RingBFT: Resilient Consensus over Sharded Ring Topology

Sajjad Rahnama, Suyash Gupta, Rohan Sogani, Dhruv Krishnan, Mohammad Sadoghi
Exploratory Systems Lab
University of California, Davis

ABSTRACT

The recent surge in federated data management applications has brought forth concerns about the security of underlying data and the consistency of replicas in the presence of malicious attacks. A prominent solution in this direction is to employ a permissioned blockchain framework that is modeled around traditional Byzantine Fault-Tolerant (Bft) consensus protocols. Any federated application expects its data to be globally scattered to achieve faster access. But, prior works have shown that traditional Bft protocols are slow.

This has led to the rise of sharded-replicated blockchains. Existing Bft protocols for these sharded blockchains are efficient if client transactions require access to a single-shard, but face performance degradation if there is a cross-shard transaction that requires access to multiple shards. As cross-shard transactions are common, to resolve this dilemma, we present RingBFT, a novel meta-Bft protocol for sharded blockchains. RingBFT requires shards to adhere to the ring order, and follow the principle of process, forward, and re-transmit while ensuring the communication between shards is linear. Our evaluation of RingBFT against state-of-the-art sharding Bft protocols illustrates that RingBFT achieves up to 18× higher throughput, gracefully scales to nearly 500 globally distributed nodes, and achieves a peak throughput of 1.2 million transactions per second.

1 INTRODUCTION

A growing interest in *federated data management* illustrates an increased demand for multi-party database management [9, 16, 61]. In these multi-party systems, a common database is maintained by several parties. As all of these parties cannot be at the same location, so the system needs to be decentralized, which implies that the database is distributed. There are two key ways in which a distributed database can be managed by multiple parties: *replication* and *sharding* [36, 38, 39, 57, 60, 66].

In a replicated system, each party holds a *copy* of the database. As a result, the effects of each client transaction are replicated across all the parties (replicas). In a sharded system, each party maintains a subset (shard) of the database. Hence, each party can independently handle incoming client transactions that require access to its shard.

One of the factors that advocates the use of replicated databases is their ability to handle *failure* of one or more replicas. This necessitates the need for keeping all the replicas at the same state. To achieve this task, databases employ *crash-fault tolerant* protocols such as Paxos [48] and Raft [55] to help all replicas reach a common order for each client transaction. However, one or more replicas can get *compromised* due to a malicious attack. A compromised replica may wish: (i) to exclude transactions of some clients, (ii) to make the system unavailable to clients, and (iii) to make replicas inconsistent. These malicious attacks are so common that one estimate shows that cyberattacks alone cost the U.S. economy around $57 billion dollars in 2016 [54]. As a result, not all the replicas can be trusted.

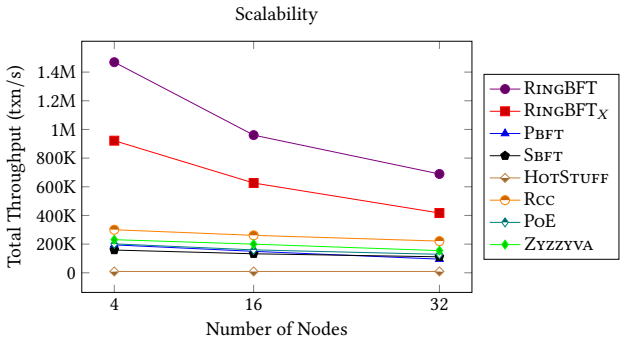

Figure 1: Comparing scalability of different Bft protocols. In this figure, we depict throughput of single-primary, multiple-primaries, geographically-scalable, and sharding Bft protocols. For RingBFT, we require each shard to have number of replicas stated on x-axis.

A recent solution to guarantee secure federated data management is through the use of *permissioned* blockchain technology [31, 37]. These permissioned blockchains require their replicas to agree on the order for each transaction by participating in a *Byzantine-Fault Tolerant* (Bft) consensus protocol. Post consensus, each replica logs the ordered transaction in a *block* that is part of an immutable append-only ledger–*blockchain*. A blockchain is termed as immutable because each new block includes the hash of the previous block, and it allows verifying the state of the participating replicas.

In this paper, we present a novel meta-Bft protocol RingBFT that guards against Byzantine attacks, achieves high throughput, and incurs low latency. Our RingBFT protocol explores the landscape of sharded-replicated databases, and helps to scale *permissioned* blockchains, which in turn helps in designing efficient federated data management systems. RingBFT aims to make consensus inexpensive even when transactions require access to *multiple shards*. In the rest of this section, we motivate our design choices. To highlight the need for RingBFT, we will be referring to Figure 1, which illustrates the throughput attained by the system when employing different Bft consensus protocols.

1.1 Challenges for Efficient BFT Consensus

Existing permissioned blockchain applications employ traditional Bft protocols to achieve consensus among their replicas [3, 4, 10, 46]. Over the past two decades, these Bft protocols have undergone a series of evolutions to guarantee resilience against Byzantine attacks, while ensuring high throughput and low latency. The seminal work by Castro and Liskov [10, 11] led to the design of the first practical Bft protocol, Pbft, which advocates a *primary-backup* paradigm where primary initiates the consensus and all the backups follow primary's lead. Pbft achieves consensus among the replicas in three phases, of which two

require quadratic communication complexity. Following this, several exciting primary-backup protocols, such as Zyzzyva [47], Sbft [23], and PoE [29], have been proposed that try to yield higher throughputs from Bft consensuses. We use Figure 1 to illustrate the benefits of these optimizations over Pbft. Prior works [2, 33] have illustrated that these *single* primary protocols are essentially *centralized* and prevent scaling the system to a large number of replicas.

An emerging solution to balance load among replicas is to employ *multi-primary* protocols like Honeybadger [53] and Rcc [30, 32] that permit all replicas to act as primaries by running multiple consensuses concurrently. However, multi-primary protocols also face scalability limitations as despite concurrent consensuses, each transaction requires communication between all the replicas. Moreover, if the replicas are separated by geographically large distances, then these protocols incur low throughput and high latency due to low bandwidth and high round-trip time. This led to the design of *topology-aware* protocols, such as Steward [2] and Geobft [33], which cluster replicas based on their geographical distances. For instance, Geobft expects each cluster to first locally order its client transaction by running the Pbft protocol, and then exchange this ordered transaction with all the other clusters. Although Geobft is highly scalable, it necessitates total replication, which forces communicating large messages among geographically distant replicas.

1.2 The Landscape for Sharding

To mitigate the costs associated with replicated databases, a common strategy is to employ the *sharded-replicated* paradigm [56]. In a sharded-replicated database, the data is distributed across a set of *shards* where each shard manages a unique *partition* of the data. Further, each shard replicates its partition of data to ensure availability under failures. If each transaction accesses only one shard, then these sharded systems can fetch high throughput as consensus is restricted to a subset of replicas.

AHL [15] was the first permissioned blockchain system to employ principles of sharding. AHL's novel design helps to scale blockchain systems to hundreds of replicas across the globe and achieve high throughput for *single-shard* transactions. To tackle *cross-shard* transactions that require access to data in multiple shards, AHL designates a set of replicas as a reference committee, which *globally orders* all such transactions. Following AHL's design, Sharper [4] presents a sharding protocol that eliminates the barrier to rely on the reference committee for ordering cross-shard transactions, but necessitates *global and quadratic communication among all replicas of all the participating shards.*

Why RingBFT? Decades of research in database community has illustrated that *cross-shard* transactions are common [14, 17, 39, 66]. In fact, heavy presence of these cross-shard transactions has led to development of several concurrency control [8, 39, 57] and commit protocols [24, 36, 62]. Hence, in this paper, we present our RingBFT protocol that significantly reduces the costs associated with cross-shard transactions.

Akin to AHL and Sharper, RingBFT assumes that the read-write sets of each transaction are known prior to the start of consensus. Given this, RingBFT guarantees consensus for each cross-shard transaction in ***at most two rotations around the ring***. In specific, RingBFT envisions each shard participating in *multiple circular flows* or rings, simultaneously. For each cross-shard transaction, RingBFT follows the principle of ***process, forward, and re-transmit***. This implies that each shard performs consensus on the transaction and forwards it to the next shard. This flow continues until each shard is aware of the fate of the transaction. However, the real challenge with cross-shard transactions is to manage conflicts and to prevent deadlocks, which RingBFT achieves by requiring cross-shard transactions to travel in ***ring order***. Despite all of this, RingBFT ensures communication between the shards is ***linear***, exhibiting a neighbor-to-neighbor communication. This minimalistic design has allowed RingBFT to achieve unprecedented gains in throughput and has allowed us to scale Bft protocols to nearly 500 nodes globally. The benefits of our RingBFT protocol are visible from Figure 1 where we run RingBFT in a system of 9 shards with each shard having 4, 16 and 32 replicas. Further, we show the throughput with 0 (RingBFT) and 15% (RingBFT$_X$) cross-shard transactions. We now list down our contributions.

(1) We present a novel meta-Bft protocol for sharded-replicated permissioned blockchain systems that requires participating shards to adhere to the ring order. We term RingBFT as "meta" because it can employ any single-primary protocols within each shard.

(2) Our RingBFT protocol presents a scalable consensus for cross-shard transactions that neither depends on any centralized committee nor requires all-to-all communication.

(3) We show that the cross-shard consensus provided by RingBFT is safe, and live, despite any Byzantine attacks.

(4) We evaluate RingBFT on our ResilientDB[1] framework [26–29, 32, 33, 35, 58] against two state-of-the-art Bft protocols for permissioned sharded systems, AHL [15], and Sharper [4]. Our results show that RingBFT easily scales to 428 globally-distributed nodes, and achieves up to 18× and 4× times higher throughput than AHL and Sharper, respectively.

2 CROSS-SHARD DILEMMA

For any sharded system, ordering a single-shard transaction is trivial as such a transaction requires access to only one shard. As a result, achieving consensus on a single-shard transaction requires running a standard Bft protocol. Further, single-shard transactions support *parallelism* as each shard can order its transaction in parallel, this without any communication between shards.

On the other hand, cross-shard transactions are *complex*. Not only do they require communication between shards but also their fate depends on the consent of each of the *involved shards*. Further, two or more cross-shard transactions can *conflict* if they require access to the same data. Such conflicts can cause one or more transactions to abort or worse, can create a *distributed deadlock* in a Byzantine setting. Hence, we need an efficient protocol to order these cross-shard transactions, which ensures that the system is both *safe* and *live*.

Designated Committee (*AHL*). One way to order cross-shard transactions is to designate a set of replicas with this task. AHL defines a reference committee that assigns an order to each cross-shard transaction, which requires running Pbft protocol among all the members of the reference committee [15]. Next, reference committee members run the Two-phase commit (2pc) protocol in a geo setting with all the replicas of involved shards. Notice that the 2pc protocol requires: (1) each shard to send a vote to the reference committee, (2) reference committee collects these votes and takes a decision (abort or commit), and (3) each shard implements the decision. Firstly, this solution requires each shard

[1] ResilientDB is open-sourced at https://resilientdb.com/ and its source-code is available at https://github.com/resilientdb/resilientdb.

to run the PBFT protocol to decide on the vote. Secondly, the reference committee needs to run PBFT again to reach a common decision. Finally, these multiple phases of 2PC require *all-to-all communication* between the replicas of each shard and the replicas of the reference committee; hence, we can no longer utilize the low latency and high bandwidth available within the same data center or region, and the effect of quadratic communication is further amplified due to the limited bandwidth across geographical regions.

Initiator Shard (SHARPER). Another way to process a cross-shard transaction is to designate one of the involved shards as the *initiator shard*. SHARPER [4] employs this approach by requiring each cross-shard transaction to be managed by the primary replica of one of the involved shards. This *initiator primary* proposes the transaction to the primaries of other shards. Next, these primaries propose this transaction within their own shards. Following this, there is an all-to-all quadratic communication among all replicas of all the involved shards in the geo setting.

3 SYSTEM MODEL

To explain our RINGBFT protocol in detail, we first lay down some notations and assumptions. Our system comprises of a set $\mathfrak{S}$ of shards where each shard S provides a replicated service. In specific, each shard S manages a *unique partition of the data*, which is replicated by a set $\mathfrak{R}_S$ of *replicas*.

In each shard S, there are $\mathcal{F} \subseteq \mathfrak{R}_S$ *Byzantine* replicas, of which $\mathcal{NF} = \mathfrak{R}_S \setminus \mathcal{F}$ are *non-faulty* replicas. We expect non-faulty replicas to follow the protocol and act deterministic, that is, on identical inputs, all non-faulty replicas must produce identical outputs. We write $\mathbf{z} = |\mathfrak{S}|$ to denote the total number of shards and $\mathbf{n} = |\mathfrak{R}_S|$, $\mathbf{f} = |\mathcal{F}|$, and $\mathbf{nf} = |\mathcal{NF}|$ to denote the number of replicas, faulty replicas, and non-faulty replicas, respectively, in each shard.

Fault-Tolerance Requirement. Traditional, BFT protocols such as PBFT, ZYZZYVA, and SBFT expect a total replicated system where the total number of Byzantine replicas are less than *one-third* of the total replicas in the system. In our sharded-replicated model, we adopt a slightly weaker setting where at each shard the total number of Byzantine replicas are less than *one-third* of the total replicas in that shard. In specific, at each shard S, we expect $\mathbf{n} \geq 3\mathbf{f}+1$. This does not imply that we want each shard to have an equal number of replicas. Each shard can have a different number of replicas till less than one-third are byzantine. This requirement is in accordance with existing works in Byzantine sharding space [4, 15].

Cross-Shard Transactions. Each shard $S \in \mathfrak{S}$ can receive a *single-shard* or cross-shard transaction. A single-shard transaction for S leads to *intra-shard* communication, that is, all the messages necessary to order this transaction are exchanged among the replicas of S. On the other hand, a cross-shard transaction requires access to data from a subset of shards (henceforth we use the abbreviation CST to refer to a cross-shard transaction). We denote this subset of shards as $\mathfrak{I}$ where $\mathfrak{I} \subseteq \mathfrak{S}$, and refer to it as *involved shards*. Each CST can be termed as *simple* or *complex*. A simple CST is a collection of fragments where each shard can independently run consensus and execute its fragment. On the other hand, a complex CST includes dependencies, that is, an involved shard may require access to data from other involved shards to execute its fragment.

Deterministic Transactions. We define a deterministic transaction as the transaction for which the data-items it will read/write are known prior to the start of the consensus [57, 66]. Given a deterministic transaction, a replica can determine which data-items accessed by this transaction are present in its shard.

Ring Order. We assume shards in set $\mathfrak{S}$ are *logically* arranged in a *ring topology*. In specific, each shard $S \in \mathfrak{S}$ has a position in the ring, which we denote by $\mathrm{id}(S)$, $1 \leq \mathrm{id}(S) \leq |\mathfrak{S}|$. RINGBFT employs these identifiers to specify the flow of a CST or ***ring order***. For instance, a simple ring policy can be that each CST is processed by the involved shards in the increasing order of their identifiers. RINGBFT can also adopt other complex permutations of these identifiers for determining the flow across the ring.

Authenticated Communication. We assume that each message exchanged among clients and replicas is *authenticated*. Further, we assume that Byzantine replicas are unable to impersonate non-faulty replicas. Notice that authenticated communication is a minimal requirement to deal with Byzantine behavior. For intra-shard communication, we employ cheap *message authentication codes* (MACs), while for cross-shard communication we employ digital signatures (DS) to achieve authenticated communication. MACs facilitate symmetric cryptography by requiring each pair of communicating nodes to share a *secret key*. We expect non-faulty replicas to keep their *secret keys* hidden. DS follow asymmetric cryptography. In specific, prior to signing a message, each replica generates a pair of *public-key* and *private-key*. The signer keeps the private-key hidden and uses it to sign a message. Each receiver authenticates the message using the corresponding public-key. Although MACs are cheaper than DS, they cannot guarantee *non-repudiation*. We require non-repudiation property during cross-shard communication as it helps to prove that the message communicated was sent by the sender and the message's contents were not fabricated.

In the rest of this manuscript, if a message m is signed by a replica R using DS, we represent it as $\langle m \rangle_R$ to explicitly identify replica R. Otherwise, we assume that the message employs MAC.

To ensure message integrity, we employ a *collision-resistant cryptographic hash function* $H(\cdot)$ that maps an arbitrary value v to a constant-sized digest $H(v)$ [44]. We assume that there is a negligible probability to find another value v', $v \neq v'$, such that $H(v) = H(v')$. Further, we refer to a message as *well-formed* if a non-faulty receiver can validate the DS or MAC, verify the integrity of the message digest, and determine that the sender of the message is also the creator.

4 RINGBFT CONSENSUS PROTOCOL

To achieve efficient consensus in sharded-replicated databases, we employ our RINGBFT protocol. While designing our RINGBFT protocol, we set following goals:

(G1) Inexpensive consensus of single-shard transactions.

(G2) Flexibility of employing different existing consensus protocols for intra-shard consensus.

(G3) Deadlock-free two-ring consensus of deterministic cross-shard transactions.

(G4) Cheap communication between globally distributed shards.

We define the *safety* and *liveness* guarantees provided by our RINGBFT protocol.

Definition 4.1. Let $\mathfrak{S}$ be a system of shards and $\mathfrak{R}_S$ be a set of replicas in some shard $S \in \mathfrak{S}$. Each run of a *consensus protocol* in this system should satisfy the following requirements:

Involvement Each $S \in \mathfrak{S}$ processes a transaction if $S \in \mathfrak{I}$.

Termination Each non-faulty replica in $\mathfrak{R}_S$ executes a transaction.

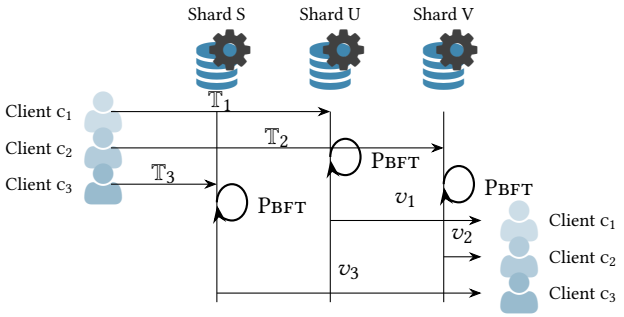

Figure 2: RingBFT consensus for single-shard transactions. Each of the three shards S, U, and V receive transactions $\mathbb{T}_1$, $\mathbb{T}_2$, and $\mathbb{T}_3$ from their respective clients c_1, c_2, and c_3 to execute. Each shard independently run Pbft consensus, and sends responses to respective clients.

Non-divergence (intra-shard) All non-faulty replicas in $\Re_S$ execute the same transaction.

Consistence (cross-shard) Each non-faulty replica in $\mathfrak{S}$ executes a conflicting transaction in same order.

In traditional replicated systems, non-divergence implies *safety*, while termination implies *liveness*. For a sharded-replicated system like RingBFT, we need stronger guarantees. If a transaction requires access to only one shard, safety is provided by involvement and non-divergence, while termination sufficiently guarantees liveness. For a cross-shard transaction, to guarantee safety, we also need consistency apart from involvement and non-divergence, while liveness is provided using involvement and termination.

RingBFT guarantees safety in an asynchronous setting. In such a setting, messages may get lost, delayed, or duplicated, and up to **f** replicas in each shard may act Byzantine. However, RingBFT can only provide liveness during periods of synchrony. Notice that these assumptions are no harder than those required by existing protocols [4, 10, 15].

4.1 Single-Shard Consensus

To order and execute single-shard transactions is trivial. For this task, RingBFT employs one of the many available primary-backup consensus protocols and runs them at each shard. In the rest of this section, without the loss of generality, we assume that RingBFT employs the Pbft consensus protocol to order single-shard transactions. We use the following example to explain RingBFT's single-shard consensus.

Example 4.2. Assume a system that comprises of three shards S, U, and V. Say client c_1 sends $\mathbb{T}_1$ to S, c_2 sends $\mathbb{T}_2$ to U, and client c_3 sends $\mathbb{T}_3$ to V. On receiving the client transaction, the primary of each shard initiates the Pbft consensus protocol among its replicas. Once each replica successfully orders the transaction, it sends a response to the client. Such a flow is depicted in Figure 2.

It is evident from Example 4.2 that there is no communication among the shards. This is the case because each transaction requires access to data available inside only one shard. Hence, ordering single-shard transactions for shard S requires running the Pbft protocol among the replicas of S without any synchronization with other shards.

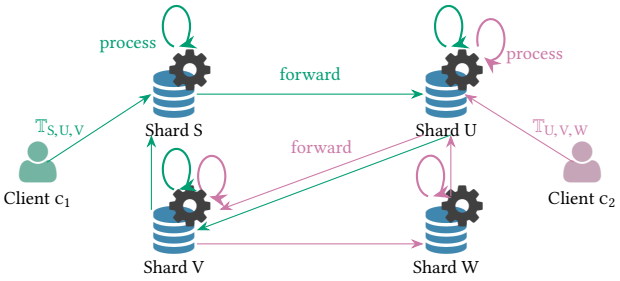

Figure 3: RingBFT's concurrent consensus of two cross-shard transactions $\mathbb{T}_{S,U,V}$ and $\mathbb{T}_{U,V,W}$ across four shards. The prescribed ring order is S → U → V → W.

4.2 Cross-Shard Consensus: Process & Forward

In this section, we illustrate how RingBFT guarantees consensus of every *deterministic* cross-shard transaction (cst) in *at most two rotations across the ring*. To order a cst, RingBFT requires shards to adhere to the *ring order*, and follow the principle of process, forward, and re-transmit while ensuring the communication between shards is *linear*. We use the following example to illustrate what we mean by following the ring order.

Example 4.3. Assume a system that comprises of four shards S, U, V, and W where the ring order has been defined as S → U → V → W. Say client c_1 wants to process a transaction $\mathbb{T}_{S,U,V}$ that requires access to data from shards S, U, and V, and client c_2 wants to process a transaction $\mathbb{T}_{U,V,W}$ that requires access to data from shards U, V, and W (refer to Figure 3). In this case, client c_1 sends its transaction to the primary of shard S while c_2 sends its transaction to primary of U. On receiving $\mathbb{T}_{S,U,V}$, replicas of S process the transaction and forward it to replicas of U. Next, replicas of U process $\mathbb{T}_{S,U,V}$ and forward it to replicas of V. Finally, replicas of V process $\mathbb{T}_{S,U,V}$ and send it back to replicas of S, which reply to client c_1. Similar flow takes place while ordering transaction $\mathbb{T}_{U,V,W}$.

Although Example 4.3 illustrates RingBFT's design, it is unclear how multiple concurrent cst are ordered in a deadlock-free manner. We explain this next.

4.2.1 Cross-shard Transactional Flow. RingBFT assumes shards are arranged in a logical ring. For the sake of explanation, we assume the ring order of *lowest to highest identifier*. For each cst, we denote one shard as the ***initiator shard***, which is responsible for starting consensus on the client transaction. How do we select the initiator shard? Of all the involved shards a cst accesses, the shard with the *lowest identifier in ring order* is denoted as the initiator shard.

RingBFT also guarantees consensus for each deterministic cst in at most two rotations across the ring. This implies that for achieving consensus on a deterministic cst, each involved shard $S \in \Im$ needs to process it at most two times. Notice that if a cst is simple, then a single rotation around the ring is sufficient to ensure that each involved shard S safely executes its fragment.

Prior to presenting our RingBFT's consensus protocol that safely orders each cst, we sketch the flow of a cst in Figure 4. In this figure, we assume a system of four shards: S, U, V, and W where id(S) < id(U) < id(V) < id(W). The client creates a transaction $\mathbb{T}_{S,U,W}$ that requires access to data in shards S, U, and W and sends this transaction to the primary P_S of S. On receiving

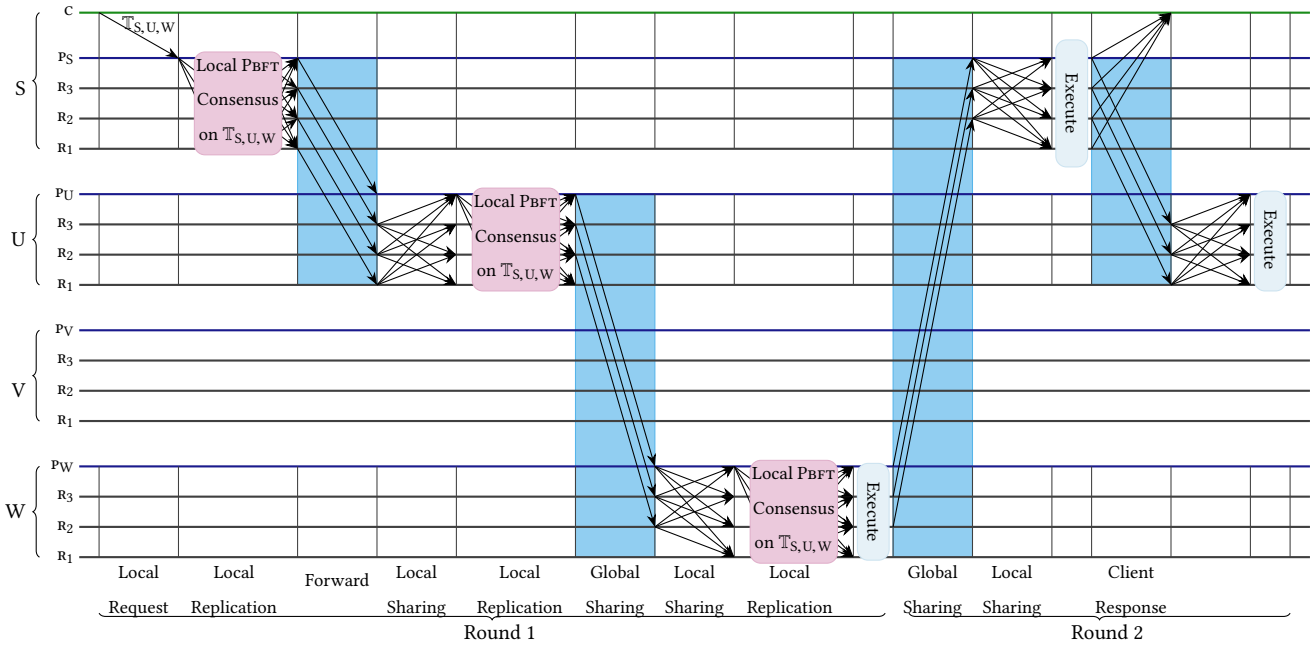

Figure 4: Representation of the normal-case flow of RingBFT in a system of four shards where client sends a cross-shard transaction $\mathbb{T}_{S,U,W}$ that requires access to data in *three* shards: S, U, and W.

this transaction, P_S initiates the PBFT consensus protocol (*local replication*) among its replicas. If the local replication is successful, then all the replicas of S ***lock*** the corresponding data. This locking of data-items in the ring-order helps in preventing deadlocks. Next, replicas of S forward the transaction to replicas of shard U. Notice that only *linear communication* takes place between replicas of S and U. Hence, to handle any failures, replicas of U share this message among themselves. Next, replicas of U also follow similar steps and forward transaction to W. As W is the last shard in the ring of involved shards, it goes ahead and executes the CST if all the dependencies are met. Finally, replicas of shards S and U also execute the transaction and replicas of S send the result of execution to the client.

4.3 Cross-Shard Consensus Algorithm

We use Figure 5 to present RingBFT's algorithm for ordering cross-shard transactions. Next, we discuss these steps in detail.

4.3.1 Client Request. When a client C wants to process a cross-shard transaction $\mathbb{T}_{\Im}$, it creates a $\langle \mathbb{T}_{\Im} \rangle_C$ message and sends it to the primary of the *first shard in ring order*. As part of this transaction, the client C specifies the information regarding all the involved shards ($\Im$), such as their identifiers and the necessary *read-write* sets of each shard. Notice that the client signs this message using DS to prevent repudiation attacks.

4.3.2 Client Request Reception. When the primary P_S of shard S receives a client request $\mathbb{T}_{\Im}$, it first checks if the message is well-formed. If this is the case, then P_S checks if among the set of involved shards $\Im$, S is the first shard in ring order. If this condition is met, then P_S assigns this request a linearly increasing sequence number k, calculates the digest Δ, and broadcasts a PREPREPARE message to all the replicas $\Re_S$ of its shard. In the case when S is not the first shard in the ring order, P_S forwards the transaction to the primary of the appropriate shard.

4.3.3 Pre-prepare Phase. When a replica $R \in \Re_S$ receives the PREPREPARE message from P_S, it checks if the request is well-formed. If this is the case and if R has not agreed to support any other request from P_S as the k-th request, then it broadcasts a PREPARE message in its shard S.

4.3.4 Prepare Phase. When a replica $R \in \Re_S$ receives identical PREPARE messages from **nf** distinct replicas, it gets an assurance that a majority of non-faulty replicas are supporting this request. At this point, each replica R broadcasts a COMMIT message to all the replicas in S. Once a transaction passes this phase, the replica R marks it *prepared*.

4.3.5 Commit and Data Locking. When a replica R receives well-formed identical COMMIT messages from **nf** distinct replicas in S, it checks if it also prepared this transaction at same sequence number. If this is the case, RingBFT requires each replica R to *lock* all the read-write sets that transaction $\mathbb{T}_{\Im}$ needs to access in shard S. In RingBFT, we allow replicas to process and broadcast PREPARE and COMMIT messages *out-of-order*, but require each replica to acquire locks on data in transactional sequence order. This out-of-ordering helps replicas to continuously perform useful work by concurrently participating in consensus of several transactions. To achieve these tasks, each replica R tracks the maximum sequence number ($k_{\max}$), which indicates the sequence number of the last transaction to lock data. If sequence number k for a transaction $\mathbb{T}_{\Im}$ is greater than $k_{\max} + 1$, we store the transaction in a list π until transaction at $k_{\max} + 1$ has acquired the locks. Once the $k_{\max} + 1$-th transaction has acquired locks, we gradually release transactions in π until there is a transaction that wishes to lock already locked data-fragments. We illustrate this through the following example.

Example 4.4. Assume the use of following notations for four transactions and the data-fragments they access at shard S: $\mathbb{T}_{1,a}$, $\mathbb{T}_{2,b}$, $\mathbb{T}_{3,a}$, and $\mathbb{T}_{4,c}$. For instance, $\mathbb{T}_{1,a}$ implies that transaction

Initialization:
// $k_{\max}$:= 0 (maximum sequence number in shard S)
// $\Sigma^{\mathfrak{I}} := \emptyset$ (set of data-fragments of each shard)
// $\pi := \emptyset$ (list of pending transactions at a replica)

Client-role (used by client c to request transaction $\mathbb{T}_{\mathfrak{I}}$) :
1: Sends $\langle \mathbb{T}_{\mathfrak{I}} \rangle_{\text{C}}$ to the primary P_{S} of shard S.
2: Awaits receipt of messages Response($\langle \mathbb{T}_{\mathfrak{I}} \rangle_{\text{C}}$, k, r) from $\mathbf{f}+1$ replicas of S.
3: Considers $\mathbb{T}_{\mathfrak{I}}$ executed, with result r, as the k-th transaction.

// This event is only triggered at the primary replica of each shard.
Primary-role (running at the primary P_{S} of shard S) :
4: **event** P_{S} receives $\langle \mathbb{T}_{\mathfrak{I}} \rangle_{\text{C}}$ **do**
5: **if** S $\in \mathfrak{I} \wedge$ id(S) = FirstInRingOrder($\mathfrak{I}$) **then**
6: Calculate digest $\Delta := H(\langle \mathbb{T} \rangle_{\text{C}})$.
7: Broadcast Preprepare($\langle \mathbb{T}_{\mathfrak{I}} \rangle_{\text{C}}$, Δ, k) in shard S (order at sequence k).
8: **else**
9: Send to primary P_{U} of shard U, U $\in \mathfrak{S} \wedge$ id(U) = FirstInRingOrder($\mathfrak{I}$)

// This event is only triggered at a non-primary replica.
Non-Primary Replica-role (running at the replica R of shard S) :
10: **event** R receives Preprepare($\langle \mathbb{T}_{\mathfrak{I}} \rangle_{\text{C}}$, Δ, k) from P_{S} such that:
message is well-formed, and R did not accept a k-th proposal from P_{S}. **do**
11: Broadcast Prepare(Δ, k) to replicas in $\mathfrak{R}_{\text{S}}$.

// Following events are triggered at every replica irrespective of whether it is the primary or a non-primary replica.
Replica-role (running at any replica R of shard S) :
12: **event** R receives well-formed Prepare(Δ, k) messages from **nf** replicas in S **do**
13: Broadcast $\langle \text{Commit}(\Delta, k) \rangle_{\text{R}}$ to replicas in $\mathfrak{R}_{\text{S}}$.

14: **event** R receives **nf** $m := \langle \text{Commit}(\Delta, k) \rangle_{\text{Q}}$ messages such that:
each message m is well-formed and is sent by a distinct replica Q $\in \mathfrak{R}_{\text{S}}$. **do**
15: U be the shard to forward such that id(U) = NextInRingOrder($\mathfrak{I}$).
16: A := set of DS of these **nf** messages.
17: **if** $k = k_{\max} + 1$ // Forward to next shard **then**
18: Lock data-fragment corresponding to $\langle \mathbb{T}_{\mathfrak{I}} \rangle_{\text{C}}$.
19: Send $\langle \text{Forward}(\langle \mathbb{T}_{\mathfrak{I}} \rangle_{\text{C}}, A, m, \Delta,) \rangle_{\text{R}}$ to replica O, where O $\in \mathfrak{R}_{\text{U}} \wedge$ id(R) = id(O)
20: **else**
21: Store $\langle \text{Forward}(\langle \mathbb{T}_{\mathfrak{I}} \rangle_{\text{C}}, A, m, \Delta,) \rangle_{\text{R}}$ in π.
22: **while** $\pi ! = \emptyset$ // Pop out waiting transaction. **do**
23: Extract transaction at $k_{\max} + 1$ from π (if any).
24: **if** Corresponding data-fragment is not locked **then**
25: $k_{\max} = k_{\max} + 1$
26: Follow lines 18 and 19.
27: **else**
28: Store transaction at $k_{\max}$ in π and *exit* the loop.

// Locally share any message from previous shard.
29: **event** R receives message $m := \langle \text{message-type} \rangle_{\text{Q}}$ such that:
m is well-formed and sent by replica Q, where
id(U) = PrevInRingOrder($\mathfrak{I}$), Q $\in \mathfrak{R}_{\text{U}} \wedge$ id(R) = id(Q) **do**
30: Broadcast m to all replicas in S.

// Forward message from previous shard.
31: **event** R receives $\mathbf{f}+1$ $m' := \langle \text{Forward}(\langle \mathbb{T}_{\mathfrak{I}} \rangle_{\text{C}}, A, m, \Delta) \rangle_{\text{Q}}$ such that:
each m' is well-formed; and set A includes valid DS from **nf** replicas for m. **do**
32: **if** Data-fragment corresponding to $\langle \mathbb{T}_{\mathfrak{I}} \rangle_{\text{C}}$ is locked // Second Rotation **then**
33: Execute data-fragment of $\langle \mathbb{T}_{\mathfrak{I}} \rangle_{\text{C}}$ and add to log.
34: Push result to set $\Sigma^{\mathfrak{I}}$.
35: Release the locks from corresponding data-fragment.
36: V be the shard to forward such that id(V) = NextInRingOrder($\mathfrak{I}$).
37: Send $\langle \text{Execute}(\Delta, \Sigma^{\mathfrak{I}}) \rangle_{\text{R}}$ to replica O, where O $\in \mathfrak{R}_{\text{V}} \wedge$ id(R) = id(O).
38: **else if** R = P_{S} // Primary initiates consensus **then**
39: Broadcast Preprepare($\langle \mathbb{T}_{\mathfrak{I}} \rangle_{\text{C}}$, Δ, k') in shard S (order at sequence k').

40: **event** R receives $m' := \langle \text{Execute}(\Delta, \Sigma^{\mathfrak{I}}) \rangle_{\text{Q}}$ such that:
m' is sent by replica Q, where Q $\in \mathfrak{R}_{\text{U}} \wedge$ id(R) = id(Q) **do**
41: **if** Already executed $\langle \mathbb{T}_{\mathfrak{I}} \rangle_{\text{C}}$ // Reply to client **then**
42: Send client C the result r.
43: **else**
44: Follow lines 33 to 37.

Figure 5: The event-based normal-case algorithm of RingBFT. Depending on the type of message the primary replica or a non-primary replica receives, specific events are triggered.

at sequence 1 requires access to data-item a. Next, due to out-of-order message processing, assume a replica R in S receives **nf** Commit messages for $\mathbb{T}_{2,b}$, $\mathbb{T}_{3,a}$, and $\mathbb{T}_{4,c}$ before $\mathbb{T}_{1,a}$. Hence, $\pi = \{\mathbb{T}_{2,b}, \mathbb{T}_{3,a}, \mathbb{T}_{4,c}\}$. Once R receiving **nf** Commit messages for $\mathbb{T}_{1,a}$, it locks data-item a and extracts $\mathbb{T}_{2,b}$ from π. As $\mathbb{T}_{2,b}$ wishes to lock a distinct data-item, so R continues processing $\mathbb{T}_{2,b}$. Next, R moves to $\mathbb{T}_{3,a}$ but it cannot process $\mathbb{T}_{3,a}$ due to lock-conflicts. Hence, it places back $\mathbb{T}_{3,a}$ in π and stops processing transactions in π until lock is available for $\mathbb{T}_{3,a}$.

Notice that if the client transaction $\mathbb{T}_{\mathfrak{I}}$ is a *single-shard* transaction, it requires access to data-items in only this shard. In such a case, this commit phase is the final phase of consensus and each replica executes $\mathbb{T}_{\mathfrak{I}}$ and replies to the client when the lock for the corresponding data-item is available.

4.3.6 Forward to next Shard via Linear Communication. Once a replica R in S locks the data corresponding to CST $\mathbb{T}_{\mathfrak{I}}$, it sends a Forward message to *only one* replica Q of the next shard in ring order. As one of the key goals of RingBFT is to ensure communication between two shards is linear, so we design a communication primitive that builds on top of the optimal bound for communication between two shards [33, 42]. We define RingBFT's cross-shard communication primitive as follows:

Linear Communication Primitive. In a system $\mathfrak{S}$ of shards, where each shard S, U $\in \mathfrak{S}$ has at most **f** Byzantine replicas, if each replica in shard S communicates with a distinct replica in shard U, then at least $\mathbf{f}+1$ non-faulty replicas from S will communicate with $\mathbf{f}+1$ non-faulty replicas in U.

Our linear communication primitive guarantees that to reliably communicate a message m between two shards requires only sending a linear number of messages in comparison to protocols like AHL and Sharper which require *quadratic* communication. Using this communication primitive, to communicate message m from shard S to shard U, we need to exchange only **n** messages.

So, how does RingBFT achieve this task? We require each replica of S to initiate communication with the replica of U having the same identifier. Hence, replica R of shard S sends a Forward message to replica Q in shard U such that id(R) = id(Q). By sending a Forward message, R is requesting Q to initiate consensus on $\langle \mathbb{T}_{\mathfrak{I}} \rangle_{\text{C}}$. For Q to support such a request, it needs a proof that $\langle \mathbb{T}_{\mathfrak{I}} \rangle_{\text{C}}$ was successfully ordered in shard S. Hence, R includes the DS on Commit messages from **nf** distinct replicas (Line 16).

Until now, we assumed that each shard has an equal number of replicas. If we forgo this assumption, it will not affect the intra-shard consensus, that is, the Bft consensus protocol running at each shard remains unchanged. Further, the transaction execution explained in the next section also remains unaffected. The only visible change occurs in our linear communication primitive. However, even this change does not impact the correctness of our RingBFT protocol as our linear communication primitive builds on the optimal bound for communication between two shards, which permits shards to have a different number of replicas while guaranteeing linear communication complexity [33, 42].

Finally, in Section 5.1.2, we illustrate how our linear communication primitive can handle attacks by byzantine replicas.

4.3.7 Execution and Final Rotation. Once a client request has been ordered on all the involved shards, we call it *one complete rotation* around the ring. This is a significant event because it implies that all the necessary data-fragments have been locked by each of the involved shards. If a CST is simple, then each shard can independently execute its fragment without any further communication between the shards. In the case a CST is complex, at the end of the first rotation, the replicas of the first shard in ring order (S) will receive a Forward message from the replicas of the last shard in ring order.

Next, the replicas of S will attempt to execute parts of transaction, which are their responsibility. Post execution, replicas

of S send Execute messages to the replicas in next shard using our communication primitive. Notice that the Execute message includes updated write sets ($\Sigma^{\Im}$), which help in resolving any dependencies during execution. Finally, when the execution is completed across all the shards, the first shard in ring order replies to the client.

5 UNCIVIL EXECUTIONS

In previous sections, we discussed transactional flows under the assumption that the network is stable and replicas will follow the stated protocol. However, any Byzantine-Fault Tolerant protocol should provide safety under asynchronous settings and liveness in the period of synchrony even if up to **f** replicas are Byzantine.

RingBFT offers safety in an asynchronous environment. To guarantee liveness during periods of synchrony, RingBFT offers several recovery protocols, such as *checkpoint*, *retransmission*, and *view-change*, to counter malicious attacks. The first step in recovery against any attack is *detection*. To do so, we require each replica R to employ a set of *timers*. When a timer at a replica R *timeouts*, then R initiates an appropriate recovery mechanism. In specific, each replica R sets following timers:

- **Local Timer** – To track successful replication of a transaction in its shard.
- **Transmit Timer** – To re-transmit a successfully replicated cross-shard transaction to next shard.
- **Remote Timer** – To track replication of a cross-shard transaction in the previous shard.

Each of these timers is initiated at the occurrence of a distinct event and its timeout leads to running a specific recovery mechanism. When a local timer expires, then the corresponding replica initiates replacement of the primary of its shard (view-change), while a remote timer timeout requires the replica to inform the previous shard in ring order about the insufficient communication. This brings us to following observation regarding the consensus offered by RingBFT:

Proposition 5.1. *If the network is reliable and the primary of each shard is non-faulty, then the Byzantine replicas in the system cannot affect the consensus protocol.*

Notice that Proposition 5.1 holds implicitly as no step in Figure 5 depends on the correct working of non-primary Byzantine replicas; in each shard S, local replication of each transaction is managed by the primary of S and communication between any two shards S and U involves all the replicas. This implies that we need only consider cases when the network is unreliable and/or primary is Byzantine. We know that RingBFT guarantees safety even in unreliable communication and requires a reliable network for assuring liveness. Hence, we will illustrate mechanisms to tackle attacks by Byzantine primaries. Next, we illustrate how RingBFT resolves all the possible attacks it encounters.

(A1) Client Behavior and Attacks. In the case, the primary is Byzantine and/or network is unreliable, client is the key entity at loss. Client requested the primary to process its transaction, but due to an ongoing Byzantine attack, client did not receive sufficient responses. Clearly, client cannot wait indefinitely to receive valid responses. Hence, we require each client C to start a timer when it sends its transaction $\mathbb{T}_{\Im}$ to the primary P_S of shard S. If the timer timeouts prior to C receiving at least $\mathbf{f}+1$ identical responses, C broadcasts $\mathbb{T}_{\Im}$ to all the replicas $R \in \mathfrak{R}_S$ of shard S.

When a non-primary replica R receives a transaction from C, it forwards that transaction to P_S and waits on a timer for P_S to initiate consensus on $\mathbb{T}_{\Im}$. During this time, R expects P_S to start consensus on at least one transaction from C, otherwise it initiates view-change protocol. Notice that a *Byzantine client* can always forward its request to all the replicas of some shard to *blame* a non-faulty primary. Such an attack will not succeed as if C sends to R an already executed request, R simply replies with the stored response. Moreover, if R belongs to some shard S, which is not the first shard in ring order, then R ignores the client transaction.

(A2) Faulty Primary and/or Unreliable network. A faulty primary can prevent successful consensus of a client transaction. Such a primary can be trivially detected as at most **f** non-faulty replicas would have successfully *committed* the transaction (received at least $\mathbf{n}-\mathbf{f}$ Commit messages).

An unreliable network can cause messages to get lost or indefinitely delayed. Such an attack is difficult to detect and non-faulty replicas may blame the primary.

Each primary represents a *view* of a shard. Hence, the term view-change is often used to imply primary replacement. Notice that each shard in RingBFT is a replicated system. Further, RingBFT is a meta-protocol, which employs existing Bft protocols, such as Pbft, to run consensus. These properties allow RingBFT to use the accompanying *view-change protocol*. Specifically, in this paper, we use Pbft's view change protocol (for MAC-based authentication) to detect and replace a faulty primary [11].

A replica $R \in \mathfrak{R}_S$ initiates the view-change protocol to replace its primary P_S in response to a timeout. As discussed earlier in this section, there are two main causes for such timeouts: (i) R does not receive **nf** identical Commit messages from distinct replicas, and (ii) P_S fails to propose a request from client C.

(A3) Malicious Primary. A malicious primary P can ensure that up to **f** non-faulty replicas in its shard S are unable to make progress (in *dark*). Under such conditions, the affected non-faulty replicas will request a view-change, but they will not be successful as the next primary may not receive sufficient ViewChange messages (from at least **nf** replicas) to initiate a new view. Further, the remaining $\mathbf{f}+1$ non-faulty replicas will not support such ViewChange requests as it is impossible for them to distinguish between this set of **f** non-faulty replicas and the actual **f** Byzantine replicas.

To ensure these replicas in dark make progress, traditional protocols periodically send checkpoint messages. These checkpoint messages include all client transactions and the corresponding **nf** Commit messages since the last checkpoint.

5.1 Cross-Shard Attacks

Until now, we have discussed attacks that can be resolved by replicas of any shard independent of the functioning of other shards. However, the existence of cross-shard transactions unravels new attacks, which may span multiple shards. We use the term *cross-shard attacks* to denote attacks that thwart successful consensus of a cst, First, we describe such attacks, and then we present solutions to recover from these attacks.

In RingBFT, we know that the consensus of each cst follows a ring order. In specific, for a cross-shard transaction $\mathbb{T}_{\Im}$, each of its involved shards $S, U \in \Im$ first run a local consensus and then communicate the data to the next shard in ring order. Earlier in this section, we observed that if at least $\mathbf{f}+1$ non-faulty replicas of any shard are unable to reach consensus on $\mathbb{T}_{\Im}$, then that shard will undergo local view-change. Hence, we are interested in those cross-shard attacks where *neither the involved shards are*

Replica-role (running at the replica Q of shard U) :

1: **event** Remote timer of Q timeouts such that:
Q has received at most **f** $\langle$FORWARD($\langle \mathbb{T}_{\mathfrak{I}} \rangle_C$, *A*, *m*, Δ,)$\rangle_R$ messages, where
id(S) = PrevInRingOrder($\mathfrak{I}$), R ∈ $\mathfrak{R}_S$ **do**
2: Send $\langle$REMOTEVIEW($\langle \mathbb{T}_{\mathfrak{I}} \rangle_C$, Δ)$\rangle_Q$ to replica O, where O ∈ $\mathfrak{R}_S$ ∧ id(Q) = id(O)

3: **event** R receives message *m* := $\langle$REMOTEVIEW($\langle \mathbb{T}_{\mathfrak{I}} \rangle_C$, Δ)$\rangle_Q$ such that:
m is well-formed and sent by replica Q, where
id(U) = NextInRingOrder($\mathfrak{I}$), Q ∈ $\mathfrak{R}_U$ ∧ id(R) = id(Q) **do**
4: Broadcast *m* to all replicas in S.

5: **event** R receives **f** + 1 $\langle$REMOTEVIEW($\langle \mathbb{T}_{\mathfrak{I}} \rangle_C$, Δ)$\rangle_Q$ messages **do**
6: Initiate Local view-change protocol.

Figure 6: The remote view-change algorithm of RingBFT.

able to trigger local view change by themselves, nor are they able to execute the transaction and reply to the client. This can only occur when all the involved shards of a cross-shard transaction $\mathbb{T}_{\mathfrak{I}}$, either successfully completed consensus on $\mathbb{T}_{\mathfrak{I}}$, or are unable to initiate the consensus on $\mathbb{T}_{\mathfrak{I}}$. Next, we describe these attacks.

Assume $\mathfrak{R}_S$ and $\mathfrak{R}_U$ represent the sets of replicas in shards S and U, respectively.

(C1) **No Communication.** Under a no communication attack, we expect that the replicas in $\mathfrak{R}_S$ are unable to send any messages to replicas of $\mathfrak{R}_U$.

(C2) **Partial Communication.** Under a partial communication attack, we expect that at least **f** + 1 replicas in $\mathfrak{R}_U$ receive less than **f** + 1 FORWARD messages from replicas in $\mathfrak{R}_S$.

Both of these attacks could occur solely due to an unreliable network that causes message loss or indefinite message delays. Further, a malicious primary can collude with an adversarial network to accelerate the frequency of such attacks. In either of the cases, to recover from such cross-shard attacks, all the involved shards may need to communicate among themselves.

5.1.1 ***Message Retransmission.*** In RingBFT, to handle a *no communication* attack, affected replicas of the preceding shard retransmit their original message to the next shard in ring order. Specifically, when a replica R of shard S successfully completes the consensus on transaction $\mathbb{T}_{\mathfrak{I}}$, it sets the *transmit timer* for this request prior to sending the FORWARD message to replica Q of shard U (next shard in ring order). When the transmit timer of R timeouts, it again sends the FORWARD message to Q.

5.1.2 ***Remote View Change.*** A *partial communication* attack could be either due to a Byzantine primary or unreliable network. If the primary P_S of shard S is Byzantine, then it will ensure that at most **f** non-faulty replicas replicate a cross-shard transaction $\mathbb{T}_{\mathfrak{I}}$ (S, U ∈ $\mathfrak{I}$), locally. As a result, replicas of next shard U will receive at most **f** FORWARD messages. Another case is where the network is unreliable, and under such conditions, replicas of U may again receive at most **f** FORWARD messages.

From Figure 5, we know that when replica Q of shard U receives a FORWARD message from replica R of shard S such that id(R) = id(Q), then Q broadcasts this FORWARD message to all the replicas in U. At this point, RingBFT also requires replica Q to start the *remote timer*. If any replica Q in shard U does not receive identical FORWARD messages from **f** + 1 distinct replicas of shard S, prior to the timeout of its remote timer, then Q detects a cross-shard attack and sends a REMOTEVIEW message to the replica R of shard S, where id(R) = id(Q). Following this, R broadcasts the received REMOTEVIEW message to all the replicas in S. Finally, when any replica R of shard S receives REMOTEVIEW messages from **f** + 1 replicas of U, it supports the view change request and initiates the view-change protocol. We illustrate this process in Figure 6.

Triggering of Timers. In RingBFT, we know that for each cross-shard transaction, each replica R of S sets three distinct timers. Although each timer helps in recovering against a specific attack, there needs to be an order in which they timeout. As local timers lead to detecting a local malicious primary, we expect a local timer to have the shortest duration. Further, a remote timer helps to detect a lack of communication due to which it has a longer duration than local timers. Similarly, we require the duration of retransmit timer to be the longest.

6 RINGBFT GUARANTEES

We now state the safety, liveness, and no deadlock guarantees provided by our RingBFT protocol.

PROPOSITION 6.1. *Let R_i, $i \in \{1, 2\}$, be two non-faulty replicas in shard S that committed to $\langle \mathbb{T}_i \rangle_{C_i}$ as the k-th transaction sent by P. If* $\mathbf{n} > 3\mathbf{f}$, *then* $\langle \mathbb{T}_1 \rangle_{C_1} = \langle \mathbb{T}_2 \rangle_{C_2}$.

PROOF. Replica R_i only committed to $\langle \mathbb{T} \rangle_{C_i}$ after R_i received identical COMMIT(Δ, k) messages from **nf** distinct replicas in S. Let X_i be the set of such **nf** replicas and $Y_i = X_i \setminus \mathcal{F}$ be the non-faulty replicas in X_i. As $|\mathcal{F}| = \mathbf{f}$, so $|Y_i| \geq \mathbf{nf} - \mathbf{f}$. We know that each non-faulty replica only supports one transaction from primary P as the k-th transaction, and it will send only one PREPARE message. This implies that sets Y_1 and Y_2 must not overlap. Hence, $|X_1 \cup X_2| \geq 2(\mathbf{nf} - \mathbf{f})$. As $|X_1 \cup X_2| = \mathbf{nf}$, the above inequality simplifies to $3\mathbf{f} \geq \mathbf{n}$, which contradicts $\mathbf{n} > 3\mathbf{f}$. Thus, we conclude $\langle \mathbb{T}_1 \rangle_{C_1} = \langle \mathbb{T}_2 \rangle_{C_2}$. □

THEOREM 6.2. ***No Deadlock:*** *In a system $\mathfrak{S}$ of shards, where S, U ∈ $\mathfrak{S}$ and S ≠ U, no two replicas R ∈ S and Q ∈ U that order two conflicting transactions $\mathbb{T}_{\mathfrak{I}_1}$ and $\mathbb{T}_{\mathfrak{I}_2}$ such that S, U ∈ $\mathfrak{I}_1 \cap \mathfrak{I}_2$ will execute $\mathbb{T}_{\mathfrak{I}_1}$ and $\mathbb{T}_{\mathfrak{I}_2}$ in different orders.*

PROOF. We know that RingBFT associates an identifier with each shard and uses this identifier to define a ring order. Let id(S) < id(U), and the ring order be defined as lowest to highest identifier. Assume that the conflicting transactions $\mathbb{T}_{\mathfrak{I}_1}$ and $\mathbb{T}_{\mathfrak{I}_2}$ are in a deadlock at shards S and U, where S, U ∈ $\mathfrak{I}_1 \cap \mathfrak{I}_2$. This implies that each non-faulty replica R ∈ S has locked some data-item for $\mathbb{T}_{\mathfrak{I}_1}$ that is required by $\mathbb{T}_{\mathfrak{I}_2}$ while each non-faulty replica Q ∈ U has locked some data-item for $\mathbb{T}_{\mathfrak{I}_2}$ that is required by $\mathbb{T}_{\mathfrak{I}_1}$ or vice versa.

As each transaction $\mathbb{T}_{\mathfrak{I}_i}$, $i \in [1, 2]$ accesses S and U in ring order, so each transaction $\mathbb{T}_{\mathfrak{I}_i}$ was initiated by S. This implies that the primary of S would have assigned these transactions distinct sequence numbers $k_i, i \in [1, 2]$, such that $k_1 < k_2$ or $k_1 > k_2$ ($k_1 = k_2$ is not possible as it will be detected as a Byzantine attack). During the commit phase, each replica R will put the transaction with larger sequence number k_i in the π list and lock the corresponding data-item (Figure 5, Line 23), while the transaction with smaller k_i is forwarded to the next shard U. The transaction present in the π list is only extracted once the data-item is unlocked. Hence, there is a contradiction, that is, shards S and U will not suffer deadlock. □

THEOREM 6.3. ***Safety:*** *In a system $\mathfrak{S}$ of shards, where each shard S ∈ $\mathfrak{S}$ has at most* **f** *Byzantine replicas, each replica R follows the Involvement, Non-divergence, and Consistence properties. Specifically, all the replicas of S execute each transaction in the same order, and every conflicting cross-shard transaction is executed by all the replicas of all the involved shards in the same order.*

Proof. Using Proposition 6.1 we have already illustrated that RingBFT safely replicates a single-shard transaction, despite a malicious primary and/or unreliable network. In specific, any non-faulty replica $R \in \Re_S$ will only commit a single-shard transaction if it receives Commit messages from **nf** distinct replicas in $\Re_S$. When a non-faulty replica receives less than **nf** Commit messages, then eventually its local timer will timeout and it will participate in the view-change protocol. Post the view-change protocol, any request that was committed by at least one non-faulty replica will persist across views.

Similarly, we can show that each cross-shard transaction is also safely replicated across all replicas of all the involved shards. In RingBFT, each cross-shard transaction $\mathbb{T}_\Im$ is processed in ring order by all the involved shards $\Im$. Let shards $S, U \in \Im$ and $id(S) < id(U)$ such that ring order is based on lowest to highest identifier. Hence, replicas of shard U will only start consensus on $\mathbb{T}_\Im$ if they receive Forward messages from $\mathbf{f} + 1$ distinct replicas of S. Further, each of these Forward messages includes DS from **nf** distinct replicas of S on identical Commit messages corresponding to $\mathbb{T}_\Im$, which guarantees that $\mathbb{T}_\Im$ was replicated in S. If the network is unreliable and/or primary of shard S is Byzantine, then replicas of U will receive less than $\mathbf{f}+1$ Forward messages. In such a case, either the remote timer at replicas of U will timeout, or one of the two timers (local timer or transmit timer) of replicas of S will timeout. In any case, following the specific recovery procedure, replicas of U will receive a sufficient number of Forward messages. □

Theorem 6.4. ***Liveness:*** *In a system $\Im$ of shards, where each shard $S \in \Im$ has at most* **f** *Byzantine replica, if the network is reliable, then each replica $\mathcal{R}$ follows the Involvement and Termination properties. Specifically, all the replicas continue making progress, and good clients continue receiving responses for their transactions.*

Proof. In the case of a single-shard transaction, if the primary is non-faulty, then each replica will continue processing client transactions. If the primary is faulty, and prevents a request from replicating by allowing at most **f** replicas to receive Commit messages, then such a primary will be replaced through view-change protocol, following which a new primary will ensure that the replicas continue processing subsequent transactions. Notice that there can be at most **f** such faulty primaries, and the system will eventually make progress. If the primary is malicious, then it can keep up to **f** non-faulty replicas in dark, which will continue making progress through periodic checkpoints. In the case of a cross-shard transaction, there is nothing extra that a faulty primary P_S can do than preventing local replication of the transaction. If P_S does that, then as discussed above, P_S will be replaced. Further, during any communication between two shards, primary has no extra advantage over other replicas in the system. Further, the existence of transmit and remote timers help replicas of all the involved shards to keep track of any malicious act by primaries. □

7 EVALUATION

In this section, we evaluate our RingBFT protocol. To do so, we implement RingBFT on our high throughput yielding permissioned blockchain fabric, ResilientDB [26–29, 32, 33, 35, 58]. ResilientDB associated a multi-threaded pipelined architecture with each replica, which helps to optimally design a consensus protocol. For implementation details, we refer the reader to the full version of this paper [59].

For experimentation, we deploy ResilientDB on Google Cloud Platform (GCP) in ***fifteen regions*** across ***five continents***, namely: Oregon, Iowa, Montreal, Netherlands, Taiwan, Sydney, Singapore, South Carolina, North Virginia, Los Angeles, Las Vegas, London, Belgium, Tokyo, and Hong Kong. In any experiment involving less than 15 shards, the choice of the shards is in the order we have mentioned above. We deploy each replica on a 16-core N1 machine having Intel Broadwell CPUs with a 2.2GHz clock and 32GB RAM. For deploying clients, we use the 4-core variants having 16GB RAM. For each experiment, we equally distribute the clients in all regions.

Benchmark. To provide workload for our experiments, we use the *Yahoo Cloud Serving Benchmark* from the BlockBench suite (YCSB) [13, 18] . Each client transaction queries a YCSB table with an active set of 600 k records. For our evaluation, we adopt transactions that read and modify existing records. Prior to each experiment, each replica initializes an identical copy of the YCSB table. YCSB workloads help us to create cross-shard client transactions with varying degrees of conflict, while other workloads aim to evaluate the cost of executing a transaction, which is orthogonal to our RingBFT consensus.

Existing Protocols. In all our experiments, we compare the performance of RingBFT against two other state-of-the-art sharding Bft protocols, AHL [15] and Sharper [4]. In Section 2, we highlighted key properties of these protocols. Like RingBFT, both AHL and Sharper employ Pbft to achieve consensus on single-shard transactions. Hence, all three protocols have identical implementations for replicating single-shard transactions. For achieving consensus on cross-shard transactions, we follow the respective algorithms and modify ResilientDB appropriately.

WAN Bandwidth and Round-Trip Costs. As the majority of experiments take place in a geo-scaled WAN environment spanning multiple continents, available bandwidth and round-trip costs between two regions play a crucial role. Prior works [2, 33] have illustrated that if the available bandwidth is low and round-trip costs are high, then the protocols dependent on a subset of replicas face performance degradation. In the case of AHL, the reference committee is responsible for managing cross-shard consensus, while for Sharper, the primary of coordinating shard leads the cross-shard consensus. Hence, both of these protocols observe low throughput and high latency in proportion to available bandwidth and round-trip costs. Although RingBFT requires cross-shard communication in the form of Forward and Execute messages, the system is comparably less burdened as all the replicas participate equally in this communication process.

Standard Settings. Unless *explicitly* stated, we use the following settings for all our experiments. We run with a mixture of single-shard and cross-shard transactions, of which 30% are cross-shard transactions. Each cross-shard transaction accesses all the 15 regions, and in each shard we deploy 28 replicas, that is, a total of 420 globally distributed replicas. The number of key-value pairs accessed by each transaction varies in accordance with the number of regions accessed. For example, if a transaction accesses three regions, then it accesses three key-value pairs. In these experiments, we allow up to 50K clients to send transactions. Further, we require clients and replicas to employ batching and create batches of transactions of size 100.

The sizes of messages communicated during RingBFT consensus are: Preprepare (5408B), Prepare (216B), Commit (269B), Forward (6147B), Checkpoint (164B), and Execute (1732B).

Note: Our RingBFT protocol provides support for standard multi-statement transactions that are widely adopted by deterministic databases [36, 39, 57, 60, 66]. Hence, the complexity of designing RingBFT is similar to running an application on top of a deterministic database. Hence, we believe a developer would not face any new challenges.

Through our experiments, we want to answer the following:

(Q1) What is the effect of increasing the number of shards on consensus provided by RingBFT?

(Q2) How does varying the number of replicas per shard affects the performance of RingBFT?

(Q3) What is the impact of increasing the percentage of cross-shard transactions on RingBFT?

(Q4) How does batching affect the system performance?

(Q5) What is the effect of varying the number of involved shards in a cross-shard transaction on RingBFT?

(Q6) What is the impact of varying number of clients on consensus provided by RingBFT?

(Q7) How do faulty primary and view change affect the performance of RingBFT?

Scaling Number of Shards. For our first set of experiments, we study the effect of scaling the number of shards. In specific, we require clients to send cross-shard transactions that can access from 3, 5, 7, 9, 11, and 15 shards, while keeping other parameters at the standard setting. We use Figures 7 (I) and (II) to illustrate the throughput and latency metrics.

RingBFT achieves 16× and 4× higher throughput than AHL and Sharper in the 15 shard setting, respectively. An increase in the number of shards only increases the length of the ring while keeping the amount of communication between two shards at constant. As a result, for RingBFT, we observe an increase in latency as there is an increase in time to go around the ring, namely, a linear neighbor-to-neighbor communication. From three shards to 15 shards, the latency increases from 1.17**s** to 6.82**s**. Notice that the throughput for RingBFT is nearly constant since the size of shards and the amount of communication among shards are constant. This is a consequence of an increase in the number of shards that can perform consensus on single-shard transactions in parallel. Although on increasing the number of shards, there is a proportional increase in the number of involved shards per transaction, the linear communication pattern of RingBFT prevents throughput degradation.

In the case of AHL, the consensus on cross-shard transactions is led by the reference committee, which essentially centralizes the communication in the global setting and affects the system performance. In contrast, Sharper scales better because there is no single reference committee leading all cross-shard consensuses. However, even Sharper sees a fall in throughput due to two rounds of communication between all replicas of all the involved shards. For a system where all the shards are globally scattered, quadratic communication complexity and communication between all the shards impacts the scalability of the system.

Scaling Number of Replicas per Shard. We now study the effects of varying different parameters within a single shard. Our next set of experiments aim to increase the amount of replication within a single shard. In specific, we allow each shard to have 10, 16, 22, and 28 replicas. We use Figures 7 (III) and (IV) to illustrate the throughput and latency metrics.

These plots reaffirm our theory that RingBFT ensures up to 16× higher throughput and 11× lower latency than the other two protocols. As the number of replicas in each shard increases, there is a corresponding decrease in throughput for RingBFT. This decrease is not surprising because RingBFT employs the Pbft protocol for local replication, which necessitates two phases of quadratic communication complexity. This, in turn increases the size (and as a result cost) of Forward messages communicated between shards.

In the case of AHL, the existence of a reference committee acts as a performance bottleneck to an extent that 30% cross-shard transactions involving all the 15 shards subsides the benefits due to reduced replication (10 or 16 replicas). Sharper also observes a drop in its performance as it relies on Pbft, and is unable to scale at smaller configurations due to expensive communication that requires an all-to-all communication between the replicas of involved shards. ***To summarize:*** RingBFT achieves up to 4× and 16× higher throughput than Sharper and AHL, respectively.

Varying percentage of Cross-shard Transactions. For our next study, we allow client workloads to have 0, 5%, 10%, 15%, 30%, 60%, and 100% cross-shard transactions. We use Figures 7 (V) and (VI) to illustrate the throughput and latency metrics.

When the workload contains no cross-shard transactions, it simply indicates a system where all the transactions access only one shard. In this case, all the three protocols attain the same throughput and latency as all of them employ Pbft for reaching consensus on single-shard transactions. They achieve **1.2 Million** txn/s throughput among 500 nodes in 15 globally distributed regions. With a small (5%) introduction of cross-shard transactions in the workload, there is a significant decrease for all the protocols. The amount of decrease is in accordance to the reasons we discussed in previous sections. However, RingBFT continues to outperform other protocols. In the extreme case of 100% cross-shard workload, RingBFTachieve 4× and 18× higher throughput and 3.3× and 7.8× lower latency than Sharper and AHL, respectively.

Varying the Batch Size. Next, we study the impact of batching transactions on system performance. We require the three protocols to run consensus on batches of client transactions with sizes 10, 50, 100, 500, 1K, and 5K. We use Figures 7 (VII) and (VIII) to illustrate the throughput and latency metrics.

As the number of transactions in a batch increases, there is a proportional decrease in the number of consensuses. For example, with a batch size of 10 and 100 for 5000 transactions, we need 500 and 50 instances of consensus. However, larger batches also cause an increase in latency due to the increased cost of communication and time for processing all the transactions in the batch. Hence, we observe an increase in throughput on moving from small batches of 10 transactions to large batches of 1K transactions. On further increase (after 1.5K), the system throughput hits saturation and eventually decreases as benefits of batching are over-shadowed by increased communication costs.

Starting from the batch size of 10, on increasing the batch size, the throughput increases up to 27× in RingBFT because, with less communication and fewer messages, we are processing more transactions. This trend lasts until the system reaches its saturation point in terms of communication and computation, which is the batch size of 1.5K for RingBFT. Once the system is at filling its network bandwidth, adding more transactions to the batch will not increase the throughput because it cannot process more, and sending those batches will be a bottleneck for the system. Ideally, it should get constant after some point but because of implementation details and queuing, it drops slightly after some time.

Ideally, we expect the latency to also decrease with an increase in batch size. However, for RingBFT, more transactions in a batch

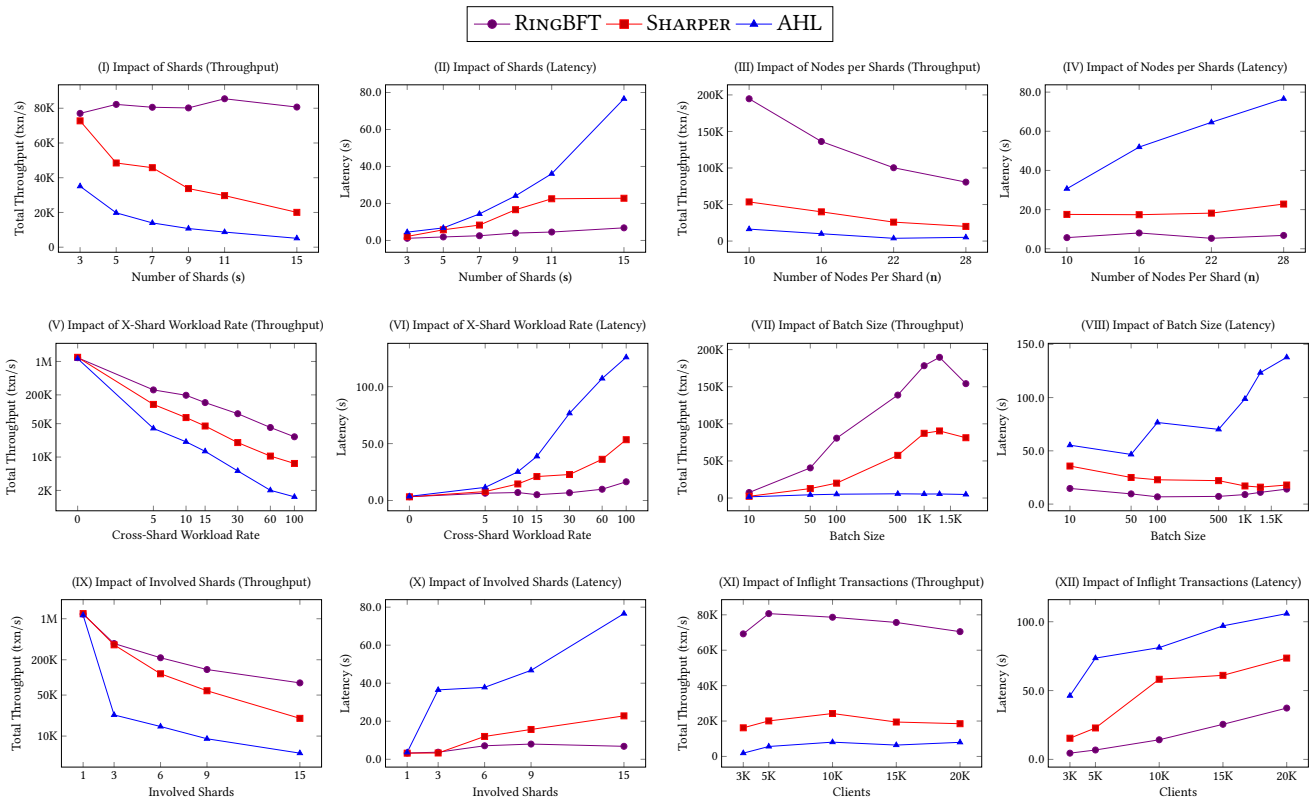

Figure 7: Measuring system throughput and average latency on running different BFT sharding consensus protocols.

implies more time spent processing the transactions around the ring. This causes an increase in latency for the client. ***To summarize:*** Using the optimal batch size improve the throughput of RingBFT, Sharper and AHL, 27×, 45×, and 3× respectively.

Varying Number of Involved Shards. We now keep the number of shards fixed at 15 and require all clients to create transactions that access a subset of these shards. In specific, clients send transactions that access 1, 3, 6, 9, and 15 shards. As our selected order for shards gives no preference to their proximity to each other (to prevent any bias), our clients select consecutive shards in order to generate the workload.

We use Figures 7 (IX) and (X) to illustrate the throughput and latency metrics. As expected, all three protocols observe a drop in performance on the increase in the number of involved shards. However, RingBFT still outperforms the other two protocols. As we increase the number of involved shards, the performance gap between RingBFT and the other two protocols increases. As shown in the graph, with three shards involved, RingBFT has a 4% performance gap, increasing to 4× with 15 shards involved.

Varying Number of Clients. Each system can reach optimal latency only if it is not overwhelmed by incoming client requests. In this section, we study the impact of the same by varying the number of incoming client transactions through a gradual increase in the number of clients from 5K to 20K. We use Figures 7 (XI) and (XII) to illustrate resulting throughput and latency metrics. As we increase the number of clients transmitting transactions, we observe a 15 – 20% increase in throughput, reaching the saturation point. Having more clients causes a decrease between 7% and 9%, which is a result of various queues being full with incoming requests, which in turn causes a replica to perform extensive memory management. Due to similar reasons, there is a significant increase in latency as the time to process each request has increased proportionally. We observed 32.75s, 58.21s, and 59.64s increase in RingBFT, Sharper, and AHL respectively. Despite this, RingBFT scales better than other protocols even when the system is overwhelmed by clients.

Impact of Primary Failure. Next, we evaluate the effect of replacing a faulty primary in different shards. For this experiment, we run experiments with 9 shards and allow workload to consist of 30% cross-shard transactions. We use Figure 8 to show the throughput attained by RingBFT when the primary of the first three shards fail, and the replicas run the view change protocol to replace the faulty primary. The primaries of these shards fail at 10s, and the system's average throughput starts decreasing while other shards are processing their clients' requests. RingBFT observes a 15% decrease in throughput and post view change; it again observes an increase in throughput.

Impact of Complex Cross-Shard Transactions. Until now, we have experimented with simple CST where for a given CST each shard could independently execute its data-fragment. However, a sharded system may encounter a complex CST where each shard may require access to data (and needs to check constraints) present in other shards while executing its data-fragment. These data-access dependencies require each shard to read the data from remote shards.

Our RingBFT protocol performs this task by requiring each shard to send its read-write sets along with the Forward message. In this section, we study the cost of communicating the read-write sets of a complex CST on our RingBFT protocol. We use Figure 9 to illustrate the throughput and latency metrics on varying the number of data-access dependencies from 0 to 64 distributed randomly across 15 shards. These figures illustrate

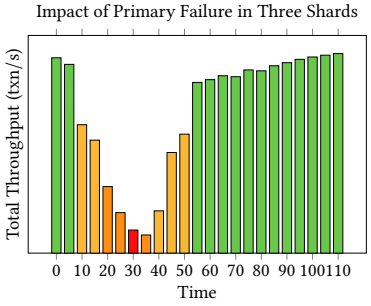

Figure 8: RingBFT's throughput under the primary failure of three shards out of nine. (s = 10) primary fails; (s = 20) replicas timeout and send view-change messages; (s = 30) new primary starts the new view; (s = 35) system's throughput start increasing and returns back to normal at s = 55.

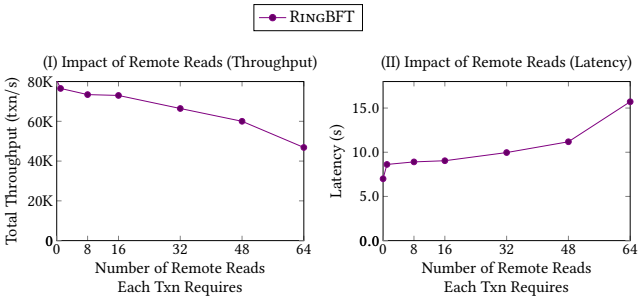

Figure 9: RingBFT's throughput and latency on encountering complex cross-shard transactions with dependencies varying from 0 to 64.

that our RingBFT protocol provides reasonable throughput and latency even for a CST with extensive dependencies.

Note that we have not included Sharper and AHL in Figure 9 as supporting complex CST is not covered in [4, 15] and remains as an open problem. For example, to support remote reads, first, there must be a consensus on the remote shard to agree on the requested operations and their values. Second, on the receiving end, there must be another local consensus on the values received. If the remote values are not received, then a consensus is needed to detect failures in order to invoke remote recovery to restore liveness. Now Sharper has a single global consensus that coordinates among all shards and their replicas. Thus, extending Sharper is nontrivial because it is unclear as to when and how the additional remote consensuses and recoveries could be invoked. In the case of AHL, due to its 2PC design, invoking remote consensus on each shard to process remote read is simple, but it is challenging to invoke remote view change when the network is unreliable or the primary of the remote shard behaves maliciously. Moreover, keeping the question of feasibility aside, we observe that in Figure 7(I), at 15 shards, the throughputs of both Sharper and AHL are under 20K while RingBFT sustains 80K transactions/second. However, in Figure 9, when we scale up to 64 remote operations across 15 shards, RingBFT yields a throughput of at least 45K transactions/second, surpassing both baselines with no remote operations.

8 RELATED WORK

In Section 1, we presented an overview of different types of Bft protocols. Further, we have extensively studied the architecture of state-of-the-art permissioned sharding Bft protocols, AHL and Sharper. We now summarize other works in the space of Byzantine Fault-Tolerance consensus.

Traditional Bft *consensus* The consensus problems such as Byzantine Agreement and Interactive Consistency have been studied in literature in great detail [19–22, 64, 65]. With the introduction of Pbft-powered *BFS*—a fault-tolerant version of the networked file system [40]—by Castro et al. [10, 11] there has been an unprecedented interest in the design of high-performance Bft consensus protocols. This has led to the design of several consensus protocols that have optimized different aspects of Pbft, e.g, Zyzzyva, Sbft, and PoE, as discussed in the Introduction. To further improve on the performance of Pbft, some consensus protocols consider providing less failure resilience [1, 51, 52], focused on a theoretical framework to support weaker consistency and isolation semantics such as dirty reads and committed reads [43], or rely on trusted components [6, 12, 34].

Bahsoun et al. [5] presented an adaptive Bft protocol that permits switching the type of Bft consensus protocol as per the system requirements. Guerraoui et al. [25] introduced the StretchingBFT protocol that aims to improve on Pbft by arranging replicas in a ring-like topology where each replica communicates with its two neighbors. Our RingBFT is a meta-protocol that can utilize any of these Bft protocol to achieve optimal intra-shard consensus. Hence, these protocols complement our design. Further, these protocols cannot scale to hundreds of replicas scattered across the globe, and this is where our vision of RingBFT acts as a resolve.

Permissionless Sharded Blockchains Permissionless space includes several sharding Bft consensus protocols, such as Conflux [49], Elastico [50], MeshCash [7], OmniLedger [45], and Spectre [63]. All of these protocols require each of their shards to run either the Proof-of-Work or Proof-of-Stake protocol during some phase of the consensus. As a result these protocols offer a magnitude lower throughput than both AHL and Sharper, which are included in our evaluation.

In our recent sharding work, we have developed a comprehensive theoretical framework to study a wide range of consistency models and isolation semantics (e.g., dirty reads, committed reads, serializability) and communication patterns (e.g., centralized vs. distributed) [43]. We have further developed a hybrid sharding protocol intended for the permissionless setting optimized for the widely used unspent transaction model [41].

9 CONCLUSIONS

In this paper, we present RingBFT–a novel meta-Bft protocol for permissioned sharded blockchains. For a single-shard transaction, RingBFT performs as efficient as any state-of-the-art sharding Bft consensus protocol. However, existing sharding Bft protocols face severe fall in throughput when they have to achieve consensus on a cross-shard transaction. RingBFT resolves this situation by requiring each shard to participate in at most two rotations around the ring. In specific, RingBFT expects each shard to adhere to the prescribed ring order, and follow the *principle of process, forward, and re-transmit*, while ensuring the communication between shards is linear. We implement RingBFT on our efficient ResilientDB fabric, and evaluate it against state-of-the-art sharding Bft protocols. Our results illustrates that RingBFT achieves up to 18× higher throughput than the most recent sharding protocols and easily scales to nearly 500 globally-distributed nodes.

REFERENCES

[1] Michael Abd-El-Malek, Gregory R. Ganger, Garth R. Goodson, Michael K. Reiter, and Jay J. Wylie. 2005. Fault-scalable Byzantine Fault-tolerant Services. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*. ACM, 59–74. https://doi.org/10.1145/1095810.1095817

[2] Yair Amir, Claudiu Danilov, Jonathan Kirsch, John Lane, Danny Dolev, Cristina Nita-Rotaru, Josh Olsen, and David Zage. 2006. Scaling Byzantine Fault-Tolerant Replication to Wide Area Networks. In *International Conference on Dependable Systems and Networks (DSN'06)*. 105–114. https://doi.org/10.1109/DSN.2006.63

[3] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. 2019. CAPER: A Cross-application Permissioned Blockchain. *Proc. VLDB Endow.* 12, 11 (2019), 1385–1398. https://doi.org/10.14778/3342263.3342275

[4] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. 2019. SharPer: Sharding Permissioned Blockchains Over Network Clusters. (2019). https://arxiv.org/abs/1910.00765v1

[5] Jean-Paul Bahsoun, Rachid Guerraoui, and Ali Shoker. 2015. Making BFT Protocols Really Adaptive. In *2015 IEEE International Parallel and Distributed Processing Symposium*. 904–913. https://doi.org/10.1109/IPDPS.2015.21

[6] Johannes Behl, Tobias Distler, and Rüdiger Kapitza. 2017. Hybrids on Steroids: SGX-Based High Performance BFT. In *Proceedings of the Twelfth European Conference on Computer Systems*. ACM, 222–237. https://doi.org/10.1145/3064176.3064213

[7] Iddo Bentov, Pavel Hubáček, Tal Moran, and Asaf Nadler. 2017. Tortoise and Hares Consensus: the Meshcash Framework for Incentive-Compatible, Scalable Cryptocurrencies. (2017). https://eprint.iacr.org/2017/300

[8] P. A. Bernstein and N. Goodman. 1983. Multiversion Concurrency Control - Theory and Algorithms. *ACM TODS* 8, 4 (1983), 465–483.

[9] Matthias Butenuth, Guido v. GÃűsseln, Michael Tiedge, Christian Heipke, Udo Lipeck, and Monika Sester. 2007. Integration of heterogeneous geospatial data in a federated database. *ISPRS Journal of Photogrammetry and Remote Sensing* 62, 5 (2007), 328 – 346. https://doi.org/10.1016/j.isprsjprs.2007.04.003 Theme Issue: Distributed Geoinformatics.

[10] Miguel Castro and Barbara Liskov. 1999. Practical Byzantine Fault Tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*. USENIX, USA, 173–186.

[11] Miguel Castro and Barbara Liskov. 2002. Practical Byzantine Fault Tolerance and Proactive Recovery. *ACM Trans. Comput. Syst.* 20, 4 (2002), 398–461. https://doi.org/10.1145/571637.571640

[12] Byung-Gon Chun, Petros Maniatis, Scott Shenker, and John Kubiatowicz. 2007. Attested Append-only Memory: Making Adversaries Stick to Their Word. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*. ACM, 189–204. https://doi.org/10.1145/1294261.1294280

[13] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*. ACM, 143–154. https://doi.org/10.1145/1807128.1807152

[14] J. C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2012. Spanner: Google's Globally-Distributed Database. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. USENIX Association, 261–264.

[15] Hung Dang, Tien Tuan Anh Dinh, Dumitrel Loghin, Ee-Chien Chang, Qian Lin, and Beng Chin Ooi. 2019. Towards Scaling Blockchain Systems via Sharding. In *Proceedings of the 2019 International Conference on Management of Data*. ACM, 123–140. https://doi.org/10.1145/3299869.3319889

[16] A. Deshpande and J. M. Hellerstein. 2002. Decoupled query optimization for federated database systems. In *Proceedings 18th International Conference on Data Engineering*. 716–727. https://doi.org/10.1109/ICDE.2002.994788

[17] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. 2013. Hekaton: SQL Server's Memory-optimized OLTP Engine. ACM, 1243–1254. https://doi.org/10.1145/2463676.2463710

[18] 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 *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 1085–1100. https://doi.org/10.1145/3035918.3064033

[19] Danny Dolev. 1982. The Byzantine generals strike again. *Journal of Algorithms* 3, 1 (1982), 14–30. https://doi.org/10.1016/0196-6774(82)90004-9

[20] Danny Dolev and Rüdiger Reischuk. 1985. Bounds on Information Exchange for Byzantine Agreement. *J. ACM* 32, 1 (1985), 191–204. https://doi.org/10.1145/2455.214112

[21] Michael J. Fischer and Nancy A. Lynch. 1982. A lower bound for the time to assure interactive consistency. *Inform. Process. Lett.* 14, 4 (1982), 183–186. https://doi.org/10.1016/0020-0190(82)90033-3

[22] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. 1985. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM* 32, 2 (1985), 374–382. https://doi.org/10.1145/3149.214121

[23] Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. 2019. SBFT: A Scalable and Decentralized Trust Infrastructure. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 568–580. https://doi.org/10.1109/DSN.2019.00063

[24] Jim Gray. 1978. Notes on Data Base Operating Systems.

[25] Rachid Guerraoui, Nikola Knezevic, Vivien Quema, and Marko Vukolic. 2010. Stretching BFT. Infoscience EPFL.

[26] Suyash Gupta. 2020. Resilient and Scalable Architecture for Permissioned Blockchain Fabrics. In *Proceedings of the VLDB 2020 PhD Workshop co-located with the 46th International Conference on Very Large Databases (CEUR Workshop Proceedings)*, Vol. 2652. CEUR-WS.org.

[27] Suyash Gupta, Jelle Hellings, Sajjad Rahnama, and Mohammad Sadoghi. 2019. An In-Depth Look of BFT Consensus in Blockchain: Challenges and Opportunities. In *Proceedings of the 20th International Middleware Conference Tutorials, Middleware*. ACM, 6–10. https://doi.org/10.1145/3366625.3369437

[28] Suyash Gupta, Jelle Hellings, Sajjad Rahnama, and Mohammad Sadoghi. 2020. Building High Throughput Permissioned Blockchain Fabrics: Challenges and Opportunities. *Proc. VLDB Endow.* 13, 12 (2020), 3441–3444. https://doi.org/10.14778/3415478.3415565

[29] Suyash Gupta, Jelle Hellings, Sajjad Rahnama, and Mohammad Sadoghi. 2021. Proof-of-Execution: Reaching Consensus through Fault-Tolerant Speculation. In *Proceedings of the 24th International Conference on Extending Database Technology, EDBT*. OpenProceedings.org, 301–312. https://doi.org/10.5441/002/edbt.2021.27

[30] Suyash Gupta, Jelle Hellings, and Mohammad Sadoghi. 2019. Brief Announcement: Revisiting Consensus Protocols through Wait-Free Parallelization. In *33rd International Symposium on Distributed Computing, DISC (LIPIcs)*, Vol. 146. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 44:1–44:3. https://doi.org/10.4230/LIPIcs.DISC.2019.44

[31] Suyash Gupta, Jelle Hellings, and Mohammad Sadoghi. 2021. *Fault-Tolerant Distributed Transactions on Blockchain*. Morgan & Claypool Publishers. https://doi.org/10.2200/S01068ED1V01Y202012DTM065

[32] Suyash Gupta, Jelle Hellings, and Mohammad Sadoghi. 2021. RCC: Resilient Concurrent Consensus for High-Throughput Secure Transaction Processing. In *37th IEEE International Conference on Data Engineering, ICDE*. 1392–1403. https://doi.org/10.1109/ICDE51399.2021.00124

[33] Suyash Gupta, Sajjad Rahnama, Jelle Hellings, and Mohammad Sadoghi. 2020. ResilientDB: Global Scale Resilient Blockchain Fabric. *Proc. VLDB Endow.* 13, 6 (2020), 868–883. https://doi.org/10.14778/3380750.3380757

[34] Suyash Gupta, Sajjad Rahnama, Shubham Pandey, Natacha Crooks, and Mohammad Sadoghi. 2022. Dissecting BFT Consensus: In Trusted Components we Trust! *CoRR* abs/2202.01354 (2022). arXiv:2202.01354

[35] Suyash Gupta, Sajjad Rahnama, and Mohammad Sadoghi. 2020. Permissioned Blockchain Through the Looking Glass: Architectural and Implementation Lessons Learned. In *40th IEEE International Conference on Distributed Computing Systems, ICDCS*. 754–764. https://doi.org/10.1109/ICDCS47774.2020.00012

[36] Suyash Gupta and Mohammad Sadoghi. 2018. EasyCommit: A Non-blocking Two-phase Commit Protocol. In *Proceedings of the 21st International Conference on Extending Database Technology, EDBT*. OpenProceedings.org, 157–168. https://doi.org/10.5441/002/edbt.2018.15

[37] Suyash Gupta and Mohammad Sadoghi. 2019. Blockchain Transaction Processing. In *Encyclopedia of Big Data Technologies*. Springer, 1–11. https://doi.org/10.1007/978-3-319-63962-8_333-1

[38] Suyash Gupta and Mohammad Sadoghi. 2020. Efficient and non-blocking agreement protocols. *Distributed Parallel Databases* 38, 2 (2020), 287–333. https://doi.org/10.1007/s10619-019-07267-w

[39] R. Harding, D. Van Aken, A. Pavlo, and M. Stonebraker. 2017. An Evaluation of Distributed Concurrency Control. *Proc. VLDB Endow.* 10, 5 (2017), 553–564. https://doi.org/10.14778/3055540.3055548

[40] Thomas Haynes and David Noveck. 2015. RFC 7530: Network File System (NFS) Version 4 Protocol. (2015). https://tools.ietf.org/html/rfc7530

[41] Jelle Hellings, Daniel P. Hughes, Joshua Primero, and Mohammad Sadoghi. 2020. Cerberus: Minimalistic Multi-shard Byzantine-resilient Transaction Processing. (2020). https://arxiv.org/abs/2008.04450

[42] Jelle Hellings and Mohammad Sadoghi. 2019. The fault-tolerant cluster-sending problem. (2019). https://arxiv.org/abs/1908.01455

[43] Jelle Hellings and Mohammad Sadoghi. 2021. ByShard: Sharding in a Byzantine Environment. *Proc. VLDB Endow.* 14, 11 (2021), 2230–2243.

[44] Jonathan Katz and Yehuda Lindell. 2014. *Introduction to Modern Cryptography* (2nd ed.).

[45] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. 2018. OmniLedger: A Secure, Scale-Out, Decentralized Ledger via Sharding. In *2018 IEEE Symposium on Security and Privacy (SP)*. 583–598. https://doi.org/10.1109/SP.2018.000-5

[46] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. 2007. Zyzzyva: Speculative Byzantine Fault Tolerance. *SIGOPS Oper. Syst. Rev.* 41, 6 (2007), 45–58. https://doi.org/10.1145/1323293.1294267

[47] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. 2010. Zyzzyva: Speculative Byzantine Fault Tolerance. *ACM Trans. Comput. Syst.* 27, 4, Article 7 (2010), 39 pages. https://doi.org/10.1145/1658357.1658358

[48] Leslie Lamport. 1998. The Part-time Parliament. (1998).

[49] Chenxing Li, Peilun Li, Dong Zhou, Wei Xu, Fan Long, and Andrew Yao. 2018. Scaling Nakamoto Consensus to Thousands of Transactions per Second. (2018). https://arxiv.org/abs/1805.03870

[50] Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. 2016. A Secure Sharding Protocol For Open Blockchains.

In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 17–30. https://doi.org/10.1145/2976749.2978389

[51] Dahlia Malkhi and Michael Reiter. 1998. Byzantine quorum systems. *Distributed Computing* 11, 4 (1998), 203–213. https://doi.org/10.1007/s004460050050

[52] Dahlia Malkhi and Michael Reiter. 1998. Secure and scalable replication in Phalanx. In *Proceedings Seventeenth IEEE Symposium on Reliable Distributed Systems*. IEEE, 51–58. https://doi.org/10.1109/RELDIS.1998.740474

[53] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. 2016. The Honey Badger of BFT Protocols. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. ACM, 31âĂŞ42. https://doi.org/10.1145/2976749.2978399

[54] The Council of Economic Advisers. 2018. *The Cost of Malicious Cyber Activity to the U.S. Economy*. Technical Report. Executive Office of the President of the United States. https://www.whitehouse.gov/wp-content/uploads/2018/03/The-Cost-of-Malicious-Cyber-Activity-to-the-U.S.-Economy.pdf

[55] Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *ATC*.

[56] M. Tamer Özsu and Patrick Valduriez. 2020. *Principles of Distributed Database Systems*. Springer. https://doi.org/10.1007/978-3-030-26253-2

[57] Thamir Qadah, Suyash Gupta, and Mohammad Sadoghi. 2020. Q-Store: Distributed, Multi-partition Transactions via Queue-oriented Execution and Communication. In *Proceedings of the 23rd International Conference on Extending Database Technology, EDBT*. OpenProceedings.org, 73–84. https://doi.org/10.5441/002/edbt.2020.08

[58] Sajjad Rahnama, Suyash Gupta, Thamir Qadah, Jelle Hellings, and Mohammad Sadoghi. 2020. Scalable, Resilient and Configurable Permissioned Blockchain Fabric. *Proc. VLDB Endow.* 13, 12 (2020), 2893–2896. https://doi.org/doi.org/10.14778/3415478.3415502

[59] Sajjad Rahnama, Suyash Gupta, Rohan Sogani, Dhruv Krishnan, and Mohammad Sadoghi. 2021. RingBFT: Resilient Consensus over Sharded Ring Topology. *CoRR* abs/2107.13047 (2021). arXiv:2107.13047

[60] Mohammad Sadoghi and Spyros Blanas. 2019. *Transaction Processing on Modern Hardware*. Morgan & Claypool Publishers. https://doi.org/10.2200/S00896ED1V01Y201901DTM058

[61] Amit P. Sheth and James A. Larson. 1990. Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. *ACM Comput. Surv.* 22, 3 (Sept. 1990), 183âĂŞ236. https://doi.org/10.1145/96602.96604

[62] Dale Skeen. 1982. *A Quorum-Based Commit Protocol*. Technical Report. Cornell University.

[63] Yonatan Sompolinsky, Yoad Lewenberg, and Aviv Zohar. 2016. SPECTRE: A Fast and Scalable Cryptocurrency Protocol. (2016). https://eprint.iacr.org/2016/1159.

[64] Gadi Taubenfeld and Shlomo Moran. 1996. Possibility and impossibility results in a shared memory environment. *Acta Informatica* 33, 1 (1996), 1–20. https://doi.org/10.1007/s002360050034

[65] Gerard Tel. 2001. *Introduction to Distributed Algorithms* (2nd ed.). Cambridge University Press.

[66] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. 2012. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD)*. ACM, 1–12. https://doi.org/10.1145/2213836.2213838