioned blockchain. But such a blockchain uses BFT protocols which do not scale to a large number of nodes, thus contradicting the first goal. As a result, one challenge is to reconcile the first two goals by making BFT protocols more scalable. We note that scalability here means fault scalability, which means that protocol's performance degrades gracefully as the number of tolerated failures increases. We address this challenge by using trusted hardware to remove the attacker's ability to equivocate. Specifically, we use a hardware-assisted PBFT protocol that requires only $N = 2f + 1$ replicas to tolerate f failures. We implement this protocol on top of Hyperledger's PBFT implementation and further improve its scalability by introducing two protocol-level optimizations that reduce the number of message broadcasts, and an implementation-specific optimization that avoids dropping important messages.

We cannot achieve the first two goals by improving BFT protocols alone, because there is a limit on scalability due to the quadratic communication cost of $O(N^2)$. Our approach is to apply the database sharding technique to partition the network and the blockchain states into smaller shards, where each shard is small enough to run a BFT protocol. In distributed databases, nodes can be assigned to shards randomly or based on their location. But in blockchain, the process of assigning nodes to shards, called *shard formation*, must be done carefully to ensure security because each shard can only tolerate a certain number of Byzantine failures. In particular, given a network of N nodes, a fraction s of which are Byzantine, shard formation must guarantee with overwhelming probability that no shard of size $n \ll N$ contains more than f Byzantine nodes over the entire lifetime of the system[1]. The relationship between f and n depends on the consensus protocol. The challenge here is to perform shard

[1]It must be the case that $\frac{f}{n} \leq s$ in order for this requirement to be met.

formation securely and efficiently. Existing solutions use expensive cryptographic protocols and the resulting shards are large. In contrast, we leverage TEE to implement an efficient trusted randomness beacon that serves as the random source for the shard formation. Furthermore, our fault-scalable PBFT protocol allows for shards of smaller size. This leads to higher throughput per shard, and more shards given the same network size.

As in distributed databases, sharding requires coordination protocols for cross-shard (or distributed) transactions. Two important properties of coordination protocols are *safety* which means atomicity and isolation, and *liveness* which means the transaction will eventually abort or commit. The challenge in realizing our third goal is to design a coordination protocol that supports non-UTXO distributed transactions, while achieving safety and liveness even when the coordinator is malicious. Existing sharded blockchains [27, 33, 47] do not fully address this challenge, as we elaborate later in Section 6. Our approach is to use the classic 2PC and 2PL protocols to ensure safety, and run 2PC in a Byzantine tolerant shard to avoid malicious blocking. This coordination protocol works for all blockchain data models and applications.

The key components of our design are summarized in Figure 1b. First, our shard formation protocol securely partitions the network into multiple committees, thereby allowing the system throughput to scale with the number of nodes in the system. This protocol relies on a trusted randomness beacon implemented inside a TEE for efficiency. Second, each shard runs our scalable BFT protocol which achieves high throughput at scale by combining TEE with other optimizations. Finally, layered on top of the shards is a distributed transaction protocol that achieves safety and liveness for general blockchain applications.

3.3 System and Threat Model

System model. We consider a blockchain system of N nodes, with a fraction s of the network under the attacker's control, while the remaining fraction is honest. The shard formation protocol partitions the nodes into k committees, each consisting of $n \ll N$ nodes. Each committee can tolerate at most $f < n$ Byzantine nodes. The committees maintain disjoint partitions of the blockchain states (i.e., shards). Unless otherwise stated, the network is partially synchronous, in which messages sent repeatedly with a finite time-out will eventually be received. This is a standard assumption in existing blockchain systems [27, 33].

In the running example above, suppose the consortium comprises 400 institutions, among which 100 members actively collude so that they can revoke transactions that transfer their assets to the remaining institutions. In such a case, $N = 400$ and $s = 25\%$. Suppose further that the consortium partitions their members into four equally-sized committees, then $n = 100$. Each committee *owns* a partition of the ledger states. The committee members run a consensus protocol to process transactions that access the committee's states. If PBFT is used, each committee can tolerate at most $f = \frac{n-1}{3} = 33$ Byzantine nodes.

Every node in the system is provisioned with TEEs. We leverage Intel SGX in our implementations, but our design can work with any other TEE instantiations, for example hardware-based TEEs such as TrustZone [8], Sanctum [18], TPMs [5], or software-based TEEs such as Overshadow [16].

Threat model. The attacker has full control of the Byzantine nodes. It can read and write to the memory of any running process, even the OS. It can modify data on disk, intercept and change the content of any system call. It can modify, reorder and delay network messages arbitrarily. It can start, stop and invoke the local TEE enclaves with arbitrary input. However, its control of the enclaves is restricted by the TEE threat model described below. The attacker is adaptive, like in Elastico [33] and OmniLedger [27], meaning that it can decide which honest nodes to corrupt. However, the corruption does not happen instantly, like in Algorand [37], but takes some time to come into effect. Furthermore, the attacker can only corrupt up to a fraction of s nodes at a time. It is computationally bounded and cannot break standard cryptographic assumptions. Finally, it does not mount denial-of-service attacks against the system.

The threat model for TEE is stronger than what SGX currently offers. In particular, SGX assumes that the adversary cannot compromise the integrity *and* confidentiality of protected enclaves. For TEE, we also assume that integrity protection mechanism is secure. But there is no guarantee about confidentiality protection, except for a number of important cryptographic primitives: attestation, key generation, random number generation, and signature. In other words, enclaves have no private memory except for areas related to its private keys, i.e., they run in a seal-glassed proof model where their execution is transparent [43]. This model admits recent side-channel attacks on SGX that leak enclave data [12]. Although attacks that leak attestation and other private keys are excluded [44], we note that there exist both software and hardware techniques to harden important cryptographic operations against side channel attacks.

4 SCALING CONSENSUS PROTOCOLS

4.1 Scaling BFT Consensus Protocol

PBFT, the most prominent instance of BFT consensus protocols, has been shown not to scale beyond a small number of nodes due to its communication overhead [21]. In the running example, this means each committee in the consortium

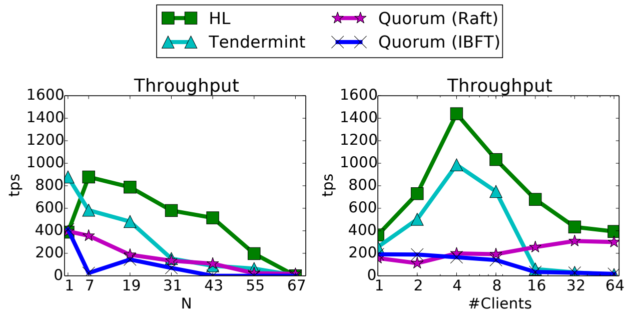


Figure 2: Comparison of BFT protocols with varying number of nodes and clients.

can only comprise dozens of institutions. Furthermore, the probability of the adversary controlling more than a third of the committee is high when the committee size is small. Our goal is to improve both the protocol's communication overhead and its fault tolerance.

Why PBFT? There are several BFT implementations for blockchains. PBFT is adopted by Hyperledger. Tendermint [28] – a variant of PBFT – is used by Ethermint and Cosmo. Istanbul BFT (IBFT) is adopted by Quorum. Raft [41], which only tolerates crash failures, is implemented by Coco to tolerate Byzantine failures by running the entire protocol inside Intel SGX [4]. Figure 2 compares the throughputs of these BFT implementations, where we use Raft implementation in Quorum as an approximation for Coco whose source code is not available. Due to space constraint, we only highlight important results here, and include detailed discussion in Appendix C.3. PBFT consistently outperforms the alternatives at scale. The reason is that PBFT design permits pipelined execution, whereas IBFT and Tendermint proceed in lockstep. Although pipelined execution is possible in Raft, this property is not exploited in Quorum. From this observation, we base our sharded blockchain design on top of Hyperledger, and focus on improving PBFT.

Reducing the number of nodes. If a consensus protocol can prevent Byzantine nodes from equivocating (i.e., issuing conflicting statements to different nodes), it is possible to tolerate up to $f = \frac{N-1}{2}$ non-equivocating Byzantine failures out of N nodes [17]. Equivocation can be eliminated by running the entire consensus protocol inside a TEE, thereby reducing the failure model from Byzantine to crash failure [4]. We do not follow this approach, however, because it incurs a large trusted code base (TCB). A large TCB is undesirable for security because it is difficult, if not impossible to conduct security analysis for the code base, and it increases the number of potential vulnerabilities [6]. Instead, we adopt the protocol proposed by Chun *et al.* [17] which uses a small trusted log abstraction called attested append-only memory to remove equivocation. The log is maintained inside the TEE so that the attacker cannot tamper with its operations. We implement this protocol on top of Hyperledger Fabric v0.6 using SGX, and call it AHL (Attested HyperLedger).

AHL maintains different logs for different consensus message types (e.g., `pre-prepare`, `prepare`, `commit`). Before sending out a new message, each node has to append the message's digest to the corresponding log. The proof of this operation contains the signature created by the TEE, and is included in the message. AHL requires all valid messages to be accompanied by such proof. Each node collects and verifies $f + 1$ `prepare` messages before moving to the commit phase, and $f + 1$ `commit` messages before executing the request. AHL periodically seals the logs and writes them to persistent storage. This mechanism, however, does not offer protection against rollback attacks [34]. We describe how to extend AHL to guard against these attacks in Appendix A.

Optimizing communications. Our evaluation of AHL in Section 7 shows that it fails to achieve the desired scalability. We observe a high number of consensus messages being dropped, which leads to low throughput when the network size increases. From this observation, we introduce two optimizations to improve the communications of the system, and refer to the resulting implementation as AHL+.

First, we improve Hyperledger implementation which uses the same network queue for both consensus and request messages. In particular, we split the original message queue into two separated channels, each for a different type of message. Messages received from the network contain metadata that determines their types and are forwarded to the corresponding channel. This separation prevents request messages from overwhelming the queue and causing consensus messages to be dropped.

Second, we note that when a replica receives a user request, the PBFT protocol specification states that the request is broadcast to all nodes [14]. However, this is not necessary as long as the request is received at the leader, for the leader will broadcast the request again during the *pre-prepare* phase. Therefore, we remove the request broadcast. The replica receiving the request from the client simply forwards it to the leader. We stress that this is a design-level optimization.

We also consider another optimization adopted by Byzcoin [26], in which the leader collects and aggregates other nodes' messages into a single authenticated message. Each node forwards its messages to the leader and verifies the aggregated message from the latter. As a result, the communication overhead is reduced to $O(N)$. This design, called AHLR (Attested HyperLedger Relay), is implemented on top of AHL via an enclave that verifies and aggregates messages. Given valid $f + 1$ signed messages for a request *req*, in phase

p of consensus round o, the enclave issues a proof indicating that there has been a quorum for $\langle req, p, o\rangle$.

Security analysis. The trusted log operations in AHL are secure because they are signed by private keys generated inside the enclave. Because the adversary cannot forge signatures of the logs' operations, it is not possible for the Byzantine nodes to equivocate. As shown in [17], given no more than $f = \frac{N-1}{2}$ non-equivocating Byzantine failures, AHL guarantees safety regardless of the network condition, and liveness under partially synchronous network. AHL+ only optimizes communication between nodes and does not change the consensus messages, therefore it preserves AHL properties. AHLR only optimizes communication in the normal case when there is no view change, and uses the same view change protocol as in AHL. Because message aggregation is done securely within the enclave, AHLR has the same safety and liveness guarantees as AHL.

4.2 Scaling PoET Consensus Protocol

Proof of Elapsed Time (PoET) is a variant of Nakamoto consensus, wherein nodes are provisioned with SGXs. Each node asks the enclave for a randomized `waitTime`. Only after such `waitTime` expires does the enclave issue a *wait certificate* or create a new `waitTime`. The node with the shortest `waitTime` becomes the leader and is able to propose the next block.

Similar to PoW, PoET suffers from forks and stale blocks. Due to propagation delays, if multiple nodes obtain their certificates roughly at the same time, they will propose conflicting blocks, creating forks in the blockchain. The fork is resolved based on the aggregate resource contributed to the branches, with blocks on the losing branches discarded as stale blocks. Stale block rate has a negative impact on both the security and throughput of the system [25].

PoET+: Improving PoET. We improve PoET by restricting the number of nodes competing to propose the next block, thereby reducing the stale block rate. We call this optimized protocol PoET+. Unlike PoET, when invoked to generate a wait certificate, PoET+ first uses `sgx_read_rand` to generate a random l-bit value `q` that is bound to the wait certificate. Only wait certificates with `q` $= 0$ are considered valid. The node with a valid certificate and the shortest `waitTime` becomes the leader. PoET+ leader selection can thus be seen as a two-stage process. The first stage samples uniformly at random a subset of $n' = n \cdot 2^{-l}$ nodes. The second stage selects uniformly at random a leader among these n' nodes. It can be shown that the expected number of stale blocks in PoET+ is smaller than that in PoET.

PoET+ vs AHL+. PoET+ safety depends not only on the Byzantine threshold, but also on network latency. In a partially synchronous network, its fault tolerance may drop below 33% [25]. This is in contrast to AHL+ whose safety does not depend on network assumption. More importantly, our performance evaluation of PoET+ (included in Appendix C) shows that it has lower throughput than AHL+. Therefore, we adopt AHL+ for the design and implementation of the sharded blockchain.

5 SHARD FORMATION

Forming shards in a blockchain system is more complex than in a distributed database. First, the nodes must be assigned to committees in an unbiased and random manner. Second, the size of each committee must be selected carefully to strike a good trade-off between performance and security. And finally, committee assignment must be performed periodically to prevent an adaptive attacker from compromising a majority of nodes in a committee. This section presents our approach of exploiting TEEs to address these challenges.

5.1 Distributed Randomness Generation

A secure shard formation requires an unbiased random number `rnd` to seed the node-to-committee assignment. Given `rnd`, the nodes derive their committee assignment by computing a random permutation π of $[1 : N]$ seeded by `rnd`. π is then divided into approximately equally-sized chunks, each of which represents the members in one committee.

We exploit TEEs to efficiently obtain `rnd` in a distributed and Byzantine environment, by equipping each node with a RandomnessBeacon enclave that returns fresh, unbiased random numbers. Similar to prior works [27, 33, 47], we assume a synchronous network with the bounded delay Δ during the distributed randomness generation procedure.

Our sharded blockchain system works in epochs. Each new epoch corresponds to a new node-to-committee assignment. At the beginning of each epoch, each node invokes the RandomnessBeacon enclave with an epoch number e. The enclave generates two random values `q` and `rnd` using two independent invocations of the `sgx_read_rand` function. It then returns a signed certificate containing $\langle e, \texttt{rnd}\rangle$ if and only if `q` $= 0$. The certificate is broadcast to the network. After a time Δ, nodes lock in the lowest `rnd` they receive for epoch e, and uses it to compute the committee assignment.

The enclave is configured such that it can only be invoked once per epoch, which is to prevent the attacker from selectively discarding the enclave's output in order to bias the final randomness. If the nodes fail to receive any message after Δ, which happens when no node can obtain $\langle e, \texttt{rnd}\rangle$ from its enclave, they increment e and repeat the process. The probability of repeating the process is $P_{\text{repeat}} = (1 - 2^{-l})^N$

where l is the bit length of q. It can be tuned to achieve a desirable trade-off between P_{repeat} and the communication overhead, which is $O(2^{-l}N^2)$. For example, when $l = \log(z)$ for some constant z, $P_{\text{repeat}} \approx 0$ and the communication is $O(N^2)$. When $l = \log(N)$, $P_{\text{repeat}} \approx \mathrm{e}^{-1}$ and the communication is $O(N)$.

Security analysis. Because q and rnd are generated independently inside a TEE, their randomness is not influenced by the attacker. Furthermore, the enclave only generates them once per epoch, therefore the attacker cannot selectively discard the outputs to bias the final result and influence the committee assignment.

5.2 Committee Size

Since committee assignment is determined by a random permutation π of $[1 : N]$ seeded by rnd, it can be seen as random sampling without replacement. Therefore, we can compute the probability of a faulty committee (i.e., a committee containing more than f Byzantine nodes) using the hypergeometric distribution. In particular, let X be a random variable that represents the number of Byzantine nodes assigned to a committee of size n, given the overall network size of N nodes among which up to $F = sN$ nodes are Byzantine. The probability of faulty committee, i.e., the probability that security is broken, is:

$$Pr[X \geq f] = \sum_{x=f}^{n} \frac{\binom{F}{x}\binom{N-F}{n-x}}{\binom{N}{n}} \tag{1}$$

Keeping the probability of faulty committee negligible. We can bound the probability of faulty committee to be negligible by carefully configuring the committee size, based on Equation 1. If $f \leq \frac{n-1}{3}$ (as in the case of PBFT), in the presence of a 25% adversarial power, each committee must contain 600+ nodes to keep the faulty committee probability negligible (i.e., $Pr[X \geq \frac{n-1}{3}] \leq 2^{-20}$). When AHL+ is used, each committee can tolerate up to $f = \frac{n-1}{2}$, thus the committees can be significantly smaller: $n = 80$ for $Pr[X \geq \frac{n-1}{2}] \leq 2^{-20}$.

Smaller committee size leads to better performance for two reasons. First, individual committees achieve higher throughput due to lower communication overhead. Second, there are more committees in the network, which can increase throughput under light contention workloads. We report the committee sizes with respect to different adversarial power and their impact on the overall throughput in Section 7.

5.3 Shard Reconfiguration

An adaptive attacker may compromise a non-faulty committee by corrupting otherwise honest nodes. Our threat model, however, assumes that such node corruption takes time. As a result, we argue that periodic committee re-assignment, or shard reconfiguration, that reshuffles nodes among committees, suffices to guard the system against an adaptive attacker.

Shard reconfiguration occurs at every epoch. At the end of epoch $e-1$, nodes obtain the random seed rnd following the protocol described in Section 5.1. They compute the new committee assignment for epoch e based on rnd. We refer to nodes whose committee assignment changes as *transitioning nodes*. We refer to the period during which transitioning nodes move to new committees as the *epoch transition* period.

During epoch transition, transitioning nodes first stop processing requests of their old committees, then start fetching the states of their new committees from current members of the corresponding committees. Only after the state fetching completes do they officially join the new committee and start processing transactions thereof. During this period, the transitioning nodes do not participate in the consensus protocol of either their old or new committees. Consequently, a naive reconfiguration approach in which all nodes transition at the same time is undesirable, as it renders the system non-operational during the transition period.

Our approach is to have nodes transitioning in batches. In particular, for each committee, only up to B nodes move to new committees at a time. The order by which nodes move is determined based on rnd, which is random and unbiased. In the following, we reason about the impact of B on the safety and liveness of the sharded blockchain.

Safety analysis. Let k be the number of shards, where each shard represents a partition of the global blockchain states. A shard reconfiguration essentially changes the set of nodes that processes requests for each of the k shards. Consider a shard sh, and denote the committee handling sh in epoch $e-1$ by C_{e-1} and in epoch e by C_e. Since B nodes are switching out of C_{e-1} at a time, and there are $\frac{n}{k}$ nodes of C_{e-1} expected to remain in C_e, there are $\frac{n(k-1)}{k \cdot \mathrm{B}}$ *intermediate* committees handling sh during the epoch transition period.

Swapping out B nodes does not violate safety of sh, because the number of Byzantine nodes in the current committee does not increase. On the other hand, when new B nodes are swapped in, the number of Byzantine nodes in the intermediate committee may exceed the safety threshold. As the transitioning nodes are chosen at random based on rnd, the probability of the intermediate committee being faulty follows Equation 1. In expectation, there are $\frac{n(k-1)}{k \cdot \mathrm{B}}$ such intermediate committees during the transitioning from C_{e-1} to C_e. We use Boole's inequality to estimate the probability that the safety of shard sh is violated during the epoch transitioning:

$$Pr(\texttt{faulty}) \leq \sum_{i=1}^{\frac{n(k-1)}{k\cdot \mathsf{B}}} \sum_{x=f}^{n} \frac{\binom{F}{x}\binom{N-F}{n-x}}{\binom{N}{n}} \tag{2}$$

For example, with $n = 80$, $f = \frac{n-1}{2}$, $k = 10$ shards, and $\mathsf{B} = \log(n) = 6$, $Pr(\texttt{faulty}) \approx 10^{-5}$. Based on Equation 2, we can configure B to balance between liveness and safety of the system during epoch transition.

Liveness analysis. During the transitioning, each committee has B nodes not processing requests. If $\mathsf{B} > f$, the shard cannot make progress because the remaining nodes cannot form a quorum. Thus, the larger B is, the higher the risk of loss of liveness during epoch transition.

6 DISTRIBUTED TRANSACTIONS

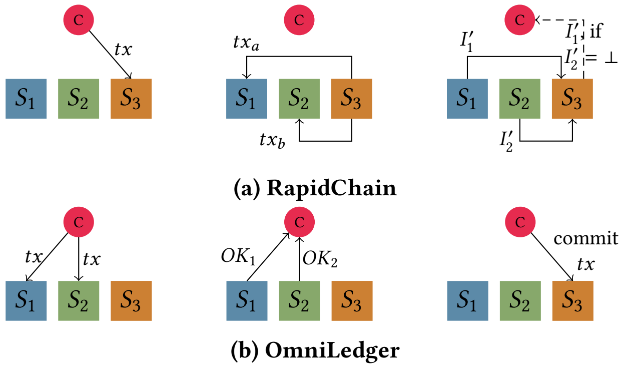
In this section, we explain the challenges in supporting distributed, general transactions for blockchains. We discuss the limitations of state-of-the-art systems: RapidChain [47] and OmniLedger [27] (Elastico [33] is not considered because it does not support distributed transactions). We then present a solution that enables fault-tolerant, distributed, general transactions, and discuss how it can be improved.

6.1 Challenges

In a sharded blockchain, a distributed (or cross-shard) transaction is executed at multiple shards. Appendix B shows that in practical blockchain applications, a vast majority of transactions are distributed. Similar to databases, supporting distributed transactions is challenging due to the safety and liveness requirements. The former means atomicity and isolation that handle failures and concurrency, the latter means that transactions do not block forever. We note that in the sharded blockchain, concurrency does not arise within a single shard, because the blockchain executes transaction sequentially. Instead, as we explain later, concurrency arises due to cross-shard transactions.

UTXO transactions. Bitcoin and many other cryptocurrencies adopt the Unspent Transaction Output (UTXO) data model. A UTXO transaction consists of a list of inputs, and a list of outputs. All the inputs must be the outputs of previous transactions that are unspent (i.e., they have not been used in another transaction). The outputs of the transaction are new, unspent coins. Given a transaction, the blockchain nodes check that its inputs are unspent, and the sum of the outputs is not greater than that of the inputs. If two transactions consume the same unspent coins, only one is accepted.

The simplicity of UTXO model is exploited in previous works to achieve atomicity without using a distributed commit protocol. Consider a simple UTXO transaction $tx = \langle (I_1,I_2),O\rangle$ that spends coins I_1,I_2 in shard S_1 and S_2, respectively, to create a new coin O belonging to shard S_3 (Figure 3a). RapidChain [47] executes tx by splitting it into three sub-transactions: $tx_a = \langle I_1,I'_1\rangle$, $tx_b = \langle I_2,I'_2\rangle$, $tx_c = \langle (I'_1,I'_2),O\rangle$, where I'_1 and I'_2 belong to S_3. tx_a and tx_b essentially transfer I_1 and I_2 to the output shard, which are spent by tx_c to create the final output O. All three sub-transactions are single-shard. In case of failures, when, for example, tx_b fails while tx_a succeeds, RapidChain sidesteps atomicity by informing the owner of I_1 to use I'_1 for future transactions, which has the same effect as rolling back the failed tx.



(a) RapidChain

(b) OmniLedger

Figure 3: Existing works' coordination protocols. C denotes a client. S_1,S_2 are input shards, S_3 is output shard.

Shard$_1$	Shard$_2$	tx_1: $\langle acc_1{+}acc_3\rangle \rightarrow \langle acc_2\rangle$	tx_2: $\langle acc_3\rangle \rightarrow \langle acc_4\rangle$
acc_1, bal_1	acc_3, bal_3	op_{1a}: $\textsc{Debt}[acc_1]$	op_{2a}: $\textsc{Debt}[acc_3]$
acc_2, bal_2	acc_4, bal_4	op_{1b}: $\textsc{Debt}[acc_3]$	op_{2b}: $\textsc{Cred}[acc_4]$
		op_{1c}: $\textsc{Cred}[acc_2]$	

Figure 4: Account-based cross-shard transactions.

RapidChain does not achieve isolation. Consider another transaction tx'_b in S_2 that spends I_2 and is submitted roughly at the same time as tx, the shard serializes the transactions, thus only one of tx_b and tx'_b succeeds. If isolation is achieved, either tx or tx'_b succeeds. But it is possible in RapidChain that both of them fail, because tx_a fails.

Safety for general transaction model. We now show examples demonstrating how RapidChain's approach fails to work for non-UTXO distributed transactions, because it violates both atomicity and isolation. Consider the account-based data model, which is used in Ethereum. Let tx_1: $\langle acc_1{+}acc_3\rangle \rightarrow \langle acc_2\rangle$ be a transaction transferring assets from acc_1 and acc_3 to acc_2, where acc_1, acc_2 belongs to shard S_1 and acc_3 belongs to shard S_2. Following RapidChain, tx_1 is split into $op_{1a}, op_{1b}, op_{1c}$ (Figure 4). If op_{1a} succeeds and op_{1b} fails, due to insufficient funds, for example, op_{1c} cannot

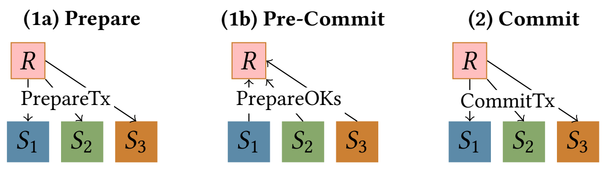


Figure 5: Our coordination protocol.

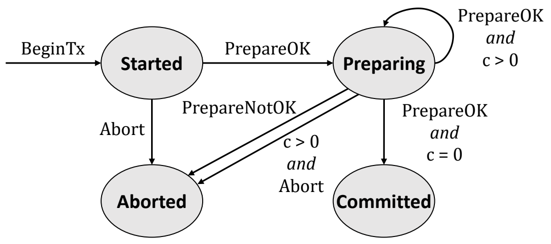


Figure 6: State machine of the reference committee.

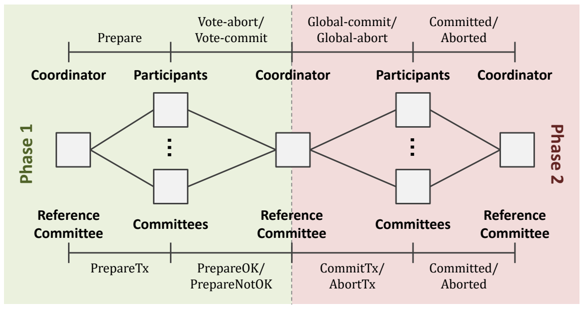


Figure 7: Correspondence between our distributed transaction management protocol (i.e., bottom half) and the original 2PC protocol (i.e., top half).

be executed. In other words, tx_1 does not achieve atomicity because it is executed only partially: acc_1 is already debited and cannot be rolled back.

Let tx_2: $\langle acc_3 \rangle \rightarrow \langle acc_4 \rangle$ be another transaction submitted roughly at the same time as tx_1. In Figure 4, the execution sequence $\langle op_{1a}, op_{1b}, op_{2a}, op_{2b}, op_{1c} \rangle$ is valid in RapidChain, but it breaks isolation (serializability) because tx_2 sees the states of a partially completed transaction.

Liveness under malicious coordinator. OmniLedger [27] achieves safety for the UTXO model by relying on a client to coordinate a *lock/unlock* protocol (Figure 3b). Given a transaction *tx* whose inputs belong to shard S_1 and S_2, and output belongs to shard S_3, the client first obtains locks from S_1 and S_2 (i.e., marking the inputs as spent), before instructing S_3 to commit *tx*. This client-driven protocol suffers from indefinite blocking if the client is malicious. For example, consider a payment channel [31, 38], in which the payee is the client that coordinates a transaction that transfers funds from a payer's account. A malicious payee may pretend to crash indefinitely during the lock/unlock protocol, hence, the payer's funds are locked forever.

6.2 Our Solution

We aim to design a distributed transaction protocol that achieves safety for general blockchain transactions (non-UTXO), and liveness against malicious coordinators. For safety, we use the classic two-phase commit (2PC) and two-phase locking (2PL) protocol as in traditional databases. To guard against a malicious coordinator, we employ a Byzantine fault-tolerant reference committee, denoted by *R*, to serve as a coordinator. *R* runs BFT consensus protocol and implements a simple state machine for 2PC. Given our system and threat model in Section 3.3, *R* is highly (eventually) available. Figure 5 illustrates the transaction flow and Figure 6 depicts the state machine of the reference committee.

The client initiates a transaction *tx* by sending `BeginTx` request to the reference committee. The transaction then proceeds in three steps.

1a) Prepare: Once *R* has executed the `BeginTx` request, it enters Started state. Nodes in *R* then send `PrepareTx` requests to the transaction committees (or tx-committees). The latter wait for a quorum of matching `PrepareTx` to ensure that `BeginTx` has been executed in *R*. Each tx-committee executes the `PrepareTx`. If consensus is reached that *tx* can be committed, which requires that *tx* can obtain all of its locks, the nodes within the committee send out `PrepareOK` messages (or `PrepareNotOK` messages otherwise).

1b) Pre-Commit: When entering Started state, *R* initializes a counter *c* with the number of tx-committees involved in *tx*. After receiving the first quorum of matching responses from a tx-committee, it either decreases *c* and enters the Preparing state, or enters the Aborted state, depending on whether the responses are `PrepareOK` or `PrepareNotOK` respectively. *R* stays in the Preparing states and decreases *c* for every new quorum of `PrepareOK` responses. It moves to Aborted as soon as it receives a quorum of `PrepareNotOK`, and to Committed states when $c = 0$.

2) Commit: Once *R* has entered Committed (or Aborted) state, the nodes in *R* send out `CommitTx` (or `AbortTx`) messages to tx-committees. The latter wait for a quorum of matching messages from *R* before executing the corresponding commit or abort operation.

We remark that the reference committee is not a bottleneck in cross-shard transaction processing, for we can scale it out by running multiple instances of *R* in parallel.

Safety analysis. The safety of our coordination protocol is based on the assumption that both R and tx-committees ensure safety for all transactions/requests they process. This assumption is realized by fine-tuning the committee size according to Equation 1 presented in Section 5.2.

We sketch the proof that our coordination protocol indeed implements the classic 2PC protocol in which reference committee R is the *coordinator*, and tx-committees are the participants. The state machine of the reference committee shown in Figure 6 is identical to that of the coordinator in the original 2PC [24].

Figure 7 illustrates the correspondence between our protocol and the original 2PC protocol. Similar to 2PC, our protocol consists of two main phases. Phase 1 aims to reach the tentative agreement of transaction commit and Phase 2 performs the actual commit of the transaction among shards. Before `BeginTx` is executed at R, the transaction is considered non-existent, hence no tx-committees would accept it. After R enters Started state (i.e., it has logged the transaction), the `PrepareTx` requests are sent to tx-committees. Phase 1 completes when R moves either to Committed or Aborted state. At this point, the current state of R reflects the tentative agreement of transaction commit. When this tentative agreement is conveyed to the tx-committees in Phase 2, they can commit (or abort) the transaction. The original 2PC requires logging at the coordinator and participants for recovery. Our protocol, however, does not need such logging, because the states of R and of tx-committees are already stored on the blockchain. In summary, our protocol always achieves safety for distributed transactions.

Liveness analysis. Recall that we assume a partially synchronous network, in which messages sent repeatedly with a finite time-out will eventually be received. Furthermore, we assume that the size of R is chosen such that the number of Byzantine nodes are less than half. Under these assumptions, the BFT protocol running in R achieves liveness. In other words, R always makes progress, and any request sent to it will eventually be processed. Such eventual availability means that R will not block indefinitely. Thus, the coordination protocol achieves liveness.

6.3 Implementation

We implement our protocol on Hyperledger Fabric which supports smart contracts called chaincodes. The blockchain states are modeled as key-value tuples and accessible to the chaincode during execution. We use the chaincode that implements the SmallBank benchmark to explain our implementation. In Hyperledger, this chaincode contains a `sendPayment` function that reads the state representing acc_1's balance, checks that it is greater than bal, then deducts the bal from acc_1 and updates the state representing acc_2's balance. This chaincode does not support sharding, because the states of acc_1 and acc_2 may belong to different shards. We modify the chaincode so that it can work with our protocol. In particular, for the `sendPayment` function, we split it into three functions: `preparePayment`, `commitPayment`, and `abortPayment`. We implement locking for an account acc by storing a boolean value to a blockchain state with the key "$L\_$"acc. During the execution of `preparePayment`, the chaincode checks if the corresponding lock, namely the tuple $\langle L\_acc, true\rangle$, exists in the blockchain state, and aborts the transaction if it does. If it does not, the chaincode writes the lock to the blockchain. The `commitPayment` function for a transaction tx writes new states (balances) to the blockchain, and removes the locks that were written for tx.

As an optimization to avoid cross-shard communication in normal case (when clients are honest), we let the clients collect and relay messages between R and tx-committees. We directly exploit the blockchain's ledger to record the progress of the commit protocol. In particular, during Prepare phase, the client sends a transaction to the blockchain that invokes the `preparePayment` function. This function returns an error if the Prepare phase fails. The client reads the status of this transaction from the blocks to determines if the result is PrepareOK or PrepareNotOK. We implement the state machine of the reference committee as a chaincode with similar functions that can be invoked during the two phases of our protocol. When interacting with R, all transactions are successful, therefore the client only needs to wait for them to appear on the blocks of R.

6.4 Discussion

Our current design uses 2PL for concurrency control, which may not be able to extract sufficient concurrency from the workload. State-of-the-art concurrency control protocols have demonstrated superior performance over 2PL [39, 46]. We note that the batching nature of blockchain presents opportunities for optimizing concurrency control protocols. We leave the study of these protocols to future work.

In the current implementation, we manually refactor existing chaincodes to support sharding. One immediate extension that makes it easier to port legacy blockchain applications to our system is to instrument Hyperledger codebase with a library containing common functionalities for sharded applications. One common function is state locking. Having such a library helps speed up the refactoring, but the developer still needs to split the original chaincode function to smaller functions that process the Prepare, Commit or Abort requests. Therefore, a more useful extension is to add programming language features that, given a single-shard chaincode implementation, automatically analyze the functions and transform them to support multi-shards execution.

Table 1: Comparisons with other sharded blockchains.

	# machines	Over-subscription	Transaction model	Distributed transaction
Elastico	800	2	UTXO	×
OmniLedger	60	67	UTXO	×
RapidChain	32	125	UTXO	✓
Ours	1400	1	General workload	✓

Another extension to improve usability is to introduce a client library that hides the details of the coordination protocols, so that the users only see single-shard transactions.

7 PERFORMANCE EVALUATION

In this section, we present a comprehensive evaluation of our design. We first demonstrate the performance of the scalable consensus protocols. Next, we report the efficiency of our shard formation protocol. Finally, we evaluate the scalability of our sharding approach. Table 1 contrasts our design and evaluation methodology against existing sharded blockchain systems.

For this evaluation, we use KVStore and Smallbank, two different benchmarks available in BLOCKBENCH [21] — a framework for benchmarking private blockchains. We use the original client driver in BLOCKBENCH, which is open-loop, for our single-shard experiments. For multi-shard experiments, we modified the driver to be closed-loop (i.e., it waits until a cross-shard transaction finishes before issuing a new one). To generate cross-shard transactions, we modified the original KVStore driver to issue 3 updates per transaction, and used the original `sendPayment` transaction in Smallbank that reads and writes two different states.

We conducted experiments in two different settings. One is an in-house (local) cluster consisting of 100 servers, each equipped with Intel Xeon E5-1650 3.5GHz CPUs, 32GB RAM and 2TB hard drive. In this setting, the blockchain node and client run on separate servers. The other setting is Google Cloud Platform (GCP), in which we have separate instances for the clients and for the nodes. A client has 16 vCPUs and 32GB RAM, while a node has 2 vCPUs and 12GB RAM. We use up to 1400 instances over 8 regions (the latency between regions is included in Appendix C).

We used Intel SGX SDK [1] to implement the trusted code base. Since SGX is not available on the local cluster and GCP, we configured the SDK to run in simulation mode. We measured the latency of each SGX operation on Skylake 6970HQ 2.80 GHz CPU with SGX Enabled BIOS support, and injected it to the simulation. Table 3 in the appendix details runtime costs of enclave operations on the SGX-enabled processor. Public key operations are expensive: signing and signature verification take about $450\mu s$ and $844\mu s$, respectively. Context switching and other symmetric key operations take less

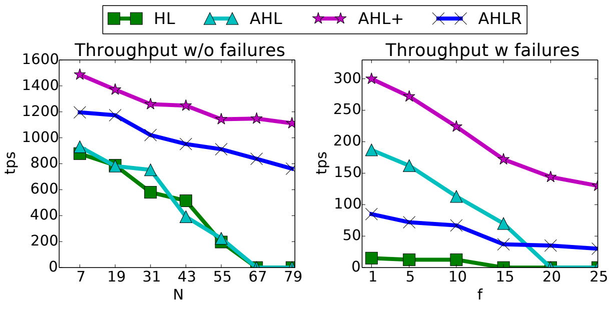


Figure 8: AHL+ performance on local cluster.

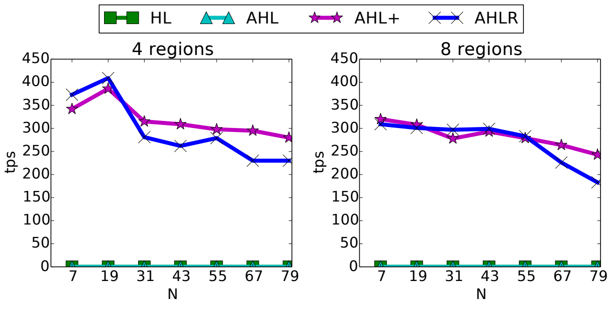


Figure 9: AHL+ performance on GCP (4 and 8 regions).

than $5\mu s$. We also measured the cost of remote attestation protocol, which is carried out between nodes of the same committee in order to verify that they are running the correct enclave. On our SGX-enabled processor, this protocol takes around $2ms$, but we note that it is executed only once per epoch, and its results can be cached.

The results reported in the following are averaged over ten independent runs. Due to space constraints, we focus on throughput performance in this section, and discuss other results in the Appendix.

7.1 Fault-scalable consensus

AHL+ vs. other variants. We compare the performance of AHL+ with the original PBFT protocol (denoted by HL), AHL and AHLR. Figure 8 and Figure 9 show the throughput with increasing number of nodes, N, on the local cluster and on GCP, when using KVStore benchmark with 10 clients. The performance with varying number of clients and fixed N is included in the Appendix.

AHL's throughput is similar to that of HL, but for the same N it tolerates more failures. Both HL and AHL show no throughput for $N > 67$ on the local cluster, and no throughput at all on GCP. We observe that these systems are live-locked when N increases, as they are stuck in the view-change phase. The number of view changes is reported in

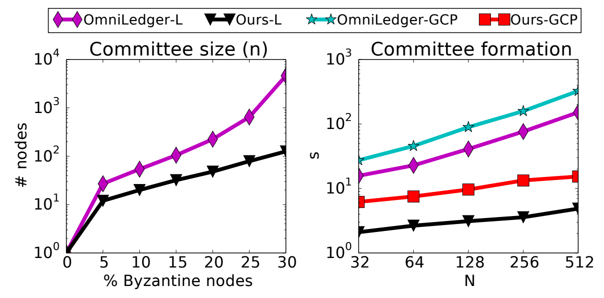


Figure 10: Evaluation of shard formation.

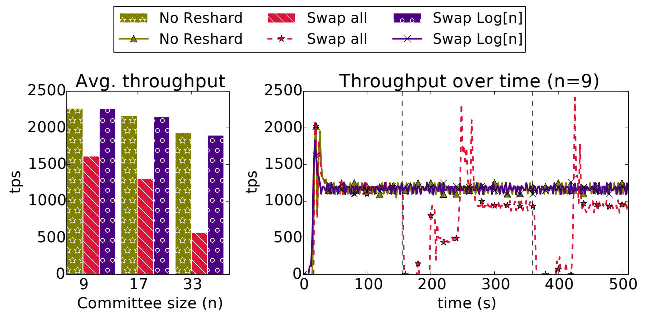


Figure 11: Performance during shard reconfiguration.

the Appendix. In contrast, both AHL+ and AHLR maintain throughputs above 700 transactions per second in the cluster and above 200 on GCP. Interestingly, AHL+ demonstrates consistently higher throughput than AHLR, even though the former has $O(N^2)$ communication overhead compared to $O(N)$ of the latter. Careful analysis of AHLR shows that the leader becomes the single point of failure. If the leader fails to collect and multicast the aggregate message before the time out, the system triggers the view change protocol which is expensive.

To understand the impact of Byzantine behavior on the overall performance, we simulate an attack in which the Byzantine nodes send conflicting messages (with different sequence numbers) to different nodes. Figure 8 (right) shows how the throughput deteriorates when the number of failures increases. We note that for a given f, HL runs with $N = 3f + 1$ nodes, whereas AHL, AHL+ and AHLR run with $N = 2f + 1$ nodes. Despite the lower throughputs than without failures, we observe a similar trend in which AHL+ outperforms the other protocols. On GCP with more than 1 zone, the Byzantine behavior causes all protocols to livelock.

7.2 Shard Formation

Figure 10 compares our shard formation protocol with OmniLedger's in terms of committee size and running time. With increasing Byzantine failures, OmniLedger needs exponentially larger committees. On the other hand, by leveraging AHL+, our protocol keeps the committees up to two orders of magnitude smaller.

We compare the cost of our distributed random number generation protocol with that of Randhound used in OmniLedger. We vary the network size from 32 to 512 nodes. On the local cluster, we oversubscribe each physical server by a factor of 8, running each node in a single-threaded virtual machine. On GCP, each node runs on an instance with 2 vCPUs. Recall that both protocols assume a synchronous network with the bounded delay Δ. We empirically derive Δ by measuring the maximum propagation delay in different network sizes for a 1KB message, then conservatively setting Δ to be 3× the measured value. On the local cluster, Δ ranges from 2 to 4.5s. On GCP, Δ ranges from 5.9 to 15s. We set $q = \log(N) - \log(\log(N))$, so that the communication overhead is $O(N \log(N))$ and $P_{repeat} < 2^{-11}$. For Randhound, we set $c = 16$ as suggested in [27]. Figure 10 shows that our protocol is up to 32× and 21× faster than RandHound on the local cluster and GCP, respectively. We attribute this gap to the difference in their communication complexity: $O(N \log(N))$ versus $O(Nc^2)$.

Shard Reconfiguration. Next, we analyze the performance of our system during epoch transition (or resharding). We consider the naive approach which *swaps all* nodes by first stopping them, assigning them to new shards based on a randomly generated permutation, and then starting them again. We compare it with our approach which swaps B nodes at a time. In our evaluation, we set B = $\log(n)$.

Figure 11 shows the throughput against the baseline where there is no resharding. We run the experiments on our local clusters with two shards, each containing up to 33 nodes. We perform the resharding twice, as depicted in Figure 11 (right). The naive approach shows a sharp drop in throughput when all nodes are restarted, followed by a period of up to 80s in which nodes discover their new peers, verify and synchronize their states. The subsequent spikes in throughput are due to nodes processing transactions from their backlog. In contrast, our approach allows the system to maintain the same throughput as the baseline.

7.3 Sharding performance

We first evaluate the performance of the coordination protocol by running many shards on the local cluster with $f = 1$. To saturate S shards, we use $4S$ clients, each maintaining 128 outstanding requests to the shard. Figure 12 (left) reports the throughput for Smallbank with network size of up to 36 nodes. The results for KVStore are similar and are included in the Appendix. When HL-based sharding is used (as in OmniLedger), there are up to 9 shards in total. Our sharding

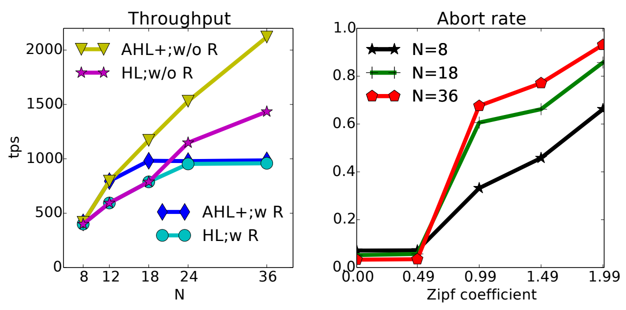


Figure 12: Sharding performance on local cluster with and without reference committee.

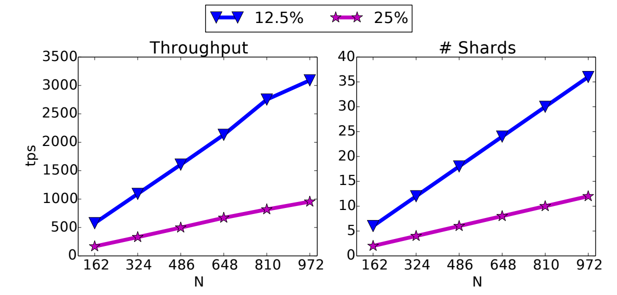


Figure 13: Sharding performance on GCP.

protocol uses AHL+, therefore allowing for up to 12 shards. It can be seen that the overall throughput of the system scales linearly with the number of shards. Furthermore, when the reference committee is involved, it becomes the bottleneck as the number of shards grows. We vary the skewness factor (Zipf coefficient) of the workload, and show the effect of contention on the overall throughput in Figure 12 (right). As expected, the abort rate increases with the Zipf value.

Finally, we evaluate the throughput scalability of our sharded blockchain system in a large-scale, realistic network environment. For this experiment, we run Smallbank without the reference committee, using up to 972 nodes and 432 clients spanning 8 regions on GCP. We consider two adversarial powers: 12.5% and 25% which are studied in other sharded blockchains [27, 33]. The former requires a committee size of 27 nodes, and the latter of 79 nodes to keep the probability of faulty committee below 2^{-20}. Figure 13 shows the overall throughput and the corresponding number of shards for the two adversarial powers. It can be seen that the throughput scales linearly with the number of shards. In particular, for 12.5% adversary we achieve over 3,000 transactions per second using 36 shards, which is expected to grow higher with more shards[2]. For 25% adversary, we observe a throughput of 954 transactions per second which is higher than that of OmniLedger with regular validations [27].

[2] We were unable to obtain enough resources to run with more shards at the time of writing.

8 RELATED WORKS

We have discussed three related sharded blockchains, namely Elastico, OmniLedger and RapidChain, extensively in the previous sections. Another related system is Chainspace [7] which proposes a sharding protocol for blockchain applications that are more general than cryptocurrencies. It allows smart contracts to assign nodes to shards, as opposed to using a distributed shard formation protocol. We do not consider Chainspace in this work, due to its complex transaction model, complex coordination protocol, and low throughput of only 300 transactions per second even with 10 shards.

Scaling blockchain with database techniques. Various works have exploited database techniques to improve aspects of the blockchain software stack other than the consensus layer. Forkbase [45] is the first storage designed for blockchains, supporting analytical queries at a much lower cost than the current key-value backends. Dickerson *et al.* [19] add concurrency to Ethereum execution engine by using software transaction memory primitives. We expect more works in improving the blockchain storage and execution engine. While orthogonal to ours, they can be combined to enable scalable blockchains with richer functionalities.

Off-chain scaling. Instead of directly improving blockchain components, another approach to scalability is to move as many transactions off the blockchain as possible. Such *off-chain* solutions allow users to execute transactions and reach consensus directly among each other, requiring minimal interaction with the blockchain. The blockchain is used only for disputes and settlements. Examples of off-chain solutions include payment channels [31] and state channels [38].

9 CONCLUSIONS

In this paper, we applied database sharding techniques to blockchains. We identified challenges that arise from the fundamental difference in failure models between traditional distributed databases and blockchain systems. We addressed them by leveraging TEEs to design fault-scalable consensus protocols and an efficient shard formation protocol. Furthermore, we proposed a coordination protocol for cross-shard transactions that supports general blockchain workloads. The coordination protocol employs a Byzantine fault-tolerant reference committee to guards against malicious coordinators. Finally, we conducted extensive, large-scale evaluation of our designs in realistic network settings, achieving over 3,000 transactions per second with many shards.

ACKNOWLEDGEMENT

This research has been supported by the Singapore Ministry of Education Academic Research Fund Tier-3 under Grant MOE2017-T3-1-007, and the National Research Foundation, Prime Minister's Office, Singapore under its Strategic Capability Research Centres Funding Initiative. We thank the anonymous reviewers their helpful feedback and insightful suggestions. All opinions and findings expressed in this work are those of the authors and do not necessarily reflect the views of any of the sponsors.

REFERENCES

[1] Intel SGX SDK for Linux. https://github.com/01org/linux-sgx.
[2] Proof of Elapsted Time. https://sawtooth.hyperledger.org/docs/core/releases/latest/introduction.html.
[3] Ripple. https://ripple.com.
[4] The Coco Framework. http://aka.ms/cocopaper.
[5] Trusted Computing Group. http://www.trustedcomputinggroup.org/.
[6] 2018. Intel Software Guard Extensions Developer Guide. https://download.01.org/intel-sgx/linux-1.7/docs/Intel_SGX_Developer_Guide.pdf.
[7] Mustafa Al-Bassam, Alberto Sonnino, Shehar Bano, Dave Hrycyszyn, and George Danezis. 2017. Chainspace: A Sharded Smart Contracts Platform. *arXiv preprint arXiv:1708.03778* (2017).
[8] Tiago Alves and Don Felton. 2004. *Trustzone: Integrated hardware and software security.* Technical Report. ARM.
[9] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. 2013. Innovative technology for CPU based attestation and sealing. In *HASP.*
[10] Johannes Behl, Tobias Distler, and Rüdiger Kapitza. 2017. Hybrids on Steroids: SGX-Based High Performance BFT. In *EuroSys.*
[11] Bitcoin Wiki. 2018. Scalability. en.bitcoin.it/wiki/scalability.
[12] Ferdinand Brasser, Urs Muller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. 2017. Software Grand Exposure: SGX Cache Attacks Are Practical. In *USENIX Security.*
[13] Vitalik Buterin. 2014. Ethereum: A next-generation smart contract and decentralized application platform. *https://github.com/ethereum/wiki/wiki/White-Paper* (2014).
[14] Miguel Castro. 2001. *Practical Byzantine Fault Tolerance.* Ph.D. Dissertation. Massachusetts Institute of Technolog.
[15] Miguel Castro, Barbara Liskov, et al. 1999. Practical Byzantine fault tolerance. In *OSDI.*
[16] Xiaoxin Chen, Tal Garfinkel, Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R.K. Ports. 2008. Overshadow: A Virtualization-Based Approach to Retrofitting Protection in Commodity Operating Systems. In *ASPLOS.*
[17] Byung-Gon Chun, Petros Maniatis, Scott Shenker, and John Kubiatowicz. 2007. Attested append-only memory: Making adversaries stick to their word. In *OSR.*
[18] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. https://eprint.iacr.org/2015/564.pdf.
[19] Thomas Dickerson, Paul Gazzillo, Maurice Herlihy, and Eric Koskinen. 2017. Adding Concurrency to Smart Contracts. In *PODC.*
[20] Tien Tuan Anh Dinh, Rui Liu, Meihui Zhang, Gang Chen, and Beng Chin Ooi. 2017. Untangling blockchain: a data processing view of blockchain systems. *TKDE* (2017).
[21] Tien Tuan Anh Dinh, Ji Wang, Gang Chen, Rui Liu, Beng Chin Ooi, and Kian-Lee Tan. 2017. BLOCKBENCH: A Framework for Analyzing Private Blockchains. In *SIGMOD.*
[22] James C. Corbett et al. 2012. Spanner: Google's Globally-Distributed Database. In *OSDI.*
[23] Fr8 Network. 2018. Blockchain enabled logistic. https://fr8.network.
[24] Hector Garcia-Molina. 2008. *Database systems: the complete book.*
[25] Arthur Gervais, Ghassan O Karame, Karl Wüst, Vasileios Glykantzis, Hubert Ritzdorf, and Srdjan Capkun. 2016. On the security and performance of proof of work blockchains. In *CCS.*
[26] Eleftherios Kokoris Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. 2016. Enhancing bitcoin security and performance with strong consistency via collective signing. In *USENIX Security.*
[27] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, and Bryan Ford. 2017. OmniLedger: A Secure, Scale-Out, Decentralized Ledger. *IACR Cryptology ePrint Archive* (2017).
[28] Jae Kwon. Tendermint: Consensus without mining. https://tendermint.com/static/docs/tendermint.pdf.
[29] Leslie Lamport et al. 2001. Paxos made simple. *ACM Sigact News* (2001).
[30] Dave Levin, John R Douceur, Jacob R Lorch, and Thomas Moscibroda. 2009. TrInc: Small Trusted Hardware for Large Distributed Systems. In *NSDI.*
[31] Lightning Network. 2018. Lightning network: scalable, instant Bitcoin/blockchain transactions. https://lightning.network.
[32] Shengyun Liu, Paolo Viotti, Christian Cachin, Vivien Quéma, and Marko Vukolic. 2016. XFT: Practical Fault Tolerance beyond Crashes. In *OSDI.*
[33] Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. 2016. A secure sharding protocol for open blockchains. In *CCS.*
[34] Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David Sommer, Arthur Gervais, Ari Juels, and Srdjan Capkun. 2017. ROTE: Rollback Protection for Trusted Execution. In *USENIX Security.*
[35] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. 2013. Innovative instructions and software model for isolated execution. In *HASP.*
[36] Medilot. 2018. Transforming healthcare for all. https://medilot.com.
[37] Silvio Micali. 2016. ALGORAND: the efficient and democratic ledger. *arXiv preprint arXiv:1607.01341* (2016).
[38] Andrew Miller, Iddo Bentov, Ranjit Kumaresan, Christopher Cordi, and Patrick McCorry. 2017. Sprites and state channels: payment networks that go faster than lightning. https://arxiv.org/pdf/1702.05812.pdf.
[39] Shuai Mu, Lamont Nelson, Wyatt Lloyd, and Jinyang Li. 2016. Consolidating Concurrency Control and Consensus for Commits under Conflicts. In *OSDI.*
[40] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system.
[41] Diego Ongaro and John K Ousterhout. 2014. In search of an understandable consensus algorithm. In *USENIX ATC.*
[42] Rafael Pass, Lior Seeman, and Abhi Shelat. 2017. Analysis of the blockchain protocol in asynchronous networks. In *EUROCRYPT.*
[43] Florian Tramer, Fan Zhang, Huang Lin, Jean-Pierre Hubaux, Ari Juels, and Elaine Shi. 2017. Sealed-glass proofs: Using transparent enclaves to prove and sell knowledge. In *EuroS&P.*
[44] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. 2018. FORESHADOW: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security.*
[45] Sheng Wang, Tien Tuan Anh Dinh, Qian Lin, Zhongle Xie, Meihui Zhang, Qingchao Cai, Gang Chen, Wanzeng Fu, Beng Chin Ooi, and Pingcheng Ruan. 2018. ForkBase: An Efficient Storage Engine for

Blockchain and Forkable Applications. In *VLDB*.

[46] Xinan Yan, Linguan Yang, Hongbo Zhang, Xiayue Charles Lin, Bernard Wong, Kenneth Salem, and Tim Brecht. 2018. Carousel: low-latency transaction processing for globally-distributed data. In *SIGMOD*.

[47] Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova. 2018. RapidChain: Scaling Blockchain via Full Sharding. In *CCS*.

[48] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. 2015. Building Consistent Transactions with Inconsistent Replication. In *SOSP*.

A DEFENSES AGAINST ROLLBACK ATTACKS

Data sealing mechanism enables enclaves to save their states to persistent storage, allowing them to resume their operations upon recovery. However, enclave recovery is vulnerable rollback attack [34].

AHL+. The adversary can cause the enclave of AHL+ to restart, and supply it with a *stale* log heads upon its resumption. The enclave resuming with stale log heads "forgets" all messages appended after the stale log heads, allowing the adversary to equivocate.

Denote by H the sequence number of the last consensus message the enclave processes prior to its restart. The recovering enclave must not accept any message with a sequence number lower than or equal to H. We derive an estimation procedure that allows the resuming enclave to estimate an upper bound, H_M, on the latest sequence number it would have observed if it were not crashed. The goal of this estimation is to guarantee that $H_M \geq H$, ensuring protocol's safety.

The enclave starts the estimation procedure by querying all its peers for the sequence number of their last checkpoint, denoted by ckp. The resuming enclave uses the responses to select ckp_M, which is a value ckp it receives from one node j such that there are f replicas other than j reporting values less than or equal to ckp_M. It then sets the value H_M to $H_M = L + ckp_M$ where L is a preset difference between the node's high and low watermarks. The test against ckp responses of f other replicas ensures that ckp_M is greater than the sequence number of any stable checkpoint the resuming enclave may have; otherwise, there must be at least f ckp responses that are larger than ckp_M, which is not possible due to quorum intersection.

The resuming enclave will not append any message to its logs until it is *fully recovered*. This effectively refrains its host node from sending any message or processing any request, for the node cannot obtain the proof of append operation generated by the enclave. The enclave is fully recovered only after it is presented with a correct stable checkpoint with a sequence number greater than or equal to H_M. At this point, it starts accepting append operations, and the host node can actively participate in the protocol. Since H_M is an upper bound on the sequence number the AHL+ enclave would observe had it not been crashed, and that the host node cannot send any message with a sequence number lower than H_M once its enclave is restarted, the protocol is safe from equivocation.

RandomnessBeacon. The random values `q` and `rnd` are bound to the epoch number e and a counter v to prevent the adversary from selectively discarding the enclave's output to bias the randomness. These values, nonetheless, are stored in the enclave's volatile memory. The adversary may attempt to restart the enclave and invoke it using the same epoch number e to collect different values of `q` and `rnd`. Fortunately, the adversary only has a window of Δ from the beginning of epoch e to bias its `q` and `rnd` in that same epoch (after Δ, nodes have already locked the value of `rnd` used in epoch e). Thus, to prevent the adversary from restarting the enclave to bias `q` and `rnd`, it suffices to bar the enclave from issuing these two random values for any input $e \neq 0$ for a duration of Δ since its instantiation. The genesis epoch requires a more subtle set-up wherein participants are forced to not restart their enclaves during that first epoch. This can be realized by involving the use of CPU's monotonic-counter. Such process needs to be conducted only once at the system's bootstrap.

B PROBABILITY OF CROSS-SHARD TRANSACTIONS

We examine the probability that a transaction is cross-shard (i.e., it affects multiple shards' states at the same time). Consider a d-argument transaction tx that affects the values (states) of d different arguments. Without loss of generality, let us assume that arguments are mapped to shards uniformly at random, based on the randomness provided by a cryptographic hash function applied on the arguments. Let k be the total number of shards formed in the system. The probability that the transaction tx affects the states of exactly $x \leq min(d,k)$ shards can be calculated based on the multinomial distribution as follows:

$$\prod_{i=1}^{x-1} \frac{k-i}{k} \sum_{p_1+p_2+p_x=d-x} \prod_{j=1}^{x} (\frac{j}{k})^{p_j} \quad (3)$$

While OmniLedger and RapidChain give a similar calculation, they only consider a specific type of UTXO transactions whose outputs are all managed by a single output committee. Unfortunately, such calculation does not extend to UTXO transactions whose outputs belong to separate committees, let alone non-UTXO distributed transactions.

C ADDITIONAL EVALUATION RESULTS

This section provides additional results to those discussed in Section 7. First, the latency among the 8 GPC regions used in

Table 2: Latency (ms) between different regions on Google Cloud Platform.

Zone	us-west1-b	us-west2-a	us-east1-b	us-east4-b	asia-east1-b	asia-southeast1-b	europe-west1-b	europe-west2-a
us-west1-b	0.0	24.7	66.7	59.0	120.2	150.8	138.9	132.7
us-west2-a	24.7	0.0	62.9	60.5	129.5	160.5	140.4	136.1
us-east1-b	66.7	62.9	0.0	12.7	183.8	216.6	93.1	88.2
us-east4-b	59.1	60.4	12.7	0.0	176.6	208.4	81.9	75.6
asia-east1-b	118.7	129.5	184.9	176.6	0.0	50.5	255.5	252.5
asia-southeast1-b	150.8	160.5	216.7	208.3	50.6	0.0	288.8	283.8
europe-west1-b	138.9	140.5	93.2	81.8	255.7	288.7	0.0	7.1
europe-west2-a	132.1	134.9	88.1	76.6	252.1	283.9	7.1	0.0

Table 3: Runtime costs of enclave operations (excluding enclave switching cost which is roughly 2.7μs).

Operations	Time (μs)
ECDSA Signing	458.4(±0.4)
ECDSA Verfication	844.2(±0.8)
SHA256	2.5(±0.1)
AHL Append	465.3(±0.8)
AHLR Message Aggregation ($f = 8$)	8031.2(±2.3)
RandomnessBeacon	482.2(±0.5)

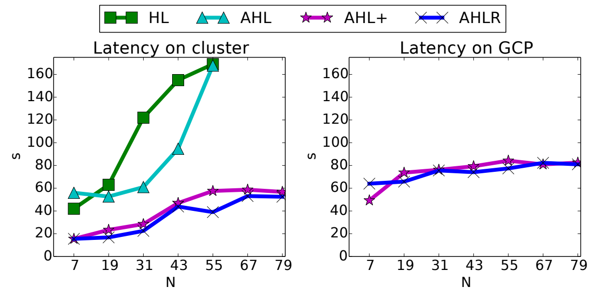


Figure 14: AHL+ latency on local cluster and GCP.

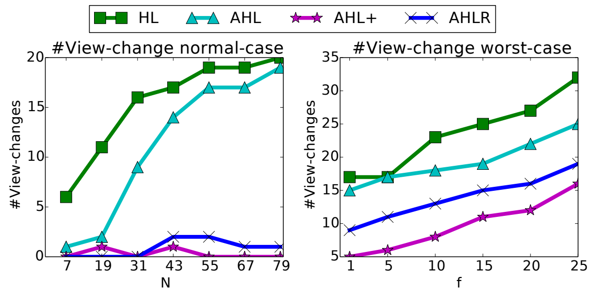


Figure 15: # View-changes of AHL+ on local cluster.

our experiments is listed in Table 2. Figure 14 and 15 show the latency and number of view-changes in different consensus protocols as the network size increases. Figure 16 presents the cost breakdown for a block of transactions, showing that the cost of transaction execution is an order of magnitude smaller than that of consensus. Figure 19 and 20 demonstrate AHL+'s throughput with varying numbers of clients on the cluster and on GCP. Figure 17 compares the sharding throughput under KVStore versus Smallbank.

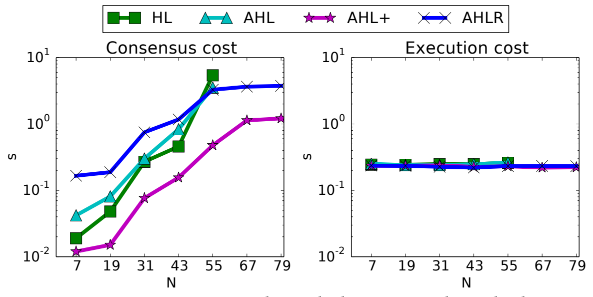


Figure 16: AHL+ cost breakdown on local cluster.

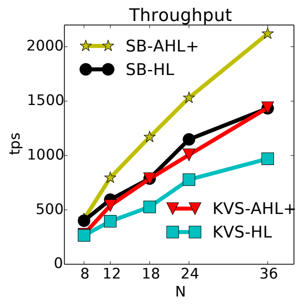


Figure 17: Sharding with KVStore vs. Smallbank.

C.1 Optimization breakdown

We examine how each optimization presented in Section 4.1 contributes to the final performance. Figure 18 shows, against the baseline of original PBFT implementation (HL), the effect of adding trusted hardware (AHL), optimization 1 (separating message queues), optimization 2 (removing request), and optimization 3 (message aggregation at the leader). The experiments, which are run on our local cluster with 10 clients, show that optimization 2 adds the most benefits when there is no failure, whereas optimization 1 is the best under Byzantine failures. This explains why AHL+, which incorporates both optimizations, has the best performance.

C.2 PoET+ vs. PoET

We evaluate the performance of PoET and PoET+based on Hyperledger Sawtooth v0.8 implementation. On the local cluster, we run 4 nodes on each physical server, and impose 50 Mbps bandwidth limit and 100ms latency on the network links. On GCP, we run each node on an instance with 2

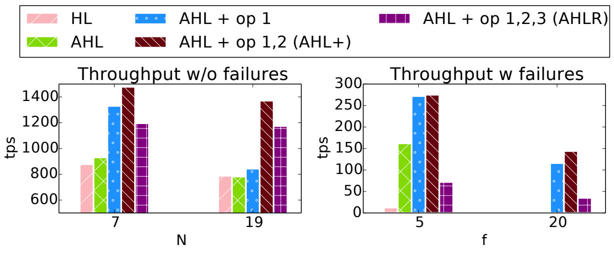


Figure 18: Effect of optimizations on throughput.

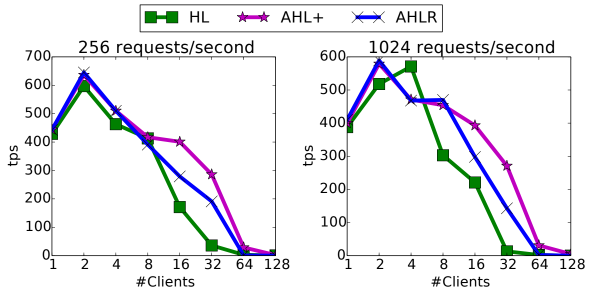


Figure 19: Throughput on GCP.

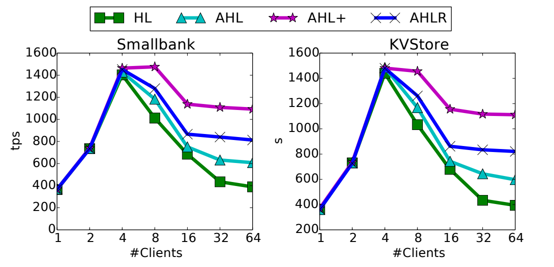


Figure 20: Throughput on local cluster.

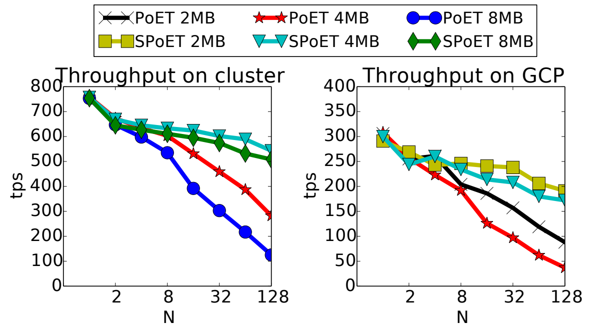


Figure 21: PoET and PoET+ performance.

vCPUs, and the nodes are distributed over 8 regions. We set $l = \frac{\log(N)}{2}$, reducing the effective network size to $\sqrt{N}$.

The block size is varied from 2MB to 8MB, and the block time from $12s$ to $24s$. As N increases, block propagation time, which depends on block time and block size, increases and leads to higher stale block rate and lower throughput, as shown in Figure 21 and 22. PoET+ maintains up to 4× higher throughput because it reduces the stale block rate significantly from 15% to 3% with $N = 128$.

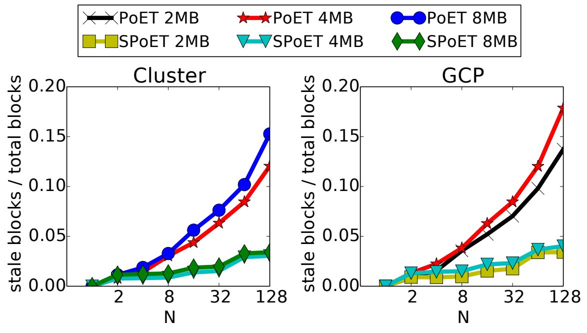


Figure 22: PoET and PoET+'s stale block rate.

C.3 Comparison of BFT protocols

Figure 2 compares the performance of four popular BFT implementations in blockchains, namely PBFT, Tendermint, IBFT and Raft (both from Quorum). For this comparison, we used the key-value benchmark in BLOCKBENCH. For Tendermint, we used the provided `tm-bench` tool that benchmarks a simple key-value application.

We examined throughputs with varying network size and workload. In both settings, PBFT outperforms the alternatives, except at $N = 1$ due to HyperLedger's limit on REST request rate (around 400 requests per second). We attribute these results to the design of Tendermint and IBFT, and implementation of Raft in Quorum.

Unlike PBFT that relies on a stable leader to drive consensus, Tendermint and IBFT rotate leaders in a round-robin fashion, and may take many rounds to agree on a block. Safety is achieved via a locking mechanism in which a node locks on a block after it receives a quorum of Prepare messages. Once locked, it does not vote for other blocks. For liveness, the lock is released when there are a quorum of Prepare messages for another block in a later round. IBFT suffers from deadlock because its locks are not released properly. In Tendermint, a new block can only be proposed when the previous one is finalized. This lock-step execution of consensus is different from HyperLedger's PBFT where a leader pipelines many blocks before the first block is finalized. In particular, a node can vote on many blocks at the same time, if the blocks are assigned consecutive sequence numbers. Such pipelined execution extracts more concurrency from the consensus implementation, achieving higher throughput.

Raft has a higher fault tolerance threshold than PBFT ($\frac{N}{2}$ vs. $\frac{N}{3}$), and is expected to have higher throughput. But we observed a lower performance for a Raft-based blockchain (Quorum), than a PBFT blockchain (HyperLedger). This is due the lack of pipelined consensus execution in Quorum, which only constructs the next block once the previous block has been finalized. As a result, consensus happens in lockstep and the overall throughput suffers.