Instachain: Breaking the Sharding Limits *via* Adjustable Quorums

Mustafa Safa Ozdayi*
The University of Texas at Dallas
mustafa.ozdayi@utdallas.edu

Yue Guo*
J.P. Morgan Research
yueguo@cs.cornell.edu

Mahdi Zamani
Visa Research
mzamani@visa.com

***Abstract*—Sharding is a key approach to scaling the performance of on-chain transactions: the network is randomly partitioned into smaller groups of validator nodes, known as *shards*, each growing a disjoint ledger of transactions via state-machine replication (SMR) in parallel to other shards. As SMR protocols typically incur a quadratic message complexity in the network size, shards can process transactions significantly faster than the entire network. On the downside, shards cannot be made arbitrarily small due to the exponentially-increasing probability that malicious nodes take over a sufficient majority in small shards, compromising the security of SMR. In practice, this dictates relatively large shards with hundreds of nodes each, significantly limiting the scalability benefits of sharding.**

In this paper, we propose Instachain, a novel sharding approach that breaks the scalability limits of sharding by reducing the shard size to significantly-smaller numbers than was previously considered possible. We achieve this by relaxing the liveness property for some of the shards while still preserving the safety property across all shards. To do this, we carefully adjust the quorum size parameter of the intra-shard SMR protocol to achieve maximum parallelism across all shards without compromising security. In addition, Instachain is the first sharding protocol to adopt the stateless blockchain model in shards, which in conjunction with a novel cross-shard verification technique allows the protocol to efficiently prevent double-spending attempts across significantly-more shards than previous work.

I. Introduction

One of the major barriers to the mass adoption of cryptocurrencies and blockchain protocols is their scalability limitations, primarily reflected by the rate at which they can process transactions. This is mainly imposed by the need for replicating data among many geographically-scattered nodes to increase resiliency to both faults and centralized manipulations.

In the past decade, several scalability approaches have been proposed which generally fall into two categories: on-chain and off-chain scalability. The former usually aims at improving the underlying consensus mechanism, adding extra trust assumptions, and/or delegating the processing task to only a small subsets of validator nodes (*aka*, *committees*). The latter approach minimizes the use of the blockchain itself by allowing parties to transact via direct, point-to-point communication, and interact with the blockchain only occasionally to settle disputes or to deposit/withdraw funds.

*This work was done when the author was an intern at Visa Research.
A preliminary version of this paper was published in [1].

Both scaling approaches have seen significant progress in the past decade, though on-chain scaling is often lagging behind in numbers primarily due to the complexities of the underlying consensus protocol in large networks. Nevertheless, off-chain mechanisms usually achieve better scalability only when certain assumptions are in place such as on-chain collateral deposits locked by each participant, or cost amortization over many (often low-value) transactions, e.g., in micropayments [2]. For applications that require instant settlement of high-liability, high-value transactions, such as real-time payments [3], on-chain scaling remains our main hope.

Slower progress in on-chain scaling is primarily due to the quadratic overhead of replication of the transaction ledger across the nodes. Luckily, there has been significant progress in sharding techniques [4], [5], [6], where the general approach is to partition the network into groups of nodes, called shards, each running a Byzantine fault tolerant (BFT) state-machine replication (SMR) protocol to build their local blockchain in parallel. As new nodes join the system, new shards are created, and hence, the throughput of the system increases with its size, i.e., the system scales.

Existing sharding protocols sample their shards in such a way that all shards satisfy the two properties of SMR: *safety* and *liveness*. Informally, the former ensures all honest nodes within a shard maintain the same state, while the latter guarantees all (valid) transactions assigned to a shard are eventually processed. To achieve both properties in a shard, the fraction of corrupt nodes in the shard must be strictly less than $1/2$ if the communication network is synchronous (i.e., messages are always delivered within a known, fixed time-bound), and $1/3$ in a partially-synchronous networks (i.e., messages are delivered within an unknown, fixed time-bound) [7].

The limit on the fraction of corrupt nodes in each shard can only be satisfied if a sufficient number of nodes are sampled randomly from the pool of all nodes when shards are being created. For example, RapidChain [6] and OmniLedger [5] require shards of least 250 and 600 nodes to ensure sufficient security in the synchronous and the partially-synchronous settings, respectively. Choosing smaller shards simply means an exponentially-higher probability of failure in the guarantee to provide *both* safety and liveness in *all* shards. Given the communication and computation complexities of SMR protocols, we believe these requirements impose a major barrier to the scalability benefits of sharding.

In this paper, we propose to relax one of the two properties in some shards in favor of smaller shards. Since safety is a critical requirement of blockchain protocols in protection against double spending, we choose to relax the liveness property in some shards. We propose Instachain, a sharding protocol that reduces shard sizes to 50-100 nodes (down from 200-250 in RapidChain [6]) in the synchronous setting with better failure probability and higher parallelism (measured as the number of shards that can make progress in growing their chains) than RapidChain. We achieve this by fine-tuning the quorum size parameter of the intra-shard SMR to relax the liveness requirement in some shards, and yet, still ensure safety in all shards. By developing efficient techniques to handle shards that cannot make progress, we prove the safety and the liveness of the overall protocol in a synchronous network with an adaptive adversary.

In sharded blockchains, having smaller shards means more shards, and that increases the likelihood of cross-shard transactions, i.e., those that depend on inputs from other shards and their atomic executions entail updating the state of the other shards. Therefore, executing cross-shard transactions is often considered an expensive operation [5], [6]. Moreover, one needs to also consider the overhead of detecting shards that cannot make progress and reassign their workload to other shards. In Instachain, we employ certain optimizations to make cross-shard transactions and state migration less expensive to be able to justify the benefits of higher parallelism obtained from smaller shards. We, however, leave it to the future work to experiment if the benefits of smaller shards outweigh their costs in practice.

A. *Our Contributions*

Our paper makes the following contributions:

- **Sharding with Adjustable Quorums.** We introduce the idea of *adjusting the quorum size* in the context of sharded, fault-tolerant distributed systems. The technique allows us to relax the liveness property of SMR for some of the shards while ensuring safety in all shards. This reduces shard sizes significantly, improving the shard throughput and the degree of parallelism as a result. We develop a sharded blockchain protocol, Instachain, that uses adjustable quorums and scales beyond existing protocols by creating smaller shards with similar failure probabilities.
- **Stateless Sharded Blockchains.** We adopt the so-called *stateless blockchain* model in the sharded setting for the first time. Our primary motivation to do so is to tackle some unique problems that arise from safe-only shards in our protocol. Nevertheless, we observe adopting the new model brings certain other benefits to our protocol. For example, it makes the reconfiguration phase of the sharded blockchains more efficient by compressing the blockchain state significantly.
- **Lock-free, Client-Driven Cross-Shard Transactions.** We describe a new cross-shard transaction verification protocol in the UTXO model that minimizes cross-shard communication by being client-driven, and removes the storage overhead of maintaining locks on transactions by avoiding strict atomicity. Our cross-shard protocol still maintains consistency with an embedded refund mechanism in scenarios where atomicity is violated. Finally, unlike previous cross-shard protocols such as in [5] and [6], our protocol is resilient to replay attacks [8] by an adversary who attempts to double spend money by presenting a cross-shard transaction to the same shard multiple times.

B. *Protocol Overview*

Consider N nodes connected via a peer-to-peer network (similar to that of Bitcoin [9]), where each node is connected to a constant number of nodes. The nodes can communicate with each other by gossiping a message to either the entire network or within a subset of nodes a shard. Instachain proceeds in fixed time periods, known as *epochs*. The first epoch starts by executing a one-time bootstrapping protocol that allows all nodes to agree on a sequence of random bits, known as the *epoch randomness*. Epoch randomness is then used by each node to learn (1) a random assignment of all nodes to shards, and (2) the sequence of leaders who later drive the intra-shard SMR. Once the shards are formed, the members of each shard run a synchronous BFT SMR protocol to construct a sequence of blocks of transactions, i.e., the blockchain. Further, at the end of each epoch, a fresh randomness is generated. This allows us to periodically reconfigure shards by shuffling nodes across them to prevent an adaptive adversary from gaining control of a shard.

Safety and Liveness. As discussed before, an SMR protocol has to satisfy two properties: safety and liveness. Safety says that all honest nodes must maintain the same state, and liveness says that valid transactions are eventually processed. In a non-sharded setting, it is trivial to satisfy either only the safety or only the liveness property. Nodes can easily satisfy safety by doing nothing. Given that all nodes start on the same state, they cannot possibly end up in different states after "doing" nothing. To satisfy liveness, nodes can simply process transactions independently. However, in this case, there is no consistent replication across the nodes. So, we observe that, relaxing either the safety, or the liveness does not give us a useful protocol in a non-sharded setting.

Now, consider a sharded setting where each shard satisfies safety but some do not satisfy liveness. We argue that we can guarantee the safety of such a setting as each shard satisfies safety (formalized in Theorem 4). Moreover, this setting exhibits liveness as some shards satisfy liveness (formalized in Theorem 5). Although relaxing liveness on some shards introduces new problems to tackle, such as handling the

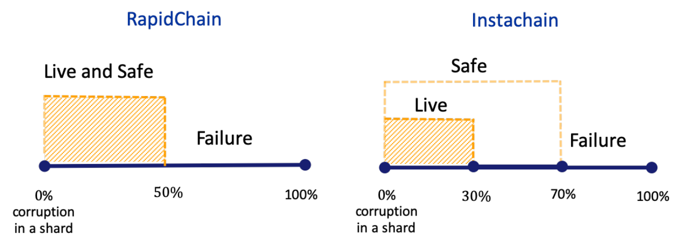


Fig. 1: Comparison of RapidChain and Instachain in a network of size $N = 2000$ with $F = 33\%$ resiliency. In this setting, each shard of RapidChain ensures both safety and liveness up to 50% shard corruption, where as each shard in Instachain can be configured to ensure liveness up to 30% corruption, and safety up to 70% corruption. Briefly, with a shard size of 200, RapidChain creates 10 shards where the failure probability of a shard is $1.9 \cdot 10^{-7}$. On the other hand, Instachain can create 40 shards of 50 nodes, and ensure a shard failure probability of $8.6 \cdot 10^{-8}$. Further, the probability of having an active shard is 0.37 for Instachain. This means the expected number of active shards is about $40 \cdot 0.37 = 14.8$ until the protocol fails.

transactions that are routed to safe-only shards, we show that we can solve these problems efficiently. As we quantify later in Section VI, relaxing liveness allows us to scale the protocol further by decreasing the shard sizes, and consequently, increasing the degree of parallelism.

Relaxing Liveness by Adjusting Quorum Size. For an SMR protocol, the quorum size q is the minimum number of honest nodes to ensure both safety and liveness. To relax liveness on some shards, we design a novel SMR protocol by extending Synch HotStuff [10] (Section IV-C). Briefly, our novel SMR protocol ensures the following a shard S,

- S satisfies both safety and liveness if the number of honest nodes in it are greater or equal to q. We say S is an **active shard** in this case.
- S satisfies neither safety nor liveness if the number of corrupt nodes in it are greater or equal to q. We say S is a **corrupt shard** in this case.
- S satisfies only safety if it is neither active nor corrupt. We say S is a **safe shard** in this case.

Given that, we can adjust the quorum size, *instead of increasing the shard size*, to obtain a sufficiently low failure probability. Figure 1 illustrates the benefit of this approach under an example setting.

Handling Safe Shards. A safe shard only ensures safety, that is, honest nodes always maintain the same shard state, but the shard is not guaranteed to make progress, as the corrupt nodes holding the majority may stay refuse to participate in the protocol. We address this problem by utilizing the honest nodes within safe shards, and a distinguished shard called *the reference shard*. The reference shard is ensured to consist of an honest majority with high probability (therefore, larger in size than other shards), hence it satisfies both safety and liveness. Briefly, the reference shard is tasked with periodically reconfiguring shards between epochs (in contrast to processing transactions) as well as maintaining and updating two tables: a *node-to-shard table*, that shows what node is assigned to which shard, and a *tx-to-shard table*, which specifies how transactions are assigned to shards. By updating, and broadcasting these tables, the reference shard can *close* safe shards, and redirect transactions that are assigned to closed shards to one of the running shards within an epoch.

Let us make concrete what problems safe shards can cause, and give an overview of our solutions. First, let S be a safe shard where the corrupt nodes in it stay inactive. After a sufficiently long time, honest nodes in S are going to detect the lack of liveness. Particularly, they are going to observe that they cannot process transactions, and they cannot replace a shard leader who fails to produce valid block proposals. After detecting the lack of liveness, honest nodes will *complain* to the reference shard. Upon receiving a sufficient number of complaints from S, the reference shard is going to update its tables: it reassigns every node of S to a special symbol $\perp$ in the node-to-shard table, to indicate the shard is closed for the ongoing epoch, and reassigns transactions of S to one of the running shards in the tx-to-shard table Consequently, transactions are not going to be routed to S anymore, and (honest) nodes of S stop participating in intra-shard SMR. This effectively closes S for the ongoing epoch[1]

A more tricky situation arises in the following case: the corrupt nodes in shard S can participate honestly in intra-shard SMR for some time before becoming inactive. This case differs from the previous one because the shard has processed some transactions and generated some UTXO before becoming inactive. Suppose S' takes over transactions of S after reassignment. To process transactions on behalf of S, S' has to know the latest state of S. In other words, we have to transfer the state across shards whenever a shard with non-empty UTXO set is closed. To solve this problem efficiently, we adopt the *stateless blockchain* model which uses cryptographic accumulators to maintain the blockchain state succinctly. Consequently, S would be able to transfer its latest state efficiently, and *provably* via the honest nodes in it to S'. S' will then be able to process transactions on behalf of S for the ongoing epoch.

II. Background & Related Work

A. Byzantine Fault Tolerant State-Machine Replication (BFT SMR)

In BFT SMR, a group of nodes want to agree on a sequence of transactions without the help of a third party, even when some of the nodes can deviate from the protocol arbitrarily (i.e., can be byzantine). The correctness of an SMR protocol is evaluated on two properties: *safety* and *liveness*. Safety guarantees that all honest nodes process the same sequence

[1]It is worth noting that, our protocol ensures an adversary cannot shutdown an active shard due to lack of sufficient majority.

of transactions, and hence maintain the same state. Liveness ensures that all correct transactions are eventually processed. The solvability of SMR depends on the maximum number of faulty nodes which varies on the network synchrony assumptions. Under a synchronous network assumption, we require an honest majority of nodes [7]. On the other hand, under a partially-synchronous network, the fraction of corrupt nodes should be strictly bounded from above by $1/3$.

B. Blockchain Sharding

Sharded blockchain protocols can increase their transaction processing power with the number of participants joining the network. This is done by allowing multiple shards of nodes process incoming transactions in parallel.

Several existing works explore the construction of sharded blockchain protocols. Luu *et al.* [4] propose Elastico, the first sharded consensus protocol for public blockchains. In every consensus epoch, each participant solves a PoW puzzle based on an epoch randomness obtained from the last state of the blockchain. The PoW's least-significant bits are used to determine the committees which coordinate with each other to process transactions.

Kokoris-Kogias *et al.* [5] propose OmniLedger, a sharded distributed ledger protocol that attempts to fix some of the issues of Elastico. Assuming a slowly-adaptive adversary that can corrupt up to a 1/4 fraction of the nodes at the beginning of each epoch, the protocol runs a global reconfiguration protocol at every epoch (about once a day) to allow new participants to join the protocol. OmniLedger generates identities and assigns participants to committees using a slow identity blockchain protocol that assumes synchronous channels. A fresh randomness is generated in each epoch using a bias-resistant random generation protocol that relies on a verifiable random function (VRF) [11] for unpredictable leader election in a way similar to the lottery algorithm of Algorand [12].

Zamani *et al.* [6] propose RapidChain, a sharded blockchain protocol in the 1/3 corruption setting that can achieve complete sharding of the communication, computation, and storage overhead of processing transactions without assuming any trusted setup. RapidChain employs an intra-committee consensus algorithm that can achieve high throughput via block pipelining, a gossiping protocol for large blocks, and a reconfiguration mechanism based on the Cuckoo rule [13].

Finally, the concurrent and independent work of David *et al.* [14] also explores liveness-safety dichotomy, and relaxes liveness on some shards as we do. Our work primarily differs from theirs by the setting we consider: we focus on the synchronous setting with adaptive adversaries, while their protocol is tailored towards the partially-synchronous setting with static adversaries. Further, their way of handling safe shards requires all the network to communicate with each other, which can become a bottleneck as the network grows, whereas Instachain solves this by having a slightly larger reference shard than regular shards. As noted before, we also adapt the stateless model to the sharded setting for the first time, and present a novel cross-shard UTXO protocol to tackle replay attacks. Both of these contributions can be of independent interest.

C. Accumulators and Stateless Blockchains

An accumulator is a binding commitment on a set. Informally, an accumulator provides a succinct representation of a set that can be queried for membership and possibly for non-membership. Accumulators have recently gained popularity in the context of blockchains due to the so-called "stateless" blockchain model [15], [16], [17]. In this model, validators maintain an accumulator over the UTXO or account sets rather than maintaining the sets explicitly. It is up to the clients to maintain the explicit set by listening to the network and providing membership proofs on their transactions to spend their UTXOs or balances. In Instachain, we assume an accumulator scheme which provides the following functionalities, such as RSA accumulators as described in [17].

- $\texttt{Add}(A, x)$. Given an accumulator A and an element x, add x to A.
- $\texttt{Delete}(A, x)$ Given an accumulator A and an element x, delete x from A assuming it exists in A.
- $\texttt{GenMemWitness}(U, x)$ Given a set U and an element $x \in S$, generate a membership witness for x.
- $\texttt{GenNonMemWitness}(U, x)$ Given a set U and an element $x \notin U$, generate a non-membership witness for x.
- $\texttt{VerMem}(A, x, w)$ Given an accumulator A which includes elements of a set U, an element x and a membership witness w, returns 1 if $w = \texttt{GenMemWitness}(U, x)$ and 0 otherwise.
- $\texttt{VerNonMem}(A, x, w)$ Given an accumulator A which includes elements of a set U, an element x and a non-membership witness w, returns 1 if $w = \texttt{GenNonMemWitness}(U, x)$ and 0 otherwise.

The term *stateless blockchain* refers to a blockchain design where nodes can process transactions without requiring to store and maintain the explicit blockchain state. This reduces the storage requirements for nodes as well as to make it easier for new nodes to join the network.

In stateful blockchains, a node has to store the explicit blockchain state which can simply be constructed from processing blocks in their respective orders. Whenever a new node joins the network, it has to synchronize his state with the rest of the network by fetching the latest state from existing nodes. The state could potentially be very large, making this synchronization process time consuming. Further, as state gets larger, some nodes might leave the network as they cannot keep up with the storage requirements.

With stateless blockchains, nodes only have to maintain an accumulator over the blockchain state rather than explicitly maintaining the state itself. Given the size of an accumulator is very small (e.g., only a single group element with RSA accumulators [17]), this effectively alleviates the storage problem of nodes. Further, it is sufficient for a new node to get only the latest accumulator before it can participate in the network.

A trade-off we face in this model is increased transactions size. Briefly, each transaction should carry a *membership proof* for each of its inputs which attests that the input is contained within the accumulator (i.e., it is unspent). To generate such proofs, clients have to know the explicit blockchain state. To do so, they can either listen to the network for block broadcasts and maintain a copy by themselves, or rely on a "semi-trusted" entity that maintains a full copy of the state on behalf of the client and is always "available"[1].

To make things concrete, consider a UTXO-based blockchain with a UTXO set U at some point in time. At this time, nodes only have the accumulator computed over U. Clients, on the other hand, know U explicitly and submit transactions with membership proofs generated with respect to U on their transactions. Suppose that a validator node P is selected by the network to propose the next block. P selects a block of transactions from its memory, verifies their membership proofs with respect to U, and updates the accumulator accordingly. It then broadcasts the explicit block as well as the latest accumulator state (e.g., attaching it to block header). Other validators who receive this block verify the accumulator state by processing transactions contained in the block, update their accumulators, and simply discard the block contents.

III. Model and Problem Definition

Network Model. We consider a peer-to-peer network with N nodes who establish identities (i.e., public/private keys) through a Sybil-resistant identity generation mechanism such as that of [18], which requires every node to solve a computationally-hard puzzle on their locally-generated identities (i.e., public keys) verified by all other nodes. Without loss of generality and similar to most hybrid blockchain protocols [19], [20], [4], [5], we assume all participants in our consensus protocol have equivalent computational resources.

We assume all messages sent in the network are authenticated with the sender's private key. The messages are propagated through a synchronous gossip protocol [21] which guarantees that there is known upper bound Δ on the message delivery, i.e., when an honest node r_1 sends a message m to another honest node r_2 at time t, r_2 receives m by $t + \Delta$. However, the order of messages are not necessarily preserved, i.e., given r_1 first sends m_1 and then m_2, r_2 might receive m_2 before m_1. This is the standard synchronous model adopted by most permissionless protocols [4], [22], [5], [23].

[1]Note that, we trust this entity only for availability, but not for validity, i.e., such entity cannot generate membership proofs on spent transactions etc.

Threat Model. We consider a probabilistic polynomial-time Byzantine adversary who controls $F < 1/3$ of the nodes at any time. The corrupt nodes may not only collude with each other but can also deviate from the protocol in any arbitrary manner, e.g., by sending invalid or inconsistent messages, by remaining silent etc. Similar to most sharded protocols [19], [24], [20], [5], [23], we assume the adversary is *epoch adaptive* (*aka slowly adaptive*). It can select the set of corrupt nodes at the beginning of the protocol and/or between each epoch but cannot change this set within an epoch, i.e., it cannot choose which node to corrupt after nodes are shuffled between shards. Nodes may disconnect from the network during an epoch or between two epochs for any reason, such as internal failure or network jitter. However, at any moment, more than $2/3$ of the nodes, and hence computational resources, belong to honest participants that are online (i.e., respond within the network time bound). Finally, our protocol does not rely on any public-key infrastructure or any secure broadcast channel, but assumes the existence of a random oracle needed for collision-resistant hash function.

Problem Definition. We assume a set of transactions are sent to the network by a set of clients who are external to the protocol. Nodes in the network are grouped into disjoint shards. Each shard batches transactions into *blocks*, and processes them by an intra-shard SMR protocol. We say a shard commits to a block if every honest node within the shard commits to the block, and we say a shard commits to a transaction if the shard commits a block containing the transaction. Formally, our protocol guarantees the following properties.

Theorem 1 (Intra-shard Safety)**.** *Let S be a safe, or a super-honest shard. Given that, if an honest node in S commits a block B_k at height k, then every honest node in S eventually commits B_k at height k.*

Theorem 2 (Intra-shard Liveness)**.** *If a valid transaction T is assigned to a super-honest shard S, T is eventually committed by S.*

Theorem 3 (Provable Commits)**.** *Every committed block B_k of a shard S carries a commit-proof $\texttt{COM}_{B_k,S}$ which attests that B_k is committed by S.*

Theorem 4 (Inter-shard Safety)**.** *Once a shard commits a transaction T, no shard can commit to a transaction T' such that T and T' spend the same output, i.e., double spending cannot happen.*

Theorem 5 (Inter-shard Liveness)**.** *When a valid transaction T is assigned to a shard S, either S commits T, or eventually S is closed and T is committed by another shard S', i.e., valid transactions are eventually processed by a shard.*

IV. Instachain Protocol

In this section, we briefly remind the execution flow of our protocol, and then describe each component thoroughly.

Overview of the Flow of Execution. Our protocol proceeds in epochs where nodes have access to a fresh randomness at the beginning of each epoch. Using the randomness, nodes are first assigned to the shards. In the very first epoch, this randomness can be generated by a one-time bootstrapping protocol, such as the one described in [6]. In the later epochs, the reference shard (described below) is tasked with generating the randomness for the next epoch at the end of the ongoing one.

There are two types of shards in our protocol: the reference shard, and the worker shards.[1] The reference shard is a distinguished shard that is tasked with the organization of the shards between and during epochs. On the other hand, worker shards process transactions that are submitted by external users. A node can participate in both the reference shard and a worker shard but not more than one worker shard. Further, the reference shard is guaranteed to satisfy both the safety and the liveness properties of SMR, whereas some worker shards might satisfy only the safety property, and hence can be inactive.

After shards are formed, each shard first runs a distributed key generation (DKG) protocol. Hash digest of the generated public key is referred as shard's identity, and each node within the shard receives a share of the corresponding secret key. Once the DKG is completed, members of the shard sign and submit their shard's identity to the reference shard (Section IV-A). Upon receiving a sufficient number of signed messages from a shard, the reference shard assigns a set of transactions to that shard by modifying his tx-to-shard table (Section IV-B). After waiting for a sufficient time, this table is broadcast by the reference shard to the network. Note that, safe shards can be excluded from processing transactions for the ongoing epoch in case they do not complete this step within the allocated time (e.g., if corrupt nodes in such shards stay inactive).

Then, the shards start to process transactions and build their local ledgers by running an intra-shard SMR protocol (Section IV-C). We adapt the stateless blockchain model. This means, the nodes only maintain an accumulator computed over the ledger state. It is up to the clients to keep track of the explicit state of the ledgers by listening to the network. The explicit state is needed to generate membership proofs on transactions' inputs. A transaction can spread over multiple shards. In such a case, clients have to facilitate the verification between involved shards by running a cross-shard verification protocol (Section IV-E).

Finally, a safe shard can begin an epoch as active, but might become inactive at any time due. Inactive shards are eventually detected, and closed by the reference shard within an epoch (Section IV-D). When a shard is closed, one of the active shards take over its transactions, and continue building upon the closed shard's state.

[1]For brevity, we refer the worker shards just as shards most of the time, and specify the reference shard explicitly when needed.

A. Intra-Shard Key Generation

After nodes are assigned to shards, each shard internally runs a distributed key generation (DKG) protocol such as of [25]. The goal is to utilize a threshold signature scheme, e.g., threshold BLS [26], to minimize the communication complexity. Shards that successfully complete this step are authenticated by the reference shard, and only they are allowed to process transactions in the current epoch.

Concretely, after running a DKG protocol, a shard S generates a key-pair sk_S-pk_S such that, pk_S is a shard-wise shared public key (where we refer its hash digest $H(\mathsf{pk}_S)$ as *shard's identity*) and the sk_S is the corresponding secret key which is verifiably q-out-of-n shared among the nodes of S. After this step, each node r of S generates a signature share $\sigma_r(H(\mathsf{pk}_S))$ on his shard's identity using his share of the secret key. Then, every node sends their share of the signature to the reference shard. After accumulating q of these shares, reference shard constructs the threshold signature and verifies it with pk_S.

Given the verification passes, reference shard adds S to the *tx-to-shard table* by assigning S to its share of transactions (see Section IV-B to see how transactions are assigned). After waiting for a sufficient time to receive enough shares from all the shards, reference shard commits the tx-to-shard table to his state, and broadcasts it to the network. Upon receiving this table, shards can start to process transactions that they are assigned to.

Note that, it is possible for corrupt nodes in a safe shard S' to stall the DKG protocol, e.g., by staying silent. However, in this case, S' is excluded from the tx-to-shard table, and every node of it is assigned to $\perp$ in the node-to-shard assignment table. This indicates that, transactions are not routed to S', and (honest) nodes within S' do not participate in intra-shard SMR. This effectively closes S' for the current epoch.

B. Assigning Transactions to Shards

When assigning new transactions to shards, there are two factors to consider: first, we must ensure that every transaction is uniquely assigned to a shard to maintain consistency, and second, we should take load-balancing into account while doing so. In this section, we describe how we achieve these in our protocol.

As noted before, the reference shard is tasked with maintaining and updating a *tx-to-shard table*. Nodes receive this table from the reference shard at the beginning of each epoch and do look-ups on it to determine whether a transaction is assigned to their shard or not. This table is implemented as a complete binary search tree. At its leaves, we have the shard identifiers, and the path from the root to a leaf specifies the prefix of the hashes of transactions that the shard is assigned. For example, if the path from the root to a shard S is 00, S processes transactions whose first 2 bits are 00. As new shards are created or the existing ones are closed, the reference shard organizes the tree accordingly and broadcasts the updated table to the network. Figure 2 illustrates this process.

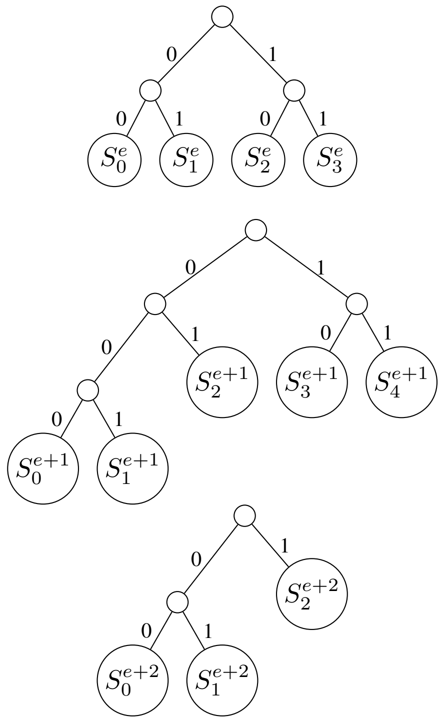


Fig. 2: Illustration of how tx-to-shard table is reorganized as nodes join/leave. In epoch e, we have 4 shards: S_0^e, S_1^e, S_2^e and S_3^e (top figure). Suppose some new nodes join the network between epoch e and $e+1$, and consequently, we split the network between 5 shards in epoch $e+1$. The tree is going to be reorganized accordingly (middle figure). Finally, suppose some nodes leave the network and the network is sharded between 3 shards in epoch $e+2$. The tree is reorganized again (bottom figure).

C. Intra-Shard SMR

We now explain the intra-shard SMR protocol. We first explain the structure of the blocks and then the details of the SMR protocol.

Block Structure. A block B_k at height k is a tuple consisting of a header B_k^h and a body B_k^b, given by,

$$B_k := (B_k^h, B_k^b).$$

The body of a block contains a set of transactions organized as a Merkle tree. The header is a tuple,

$$B_k^h := (k, e, \mathtt{MT}, A_S^{\mathtt{UTXO}}, H(B_{k-1}^h), H(\mathsf{pk}_S)),$$

where k is the height of the block, e is the epoch in which the block is created, $\mathtt{MT}$ is the root of the Merkle tree of transactions in the block's body, $A_S^{\mathtt{UTXO}}$ is an accumulator over the latest UTXO state of the shard S, $H(B_{k-1}^h)$ is the digest of the previous block's header, and $H(\mathsf{pk}_S)$ is the identity of the shard that creates the block. Due to the stateless blockchain model, after committing a block, nodes discard its body and only maintain the latest block header along with its *commit-certificate* $CC(B_k)$. As explained later in this section, a commit certificate on a header attests that the corresponding block is committed.

SMR. The goal of our intra-shard SMR is to satisfy safety for safe shards and satisfy both safety and liveness for super-honest shards. To achieve our goal, we start with the Sync HotStuff protocol [10], which satisfies both safety and liveness under an honest-majority assumption, and modify it accordingly. As is common in SMR protocols (and as done in Sync HotStuff), our protocol proceeds in two phases: a *steady-state* phase and a *view-change* phase. Steady-state proceeds in iterations, where in each iteration current leader of the shard proposes blocks to the rest of the shard. If a leader misbehaves during steady-state, by either proposing equivocating blocks or by not proposing any block at all, nodes execute the view-change protocol to replace the current leader, and then fall back to the steady-state. The explicit description of steady-state and view-change can be seen in Figures 3 and 4, respectively.

Quorum Certificates. Whenever a node r sends a message m, he signs its hash digest with his share of the secret key. We say q signature shares on m constitute a *quorum-certificate* for m. Technically, a quorum-certificate is the threshold signature generated out of the individual shares. Depending on the contents of m, we name these certificates differently: we call q **vote** messages on a block B_k as *block certificate*, denoted by $BC(B_k)$, and we call q **pre-commit** messages on a block B_k as *commit certificate*, denoted by $CC(B_k)$. Finally, we call q **blame** messages on a view v as *view-change certificate* and denote it by $VC(v)$.

Formally, we state the properties guaranteed by our intra-shard SMR as follows and defer their proofs to Section V.

Theorem 1 (Intra-shard Safety)**.** *Let S be a safe, or a super-honest shard. Given that, if an honest node in S commits a block B_k at height k, then every honest node in S eventually commits B_k at height k.*

Theorem 2 (Intra-shard Liveness)**.** *If a valid transaction T is assigned to a super-honest shard S, T is eventually committed by S.*

Theorem 3 (Provable Commits)**.** *Every committed block B_k of a shard S carries a commit-proof $\mathtt{COM}_{B_k,S}$ which attests that B_k is committed by S.*

D. Safe Shard Detector

As discussed before, safe shards lack liveliness. This means transactions that are assigned to safe shards are not guaranteed to be processed, e.g., in case corrupt nodes within them stay silent. To ensure such transactions are eventually processed, we use the honest nodes in a safe shard to detect the lack of liveness. We achieve this by augmenting each node with a *safe shard detector*, and by having them communicate with the reference shard.

During an epoch, each node monitors the state of his shard by running a safe shard detector whose specifications are given in Figure 5. Briefly, the safe shard detector keeps track of events that could only happen in a safe shard. Once a node detects

Let v be the current view number and let node L be the leader of the current view.

1) **Propose.** The leader L broadcasts $\langle \textbf{propose}, \langle B_k \rangle_L, BC(B_{k-1}), v\rangle_L$.
2) **Vote.** Upon receiving the first valid height-k block $\langle B_k \rangle_L$ from L or through a vote by some other node, set **precommit-timer**$_k$ to 2Δ and broadcast a vote in form $\langle \textbf{vote}, \langle B_k \rangle_L, v\rangle$.
3) **Precommit.** When **commit-timer**$_k$ reaches 0, if no equivocation for height-k is detected and if B_k has a block certificate $BC(B_k)$, broadcast $\langle \textbf{precommit}, \langle B_k \rangle_L, BC(B_k), v\rangle$.
4) **Commit.** Upon receiving q $\langle \textbf{precommit}, \langle B_k \rangle_L, BC(B_k), v\rangle$ messages such that they all contain a valid block certificate $BC(B_k)$ for B_k, if B_k is not already committed, commit B_k and all of its ancestors. Further, broadcast q received **precommit** messages which constitute a *commit-certificate* $CC(B_k)$ for B_k.

Fig. 3: The steady state phase of intra-shard SMR. The notation $\langle m \rangle_r$ indicates message m is signed by the secret key share of user r.

Let L and L' be the leader of view v and $v+1$, respectively.

1) **Blame.** If less than p blocks are received from L in $(2p+1)\Delta$ time in view v, broadcast $\langle \textbf{blame}, v\rangle$. If more than one block is proposed by L at any height, broadcast $\langle \textbf{blame}, v\rangle$ and the two equivocated blocks.
2) **Quit old view.** Upon gathering q $\langle \textbf{blame}, v\rangle$ messages, broadcast them and quit view v by aborting running **precommit-timer**s and stop voting in view v.
3) **Status.** Wait for 2Δ time and enter view $v+1$. Upon entering view $v+1$, send a highest block with a valid block certificate to L' and transition back to steady state.

Fig. 4: The view-change phase of intra-shard SMR.

such an event, he sends a **close** request on his shard to the reference shard.

Once the reference shard receives $n-q+1$ **close** requests from a shard S_i, it updates the node-to-shard and tx-to-shard tables: every node of S_i is assigned to $\perp$ in node-to-shard table, and transaction that were previously assigned to S_i is now assigned to one of the running shards S_j. The updated tables are broadcast to the network, and upon receiving them, honest nodes of S_i stop participating in intra-shard SMR for the current epoch. Further, nodes of S_j request and receive the latest state of S_i from the honest nodes in it. After receiving the latest state of S_i, S_j start to process transactions on behalf of S_i for the current epoch. Details of this process is further elaborated in Figure 6.

Formally, the safe-shard detector ensures the following properties. We defer their proofs to Section V-B.

Lemma 1. *Safe shard detector keeps track of events that could only happen in a safe shard. Consequently, an honest node in a super-honest shard never sends a* **close** *request, and thus, a super-honest shard is never closed.*

Lemma 2. *A safe shard either eventually processes transactions, or is eventually closed.*

E. Transaction Verification

We now describe how transactions are verified. We start by describing the transaction format, and then explain how single-shard and cross-shard transactions are verified.

Transaction Format. We adopt the well-known UTXO model where a transaction contains a set of *inputs* and a set of *outputs*. For simplicity, consider a simple transaction T with a single input I and a single output O, denoted as,

$$T := \langle I, O \rangle.$$

The output is an address-value pair, $O := (\mathsf{pk}, v)$, indicating that v amount of currency should be sent to address pk. The input is a tuple denoted by,

$$I := (O', \mathsf{sig}, \mathtt{MEM}_{(O',S)}),$$

where $O' := (\mathtt{ctr}, \mathsf{pk}', v')$ refers to an UTXO, contained in the accumulator of shard S with $\mathtt{ctr}$ as its addition-order in the accumulator, and sig is a signature valid under pk'. Finally, $\mathtt{MEM}_{(O',S)}$ is a membership-proof on O' with respect to $A_S^{\mathtt{UTXO}}$.

Single-Shard Verification. If T is assigned to the shard S where O' resides, then verification of T can be done locally, and consist of: (1) verifying T is assigned to S by doing a look-up on tx-to-shard table, (2) checking $v' \leq v$, and (3) verifying validness of sig under pk', and (4) verifying $\mathtt{MEM}_{\langle O',S\rangle}$ with respect to $A_S^{\mathtt{UTXO}}$. Assuming T passes the verification, S processes T by including it in a block, and consequently removing O' from $A_S^{\mathtt{UTXO}}$ and adding O to $A_S^{\mathtt{UTXO}}$.

Cross-shard Verification. Now let $T := (\langle I_1, I_2\rangle, O_3)$ and assume that UTXOs O_1 and O_2, referenced by I_1 and I_2, are contained in shards S_1 and S_2, respectively. Further, assume T is assigned to shard S_3. Verification of T requires the client to execute a *cross-shard verification* protocol.

To create a valid cross-shard transaction, client first requests a proof-of-inclusion (PoI) from the inputs shards. A PoI attests that the output referenced by an input is *removed* from the UTXO of the shard maintaining it. Concretely, client sends T to S_1 and S_2, and they: (1) first verify I_1 and I_2, (2) then

Let r be an arbitrary honest node in a shard S whose current leader is L in view v.

- **Vote inactivity.** After voting for a block B_k, set **inactivity-timer**$_{vt}$ to 2Δ. Send a $\langle$**close**$, S\rangle_r$ message to the reference shard if none of the following happened when **inactivity-timer**$_{vt}$ is 0.
  - An equivocation proof for L is received.
  - A view-change certificate $VC(v)$ is received.
  - A block-certificate $BC(B_k)$ is received.
- **Precommit inactivity.** After broadcasting a **precommit** message for a block B_k, set **inactivity-timer**$_{pc}$ to 2Δ. Send a $\langle$**close**$, S\rangle_r$ message to the reference shard if none of the following happened when **inactivity-timer**$_{pc}$ is 0.
  - An equivocation proof for L is received.
  - A commit-certificate $CC(B_k)$ is received.
- **Blame inactivity.** After broadcasting an equivocation proof, set **inactivity-timer**$_{bl}$ to 2Δ. If a view-change certificate $VC(v)$ is not received when **inactivity-timer**$_{bl}$ is 0, then send a $\langle$**close**$, S\rangle_r$ message to the reference shard.
- **View-change inactivity.** After quitting the view v, set **inactivity-timer**$_{vc}$ to 6Δ. Send a $\langle$**close**$, S\rangle_r$ message to the reference shard if none of the following happened when **inactivity-timer**$_{vc}$ is 0.
  - A valid block proposal is received.
  - A view-change certificate $VC(v+1)$ is received.
- **View-change bound.** After doing $n-q+1$ view-changes, send a $\langle$**close**$, S\rangle_r$ message to the reference shard.

Fig. 5: Specifications of the safe shard detector.

Let L_R be the current leader of the reference shard S_R. Upon receiving $n-q+1$ **close** messages from the nodes of a shard S_i, L_R initiates the following protocol to close S_i for the ongoing epoch and to enforce S_i transfers its state to one of the active shards S_j.

1. **Updating the tables.** L_R updates node-to-shard table by assigning every member of S_i to $\perp$, and updates transaction-to-shard table by reorganizing the underlying tree structure after removing S_i. Suppose S_j is assigned to transactions which were previously assigned to S_i due to this reorganization.
2. **Committing to the new tables.** L_R proposes updated tables along with close request to the reference shard. Nodes of the reference shard runs an SMR protocol to commit the proposal. Note that, since reference shard is an honest-majority shard, they do not run the protocol described in Section IV-C, but rather an SMR protocol that works under honest-majority such as the plain Sync HotStuff,
3. **Broadcasting the new tables.** Once proposal is committed, L_R broadcasts it to the rest of the network which carries the new tables.
4. **Closing the shard and state-transfer.** Upon receiving the proposal and verifying its correctness, nodes of S_i stop to participate in the intra-shard SMR, and nodes of S_j request the latest state from nodes of S_i. Nodes of S_i reply to this request by sending their latest block headers. After receiving the state of S_i, S_j starts to process transactions which were previously assigned to S_i by running an independent SMR instance along with his running SMR instance(s).

Fig. 6: Steps of closing a shard during an epoch. This process involves the reference shard updating its node-to-shard and tx-to-shard tables, and a state transfer from the closed shard to one of the active shards.

include $(T,1)$ and $(T,2)$ in a block, (3) and consequently remove O_1 and O_2 from $A^{\texttt{UTXO}}_{S_1}$, and $A^{\texttt{UTXO}}_{S_2}$, respectively. Given that, if S_i is the shard which removes the output referenced by I_i, we denote the respective PoI as,

$$\texttt{POI}_{\langle I_i,S_i\rangle} := (\texttt{MT}_{\langle (T,i),B\rangle}, B^h, \texttt{COM}_{\langle B,S_i\rangle}),$$

where $\texttt{MT}_{(T,i),B}$ is the Merkle-proof on (T,i) with respect to block B that contains (T,i), where i refers to which input of T is processed. Together with the header of block B^h, and the commitment proof on block $\texttt{COM}_{(B,S_i)}$, one can verify (T,i) is committed by S_i, and consequently, O_i is removed from the shard's state.

Depending on whether I_1 and I_2 are valid (e.g., they refer to valid UTXOs in their respective shards), there are three possible cases of how T can be processed. We describe each below.

- **Both I_1 and I_2 are valid.** S_1 and S_2 process T, so client obtains both $\texttt{POI}_{\langle I_1,S_1\rangle}$ and $\texttt{POI}_{\langle I_2,S_2\rangle}$. Client then creates the *commit* transaction,

$$T_{com} := (T, \texttt{POI}_{\langle I_1,S_1\rangle}, \texttt{POI}_{\langle I_2,S_2\rangle}),$$

and sends it to S_3. Upon receiving T_{com}, S_3 first verifies $O_1.v_1 + O_2.v_2 = O_3.v_3$, and then verifies the POIs. Verification of a $\texttt{POI}_{\langle I_i,S_i\rangle}$ consists of the following steps: (1) deducing S_i from B^h, (2) verifying $\texttt{COM}_{\langle B,S_i\rangle}$ with identity of S_i, (3) verifying the Merkle-proof on

(T, i) with respect to B^h. Given both I_1 and I_2 are valid, both POIs are valid too. Thus, S_3 includes T_{com} in the next block, and adds O_3 in $A_{S_3}^{\texttt{UTXO}}$.

- **Only one of I_1 and I_2 is valid.** Without loss of generality, assume I_1 is valid and I_2 is not valid. In this case, client only receives $\texttt{POI}_{\langle I_1, S_1\rangle}$. This violates atomicity as only O_1 is removed. However, we allow the client to refund the amount of O_1, to ensure consistency, by sending a partial commit transaction,

$$T_{com} := (T, \texttt{POI}_{\langle I_1, S_1\rangle}),$$

  to S_3. Upon receiving T_{com}, S_3 verifies the only PoI, includes T_{com} in the next block, and adds O_3' in $A_{S_3}^{\texttt{UTXO}}$ such that $O_3'.v_3' = O_1.v_1$.

- **Both I_1 and I_2 are invalid.** In this case, client does not receive any PoI, and hence, he cannot create any valid cross-shard transaction.

Finally, we discuss *replay attacks*. As discussed in the work of Sonnino *et al.* [8], cross-chain transaction protocols of OmniLedger and RapidChain are susceptible to such attacks. The cross-shard verification protocol we described so far is susceptible to replay attacks too. We explain those attacks and how we augment our protocol against them.

Replay Attacks. Again consider the transaction $T := \langle\langle I_1, I_2\rangle, O\rangle$ and suppose both I_1 and I_2 are valid so that the client eventually creates the valid commit transaction T_{com}. Upon receiving T_{com}, S_3 includes it in a block and creates O_3. One can observe that, an adversary could simply send T_{com} to S_3 multiple times. Given the verification passes each time, S_3 creates the output O_3 multiple times.

To prevent this, shards can simply keep track of PoIs they processed. Doing this naively forces nodes to maintain an ever-growing list. To alleviate this, we utilize an another accumulator. That is, a shard S maintains an accumulator $A_S^{\texttt{POI}}$ on the set of processed PoIs, and whenever a client executes a cross-shard transaction with outputs to be created in S, he must provide valid *non-membership* proofs on PoIs to this accumulator.

For example, assuming both I_1, I_2 are valid, the final commit transaction T_{com} with replay resistance is given by,

$$T_{com} := (T, \texttt{POI}_{\langle I_1,S_1\rangle}, \texttt{POI}_{\langle I_2,S_2\rangle}, \texttt{NMEM}_{\texttt{POI}_{\langle I_1,S_1\rangle}}, \texttt{NMEM}_{\texttt{POI}_{\langle I_2,S_2\rangle}}),$$

where $\texttt{NMEM}_{\texttt{POI}_{\langle I_i,S_i\rangle}}$ is a non-membership proof on $\texttt{POI}_{\langle I_i,S_i\rangle}$ with respect to accumulator $A_{S_i}^{\texttt{POI}}$. Further, by extending the block headers to maintain PoI accumulators, we do not need modify our state transfer protocol to transfer the state processed PoIs. That is, the final block header structure under replay resistance becomes,

$$B_k^h := (k, e, \texttt{MT}, A_S^{\texttt{UTXO}}, A_S^{\texttt{POI}} H(B_{k-1}^h), H(\mathsf{pk}_S)).$$

V. Security Analysis

A. Intra-Shard Safety and Liveness

We provide the proofs of safety and liveness for our intra-shard SMR here. Our proofs are very similar to the proofs of Sync HotStuff [10] with a few minor modifications, and we adopt their terminology of direct/indirect commits. That is, we say a block B_k is committed directly if an honest node commits B_k by obtaining a *commit-certificate* on it. Otherwise, we say a block B_k is committed indirectly if B_k is committed a result of directly committing another block extending on B_k. We start by proving Theorem 1 using Lemma 3 and 4.

Lemma 3. *Within a safe or a super-honest shard, if an honest node broadcasts a* ***precommit*** *message on a block B_k in a view, (i) every honest node votes for B_k in that view, and (ii) every honest node receives $BC(B_k)$ before entering the next view.*

Proof. Suppose an honest node r broadcasts a **precommit** message for B_k at time $t + 2\Delta$ in view v. This implies r has received and voted for B_k at time t. Further, every honest node must have received r's vote by $t + \Delta$. So, no later than $t + \Delta$, every honest node should have received and voted for B_k, unless (i) they have voted for an equivocating block B_k', or (ii) they have quitted view v, consequently they received and broadcast q **blame** messages on v before $t + \Delta$. However, if either of these had happened, r would not have broadcast **precommit** at $t + 2\Delta$ as he would have received either the equivocating blocks or q blame messages before $t+2\Delta$. Thus, it must be the case all honest nodes voted for B_k at $t+\Delta$. This further implies all honest nodes are still in view v until $t+3\Delta$, and by then, they would receive r's **precommit** message which contains $BC(B_k)$. ■

Lemma 4. *Within a safe or a super-honest shard, if an honest node broadcasts a* ***precommit*** *message on a block B_k, then there does not exist $BC(B_k')$ where $B_k' \neq B_k$, i.e.,, B_k is the only height-k block with a block-certificate.*

Proof. Suppose an honest node r broadcasts a **precommit** message on B_k in view v. We show a block-certificate on an equivocating block does not exist prior to, in, or after view v. First, if an equivocating block-certificate exists prior to view v, then at least one honest node must have voted for B_k'. This is because, in both safe and super-honest shards, number of corrupt nodes is strictly bounded from above by q. This vote would have reached r before view v, and would have prevented r from broadcasting **precommit**. Second, given by Lemma 3, all honest nodes vote for B_k in view v. Thus, an equivocating block-certificate for height-k cannot be formed in view v. Finally, again by Lemma 3, every honest node receives $BC(B_k)$ before entering view $v+1$. From $v+1$ and onwards, the highest certified block of every honest node is at least

B_k so, no honest node will vote for an equivocating height-k block. Thus, no $BC(B'_k)$ where $B'_k \neq B_k$ can come into existence in views greater than v. ■

Theorem 1 (Intra-shard Safety). *Let S be a safe, or a super-honest shard. Given that, if an honest node in S commits a block B_k at height k, then every honest node in S eventually commits B_k at height k.*

Proof. Suppose an honest node r commits to B_k at height-k in view v. Then, it must be the case that at least one honest node broadcast **precommit** on B_k. Due to Lemma 4, B_k is the unique block with a block-certificate at height-k. Hence, no other height-k block could have q **precommit** messages on it as a precommit must carry a block-certificate. Finally, when r commits to B_k, he broadcasts the commit-certificate $CC(B_k)$ which is eventually received by every honest node. Thus, by latest, they commit to B_k after receiving the commit-certificate from r. ■

Theorem 2 (Intra-shard Liveness). *If a valid transaction T is assigned to a super-honest shard S, T is eventually committed by S.*

Proof. In a super-honest shard, honest nodes can form a quorum among themselves. Thus, liveness of super-honest shards directly follows from the liveness of Sync HotStuff (Theorem 4 of [10]). ■

Theorem 3 (Provable Commits). *Every committed block B_k of a shard S carries a commit-proof $\texttt{COM}_{B_k,S}$ which attests that B_k is committed by S.*

Proof. From Theorem 1, we see that a block B_k is directly committed iff there exists a commit-certificate on B_k. Further, if a sequence of preceding blocks $B_{k-i} \leftarrow B_{k-i+1}, \ldots, \leftarrow B_{k-1}$ are committed indirectly as a result of committing B_k, then for each of these blocks, there exists a hash-chain starting from B_k and going back to the individual block. That is, a committed block either carries a commit-certificate, or there exists a hash-chain starting from a block with commit-certificate and referencing all the way back to the block itself. These constitute a commit proof for directly and indirectly committed blocks, respectively. Finally, since header of a block carries the identity of the shard that committed block, commit proof further identifies the shard which committed the block. ■

B. Safe Shard Detector

Lemma 1. *Safe shard detector keeps track of events that could only happen in a safe shard. Consequently, an honest node in a super-honest shard never sends a* **close** *request, and thus, a super-honest shard is never closed.*

Proof. We prove this lemma by analyzing each case specified by points in Figure 5, and showing that they cannot happen in a super-honest shard.

Let r be an honest node in some super-honest shard. Consider the **vote inactivity** case and suppose r votes for a block B_k at time t. Then, every other honest node receives and votes for B_k by $t + \Delta$ unless, (i) they have received and broadcast an equivocating block, or (ii) they have received and broadcast q blame messages on v which constitute $VC(v)$. Thus, if neither (i) nor (ii) happened and r is in a super-honest shard, it receives $BC(B_k)$ by $t + 2\Delta$. On the other hand, if either (i) or (ii) happens, r either receives equivocating blocks or $VC(v)$, respectively.

Now consider the **precommit inactivity** case. Suppose r broadcasts a **precommit** message for a block B_k at time t. Then, either all honest nodes broadcast a **precommit** for B_k by $t + \Delta$, or it could be that the current leader is corrupt and sends an equivocating block to some of the nodes to prevent them from broadcasting **precommit**. Note that equivocating blocks must be received by a node before $t + \Delta$ to prevent the node from broadcasting **precommit** for B_k. Given that, r receives either an equivocating block, or at least q **precommit** messages which constitute $CC(B_k)$ before $t + 2\Delta$ if it is in a super-honest shard.

For **blame inactivity** case, suppose r receives and broadcasts a pair of conflicting blocks at time t. Every honest node receives this equivocation proof by $t + \Delta$, and they broadcast a blame message once they do. This implies, in a super-honest shard, at least q blame messages must have been received by r until $t + 2\Delta$.

Now for **view-change inactivity** case, suppose r quits view v at time t. Since r broadcasts $VC(v)$ at t, every honest node receive $VC(v)$ and quit view v by $t+\Delta$. After quitting view v, every honest node waits for 2Δ and report their highest block with a valid block certificate to the leader of $v+1$. Given that, leader of $v+1$ is able to propose a valid block at time $t+4\Delta$. If leader did not propose any block at all, all honest nodes broadcast **blame** on $v+1$ at $t+5\Delta$. Given there are q or more honest nodes, i.e., the shard is super-honest, r obtains $VC(v+1)$ by $t + 6\Delta$. On the other hand, the leader could propose to a subset of honest nodes to prevent from being overthrown. Those nodes must receive the proposed block before $t + 5\Delta$. Since an honest node broadcasts his votes, r should receive the proposed block by $t + 6\Delta$ the latest.

Final point, **view-change bound**, merely observes that the number of corrupt nodes in a super-honest shard is at most $n-q$. In other words, there can be at most $n-q$ view-changes during an epoch in a super-honest shard before the shard finds an honest leader. Therefore, a node who does more than $n-q$ view changes must be in a safe-shard.

All in all, we showed that all the mentioned events could not happen in a super-honest shard, and an honest node from a super-honest shard never sends a **close** request. Thus, even if all corrupt nodes in a super-honest shard sends a malicious close request, such shard cannot be closed as the reference shard requires at least $n - q + 1$ close requests from a shard to close it, and a super-honest shard has at most $n - q$ corrupt nodes by definition. ■

Lemma 2. *A safe shard either eventually processes transactions, or is eventually closed.*

Proof. Due to Lemma 1, we see that a safe shard has to emulate a super-honest shard from preventing honest nodes within it to send **close** requests, and eventually closing down the shard. This means, a safe shard either has to process transactions by producing valid blocks, or keep doing view-changes. Since the number of view-changes that can be done is bounded, a safe shard either eventually processes transactions or is eventually closed. ■

C. Inter-Shard Safety And Liveness

We first prove inter-shard safety, as defined in Theorem 4, by making use of Lemma 5 and Lemma 6, and then prove inter-shard liveness as defined in Theorem 5.

Lemma 5. *Any two different proof-of-inclusions generated during the execution of the protocol are not for the two inputs referring to the same UTXO.*

Proof. According to the tx-to-shard table, the shard that can generate the proof-of-inclusion of an input I referring to some UTXO O is uniquely determined by the hash of the transaction which outputs O. Then with the intra-shard safety property in Theorem 1, the shard will not generate two different proof-of-inclusions for the same UTXO. ■

Lemma 6. *Let T^1_{com} and T^2_{com} be two different commit transactions that are confirmed during the execution of the protocol. Then, they cannot contain the same proof-of-inclusion.*

Proof. For the sake of contradiction, we assume that there is a proof-of-inclusion $\texttt{POI}_{\langle I,S \rangle}$ that appears in two different commit transactions. As for a commit transaction to be confirmed, its base transaction must match the transaction indicated by all proof-of-inclusions, these two commit transaction must contain the same base transaction. Denote this base transaction with T. Assume that the first commit transaction is confirmed by shard S_1 in epoch e_1 and second one is confirmed by shard S_2 in epoch e_2. Then there are two cases:

- $e_1 = e_2$. Then $S_1 = S_2$ as the prefix of $H(T)$ uniquely determines which shard processes the transaction in one epoch, where $H(T)$ denotes the hash value of the base transaction of these two commit transactions. Without loss of generality, assume that the first commit transaction is processed by S_1 before the second commit transaction. Then after the first commit transaction is confirmed by S_1, $\texttt{POI}_{\langle I,S \rangle}$ is added to $A^{\texttt{POI}}_{S_1}$, thus the verification of non-membership proof $\texttt{NMEM}_{\texttt{POI}_{\langle I,S \rangle}}$ will not pass when the shard S_1 processes the second commit transaction.
- $e_1 < e_2$. Denote the hash of the base transaction with $h = H(T)$ and the prefix of length l of h with $h[l]$. Let S_1 be assigned to $h[l_1]$ in epoch e_1 and S_2 be assigned to $h[l_2]$ in epoch e_2. Then there are two cases: $l_1 \leq l_2$ or $l_1 > l_2$. As described before, in either case the shard S_2 will check the accumulator for proof-of-inclusion once handled by S_1 in e_1, and the non-membership proof $\texttt{NMEM}_{\texttt{POI}_{\langle I,S \rangle}}$ cannot pass the verification.

■

Theorem 4 (Inter-shard Safety)**.** *Once a shard commits a transaction T, no shard can commit to a transaction T' such that T and T' spend the same output, i.e., double spending cannot happen.*

Proof. Lemma 6 shows that any proof-of-inclusion can only be used once. Together with Lemma 5, it trivially guarantees that any output can be spent only once. ■

Theorem 5 (Inter-shard Liveness)**.** *When a valid transaction T is assigned to a shard S, either S commits T, or eventually S is closed and T is committed by another shard S', i.e., valid transactions are eventually processed by a shard.*

Proof. This simply follows from (i) the liveness of super-honest shards (Theorem 2), and (ii) second property of the safe-shard detector (Lemma 2). Briefly, if S_i is a super-honest shard, it eventually commits T due to Theorem 2. If S_i is a safe shard on the other hand, Lemma 2 says that, either S_i eventually commits to T, or S_i is eventually closed and transactions that are assigned to it are reassigned to a running shard S_j. This process could cascade for a few shards (e.g.,, in case S_j is a safe shard too), however, since there is at least one super-honest shard with high probability, T eventually reaches to a super-honest shard, and committed by it. ■

VI. Performance Analysis

We now do a performance analysis of our protocol and compare it with RapidChain [27], the current state-of-the-art sharding protocol in synchronous setting to the best of our knowledge. Concretely, we compare two quantities for these protocols: the failure probability of an epoch, and the expected number of active shards per epoch.

Notation. We denote the network size by N, and the network resiliency by F. Then, $T = \lfloor N \cdot F \rfloor$ gives us the maximum number of corrupt nodes that we can tolerate. We denote the shard size by n which we assume it divides N exactly for simplicity. Let S be an arbitrary shard. Then, we denote the number of honest nodes in S via the random variable H_S, and the number of corrupt nodes in it via the random variable C_S where $n = H_S + C_S$. Finally, we denote the number of active shards (that satisfy both liveness and safety) in the protocol by m.

RapidChain. In RapidChain, each shard has to consist of an honest-majority, and be active, or else the protocol fails. Therefore, the number of active shards (until failure) is simply given by,

$$\boldsymbol{m_{Rapid}} = N/n.$$

Further, the failure probability of an arbitrary shard S can be computed as the probability of not having an honest majority in S. We can upper-bound this probability via hypergeometric distribution as follows,

$$\Pr\left[C_S \geq \lfloor n/2 \rfloor\right] = \sum_{i=\lfloor n/2 \rfloor}^{n} \frac{\binom{T}{i}\binom{N-T}{n-i}}{\binom{N}{n}}.$$

And by union-bound, we can upper-bound the failure probability for the protocol itself,

$$\boldsymbol{Pr_{fail}^{Rapid}} = \Pr\left[C_S \geq \lfloor n/2 \rfloor\right] \cdot N/n.$$

Instachain. In Instachain, we have another parameter that can be tuned in addition to n, the quorum size parameter $q < n$. Remember that, for a given value of q, a (worker) shard S can be in one of the three different states,

- S is active if $H_S \geq q$.
- S is corrupt if $C_S \geq q$.
- S is safe if it is neither active nor corrupt, i.e., $H_S < q$ and $C_S < q$.

Given that, the probability of a shard being active can be computed as,

$$\Pr\left[H_S \geq q\right] = \sum_{i=q}^{n} \frac{\binom{T}{n-i}\binom{N-T}{i}}{\binom{N}{n}}.$$

By linearity of expectation, the expected number of active shard is then given by,

$$\boldsymbol{m_{Insta}} = \Pr\left[S \textit{ is active}\right] \cdot N/n.$$

Similarly, we can compute the probability of having a corrupt shard as,

$$\Pr\left[C_S \geq q\right] = \sum_{i=q}^{n} \frac{\binom{T}{i}\binom{N-T}{n-i}}{\binom{N}{n}}.$$

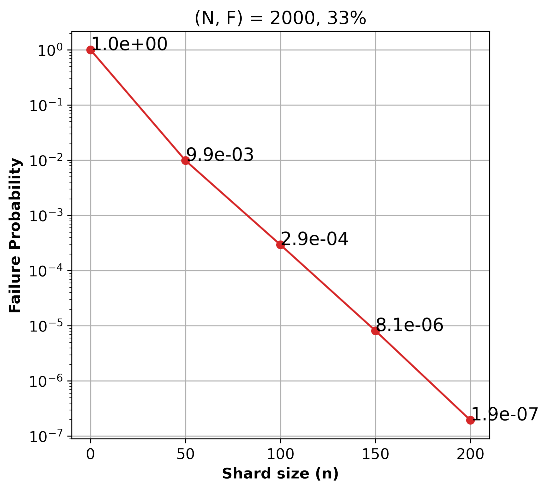
To compute the failure probability of the protocol, we have consider the reference shard as well. Remember, the reference shard S_{ref} is a distinct shard that must be active, and thus, must consist of an honest majority *with high probability*, like RapidChain's shards. So, if we denote the size of S_{ref} by $n_{ref} > n$, we can compute its failure probability as exactly done in RapidChain,

$$\Pr\left[S_{ref} \textit{ is corrupt}\right] = \sum_{i=\lfloor n_{ref}/2 \rfloor}^{n_{ref}} \frac{\binom{T}{i}\binom{N-T}{n_{ref}-i}}{\binom{N}{n_{ref}}}.$$

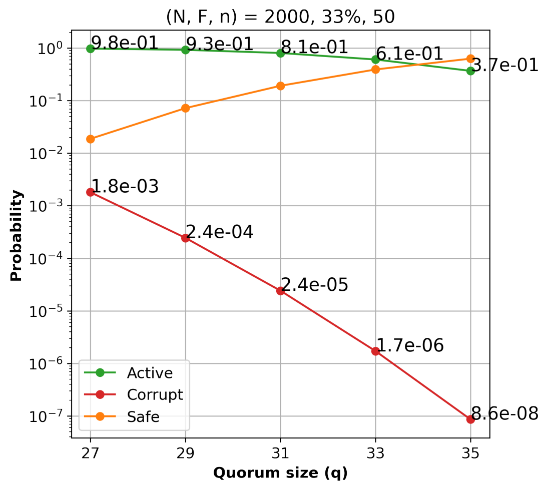
Finally, the protocol fails when either the reference shard fails, or any of the worker shards fail. Therefore, we can upper-bound the failure probability of Instachain as,

$$\boldsymbol{Pr_{fail}^{Insta}} = \Pr\left[S_{ref} \textit{ is corrupt}\right] + \Pr\left[C_S \geq q\right] \cdot N/n.$$

Comparison. We first compare two protocols at the shard level in Figure 7. As can be seen in the plot, we can obtain similar shard failure probabilities in Instachain under smaller shard sizes. This is achieved by increasing the quorum size instead of increasing the shard size.



(a) RapidChain



(b) Instachain

Fig. 7: Comparison of RapidChain and Instachain at the shard level with parameters $N = 2000$, $F = 1/3$. For RapidChain, we plot the failure probability of a shard as its size increases. For Instachain however, we do not necessarily has to increase the shard size to obtain smaller failure probabilities. To demonstrate this, we arbitrarily fix the shard size to $n = 50$, and increase the quorum size instead. For example, the failure probability of a shard is around $1.9e - 7$ when $n = 200$ for RapidChain. By setting $q = 35$, we obtain a lower failure probability of $8.6e - 8$ for Instachain. Further, the probability of the shard being active is around 0.37 at the same setting. This means, the expected number of active shards is around 14.8 until failure, where as this figure is only 10 for RapidChain.

Finally, we compare both works at a protocol level in Table I. For these figures, we assumed the reference shard size is of 4 times of a worker shard in Instachain. Briefly, these figures concretely demonstrate that Instachain scales beyond RapidChain, by having more active shards, under a similar/smaller failure probability.

VII. Conclusion

In this paper, we introduced a sharded blockchain protocol, named Instachain. Through our analysis in Section VI, we showed that it can scale beyond the current state-of-the-art by creating smaller, and more active shards. The main novelty that allowed this improvement was relaxing the liveness property for some shards in the network. This was done by adjusting the quorum size parameter of the intra-shard SMR appropriately. We have further devised efficient ways to deal with safe shards, and proved the security of our protocol in this setting.

Yet, we are aware the current state of this works lack concrete figures for throughput, storage, latency, etc. This prevents us from comparing our protocol with existing ones at a finer granularity. We hope to adress this issue in a further iteration of our work by computing these quantities under a prototype implementation.

Protocol	**Network Size (N)**	**Network Resiliency (F)**	**Shard Size (n)**	**Shard Resiliency**	**Expected TTF**	**Expected # of Active Shards**
RapidChain [6]	$N = 1,000$	$F = 33\%$	$n = 200$	$f = 50\%$	$17,128$ years	5
RapidChain [6]	$N = 2,000$	$F = 33\%$	$n = 200$	$f = 50\%$	$1,412$ years	10
RapidChain [6]	$N = 4,000$	$F = 33\%$	$n = 200$	$f = 50\%$	312 years	20
RapidChain [6]	$N = 8,000$	$F = 33\%$	$n = 200$	$f = 50\%$	114 years	40
Instachain	$N = 1,000$	$F = 33\%$	$n = 100$	$f = 60\%$	$39,638$ years	9.4
Instachain	$N = 2,000$	$F = 33\%$	$n = 100$	$f = 60\%$	$7,666$ years	18.8
Instachain	$N = 4,000$	$F = 33\%$	$n = 100$	$f = 60\%$	$2,407$ years	37.5
Instachain	$N = 8,000$	$F = 33\%$	$n = 100$	$f = 60\%$	972 years	74.8

TABLE I: Comparison of Instachain with RapidChain at the protocol level. Time-to-failure (TTF) is computed by assuming an epoch duration of 24 hours. For RapidChain, we arbitrarily set the shard size to $n = 200$, and computed the TTF under different values of N. As for Instachain, we arbitrarily set the shard size to half of RapidChain's. We then searched over different quorum sizes, and tried to find a value that gives us a greater TTF, and number of active shards than RapidChain. This quorum size value turned out to be 60 (hence, its shard resiliency is 60%). All in all, we demonstrate that Instachain can scale much beyond RapidChain by constructing smaller shards, while maintaining a smaller failure probability. Finally, it might have been possible to scale Instachain beyond the values we present above, by experimenting with different combinations shard and quorum sizes, but we find the figures above good enough to demonstrate the better scalability of our approach.

References

[1] M. Zamani and M. Ozdayi, “Blockchain sharding with adjustable quorums,” March 2021. https://patents.google.com/patent/WO2021050929A1.

[2] J. Poon and T. Dryja, “The bitcoin lightning network: Scalable off-chain instant payments.” https://lightning.network/lightning-network-paper.pdf, 2016.

[3] T. Groenfeldt, “Real-time payments will get real in the u.s. in 2019.” https://www.forbes.com/sites/tomgroenfeldt/2019/03/05/real-time-payments-will-get-real-in-the-u-s-in-2019/#65552e787dfe, March 2019. (Accessed on 11/08/2019).

[4] L. Luu, V. Narayanan, C. Zheng, K. Baweja, S. Gilbert, and P. Saxena, “A secure sharding protocol for open blockchains,” in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS ’16, pp. 17–30, ACM, 2016.

[5] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford, “OmniLedger: A secure, scale-out, decentralized ledger via sharding,” in *2018 IEEE Symposium on Security and Privacy (S&P)*, pp. 19–34, 2018.

[6] M. Zamani, M. Movahedi, and M. Raykova, “RapidChain: Scaling blockchain via full sharding,” in *2018 ACM Conference on Computer and Communications Security (CCS)*, 2018.

[7] M. Pease, R. Shostak, and L. Lamport, “Reaching agreement in the presence of faults,” *Journal of the ACM (JACM)*, vol. 27, no. 2, pp. 228–234, 1980.

[8] A. Sonnino, S. Bano, M. Al-Bassam, and G. Danezis, “Replay attacks and defenses against cross-shard consensus in sharded distributed ledgers,” *CoRR*, vol. abs/1901.11218, 2019.

[9] S. Nakamoto, “Bitcoin: A peer-to-peer electronic cash system,” 10 2008. Available at https://bitcoin.org/bitcoin.pdf.

[10] I. Abraham, D. Malkhi, K. Nayak, L. Ren, and M. Yin, “Sync hotstuff: Simple and practical synchronous state machine replication.” Cryptology ePrint Archive, Report 2019/270, 2019. https://eprint.iacr.org/2019/270.

[11] S. Micali, S. Vadhan, and M. Rabin, “Verifiable random functions,” in *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, FOCS ’99, pp. 120–, IEEE Computer Society, 1999.

[12] S. Micali, “ALGORAND: the efficient and democratic ledger,” *CoRR*, vol. abs/1607.01341, 2016.

[13] B. Awerbuch and C. Scheideler, “Towards a scalable and robust DHT,” in *Proceedings of the Eighteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA ’06, pp. 318–327, ACM, 2006.

[14] B. David, B. Magri, C. Matt, J. B. Nielsen, and D. Tschudi, “Gearbox: An efficient uc sharded ledger leveraging the safety-liveness dichotomy,” *Cryptology ePrint Archive*, 2021.

[15] V. Buterin, “The stateless client concept - sharding - ethereum research.” https://ethresear.ch/t/the-stateless-client-concept/172, October 2017. (Accessed on 01/30/2020).

[16] J. Drake, “Accumulators, scalability of utxo blockchains, and data availability - sharding - ethereum research.” https://ethresear.ch/t/accumulators-scalability-of-utxo-blockchains-and-data-availability/176, October 2017. (Accessed on 01/30/2020).

[17] D. Boneh, B. Bünz, and B. Fisch, “Batching techniques for accumulators with applications to iops and stateless blockchains.,” *IACR Cryptology ePrint Archive*, vol. 2018, p. 1188, 2018.

[18] M. Andrychowicz and S. Dziembowski, *PoW-Based Distributed Cryptography with No Trusted Setup*, pp. 379–399. Springer Berlin Heidelberg, 2015.

[19] C. Decker, J. Seidel, and R. Wattenhofer, “Bitcoin meets strong consistency,” in *Proceedings of the 17th International Conference on Distributed Computing and Networking*, ICDCN ’16, pp. 13:1–13:10, ACM, 2016.

[20] R. Pass and E. Shi, “Hybrid consensus: Efficient consensus in the permissionless model.” Cryptology ePrint Archive, Report 2016/917, 2016. http://eprint.iacr.org/2016/917.

[21] R. Karp, C. Schindelhauer, S. Shenker, and B. Vocking, “Randomized rumor spreading,” in *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, FOCS ’00, pp. 565–, IEEE Computer Society, 2000.

[22] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich, “Algorand: Scaling byzantine agreements for cryptocurrencies,” in *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP ’17, pp. 51–68, ACM, 2017.

[23] I. Abraham, D. Malkhi, K. Nayak, L. Ren, and A. Spiegelman, “Solida: A blockchain protocol based on reconfigurable byzantine consensus,” in *Proceedings of the 21st International Conference on Principles of Distributed Systems*, OPODIS ’17, 2017.

[24] E. K. Kogias, P. Jovanovic, N. Gailly, I. Khoffi, L. Gasser, and B. Ford, “Enhancing bitcoin security and performance with strong consistency via collective signing,” in *25th {USENIX} Security Symposium ({USENIX} Security 16)*, pp. 279–296, 2016.

[25] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin, “Secure distributed key generation for discrete-log based cryptosystems,” in *Proceedings of the 17th International Conference on Theory and Application of Cryptographic Techniques*, EUROCRYPT’99, (Berlin, Heidelberg), pp. 295–310, Springer-Verlag, 1999.

[26] T. Hanke, M. Movahedi, and D. Williams, “Dfinity technology overview series, consensus system,” *arXiv preprint arXiv:1805.04548*, 2018.

[27] M. Zamani, M. Movahedi, and M. Raykova, “Rapidchain: Scaling blockchain via full sharding (full paper),” 2018. https://eprint.iacr.org/2018/460.

Disclaimer