ByShard: Sharding in a Byzantine Environment

Jelle Hellings
Exploratory Systems Lab
Department of Computer Science
University of California, Davis
jhellings@ucdavis.edu

Mohammad Sadoghi
Exploratory Systems Lab
Department of Computer Science
University of California, Davis
msadoghi@ucdavis.edu

ABSTRACT

The emergence of blockchains has fueled the development of resilient systems that can deal with *Byzantine failures* due to crashes, bugs, or even malicious behavior. Recently, we have also seen the exploration of *sharding* in these resilient systems, this to provide the scalability required by very large data-based applications. Unfortunately, current sharded resilient systems all use system-specific specialized approaches toward sharding that do not provide the flexibility of traditional sharded data management systems.

To improve on this situation, we fundamentally look at the design of sharded resilient systems. We do so by introducing ByShard, a unifying framework for the study of sharded resilient systems. Within this framework, we show how *two-phase commit* and *two-phase locking*—two techniques central to providing *atomicity* and *isolation* in traditional sharded databases—can be implemented efficiently in a Byzantine environment, this with a minimal usage of costly Byzantine resilient primitives. Based on these techniques, we propose *eighteen* multi-shard transaction processing protocols. Finally, we practically evaluate these protocols and show that each protocol supports high transaction throughput and provides scalability while each striking its own trade-off between *throughput*, *isolation level*, *latency*, and *abort rate*. As such, our work provides a strong foundation for the development of ACID-compliant general-purpose and flexible sharded resilient data management systems.

1 INTRODUCTION

The emergence of blockchains is fueling interest in new *resilient systems* that provide data and transaction processing in the presence of *Byzantine behavior*, e.g., faulty behavior originating from software, hardware, or network failures, or from coordinated malicious attacks [2, 4, 20, 26, 28, 43]. These blockchain-inspired systems are attractive, as they can provide resilience among many independent participants [8, 10, 12, 13, 22, 23, 25, 26, 33, 36, 37, 41, 42, 44, 48, 49, 56]. As such, blockchain-inspired systems can *prevent service disruption* due to failures that compromise part of the system, and can *improve data quality* of data that is managed by many independent parties, potentially reducing the huge costs associated with both [10, 19, 35, 45, 46, 50, 53].

Unfortunately, typical blockchain-inspired systems utilize a fully-replicated design in which every participating replica holds all data and processes all transactions, which is at odds with the scalability requirements of modern very large data-based applications [48, 51]. Consequently, recent blockchain-inspired data processing systems such as AHL [15], Caper [2], Cerberus [30], Chainspace [1], and SharPer [3] propose to provide *scalability* by introducing *sharding*: instead of operating a single fully-replicated system, one partitions the data (e.g., based on location) among several independently-run blockchain-based resilient clusters that each operate as a single shard. We have sketched this design in Figure 1.1.

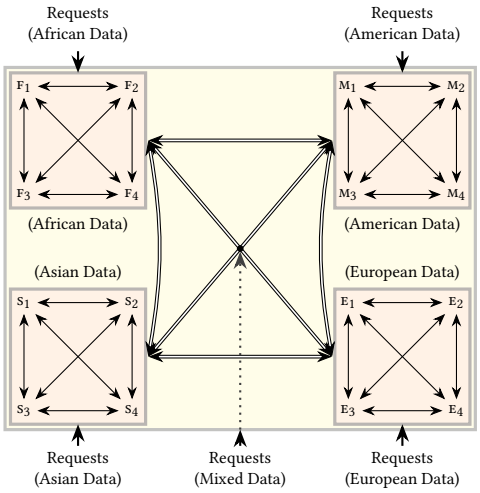

Figure 1.1: A *geo-scale aware sharded* design in which four resilient clusters hold only a part of the data. Local decisions within a cluster are made via *consensus* (⟷), whereas multi-shard coordination to process multi-shard transactions requires *cluster-sending* (⟺).

In such a sharded design, several resilient clusters together maintain all data, while each cluster only holds part of the data. Consequently, sharded designs provide *storage scalability* as adding shards increases overall storage capacity. Furthermore, sharded designs promise *processing scalability* as transactions on data held by different shards can be processed in *parallel*. To deliver on the promises of sharding, one needs an efficient way to process *multi-shard transactions* that affect data on multiple shards, however [47].

Unfortunately, existing sharded resilient systems use system-specific solutions to provide multi-shard transaction processing: they either are mainly optimized for single-shard transactions [15], are mainly optimized for transactions that do not content for the same resources [2, 3], or depend on the specifics of a UTXO-based data model to deal with contention [1, 30]. This is in contrast with traditional distributed databases, which can provide application-agnostic ACID-compliant data and transaction processing that can be tuned to a wide range of application-specific requirements. E.g., by offering flexible multi-shard transaction capabilities using two-phase commit [24, 47, 52] and two-phase locking [47].

This raises the question whether such flexible multi-shard transaction capabilities can be provided in a Byzantine environment. In this paper, we positively answer this question in three steps. First,

we take a structured look at providing resilience in a Byzantine environment and how this affects sharded transaction processing. Next, we introduce the ByShard framework, a formalization of sharded resilient systems, and show how the design principles of traditional distributed databases can be expressed within this framework. Finally, we use the ByShard framework to evaluate the resulting design space for multi-shard transaction processing in a Byzantine environment.

To process multi-shard transactions, ByShard introduces the *orchestrate-execute model* (OEM). This model can incorporate all commit, locking, and execution operations required for processing a multi-shard transaction in at-most two consensus steps per involved shard. The first component of OEM is *orchestration*: the replication of transactions among all involved shards while also *reaching an atomic decision* on whether the transaction can be committed or not. To provide orchestration, we show how to adapt *two-phase commit style* orchestration to a Byzantine environment at a minimal cost (in terms of consensus steps at the involved shards). In specific:

(1) We provide *linear orchestration* that minimizes the overall number of consensus and cluster-sending steps necessary to reach an agreement decision, this at the cost of latency.
(2) We provide *centralized orchestration and distributed orchestration* that both minimize the latency necessary to reach an agreement decision by reaching such decisions in at-most three or four consecutive consensus steps, respectively, this at the cost of additional consensus and cluster-sending steps.
(3) To enable centralized and distributed orchestration, we introduce Byzantine primitives to process *all* commit and abort votes using only a single consensus step per involved shard.

The second component of OEM is *execution* of transactions. To provide execution capabilities that maintain *data consistency* among shards, we show how to adapt standard *two-phase locking style* execution to a Byzantine environment at a minimal cost (in terms of consensus steps at the involved shards). In specific:

(4) We introduce Byzantine primitives to provide *blocking locks* that can be processed without any additional consensus steps for the involved shards. Furthermore, we show how these primitives also support *non-blocking locks*.
(5) Based on these primitives, we show how *read uncommitted*, *read committed*, and *serializable* execution of transactions can be provided.
(6) As a baseline, we also include isolation-free execution.

These orchestration and execution methods result in *eighteen* practical protocols for processing multi-shard transaction. To further showcase the flexibility of ByShard, we show that both AHL [15] and a generalization of Chainspace [1] can be expressed within OEM. Finally, we combine the above techniques with a data and transaction model representative for a Byzantine sharded environment and evaluate the behavior of the resulting designs:

(7) Our evaluation shows that all eighteen ByShard protocols can effectively deal with multi-shard transaction workloads and have excellent *scalability*: increasing the number of shards will always decrease the work done per shard.
(8) Furthermore, all eighteen ByShard protocols have excellent transaction *throughput* when contention is low. When contention is high, each of the protocols makes their own trade-off between *isolation level*, *latency*, and *abort rate* while maximizing throughput.

We believe our work provides a solid foundation for the development of flexible *general-purpose* scalable Byzantine data management systems.

2 BACKGROUND ON RESILIENT SYSTEMS

Before we look at the design of sharded resilient systems, we take a look at the operations of traditional (non-sharded) resilient systems that can deal with *Byzantine* behavior (e.g., replicas that crash, behave faulty, or act malicious). Typical resilient systems process a transaction τ requested by client c by performing five steps:

(1) first, τ needs to be *received* by the system;
(2) second, τ must be reliably *replicated* among all replicas in the system;
(3) third, the replicas need to agree on an *execution order* for τ;
(4) next, the replicas each need to *execute* τ and *update* their current state accordingly; and
(5) finally, client c needs to be *informed* about the result.

At the core of resilient systems are *consensus protocols* [9, 11, 26, 39, 40] that coordinate the operations of individual replicas in the system by *replicating* transactions among all non-faulty replicas in a fault-tolerant manner, e.g., a Byzantine fault-tolerant system driven by Pbft [11] or a crash fault-tolerant system driven by Paxos [39]:

Definition 2.1. A *consensus protocol* coordinates decision making among the replicas of a resilient cluster (of replicas) $\mathcal{S}$ by providing a reliable ordered replication of *decisions*. To do so, consensus protocols provide the following guarantees:

(1) if non-faulty replica $\mathrm{R} \in \mathcal{S}$ makes an i-th decision, then all non-faulty replicas $\mathrm{R}' \in \mathcal{S}$ will make an i-th decision (whenever communication becomes reliable);
(2) if non-faulty replicas $\mathrm{R}_1, \mathrm{R}_2 \in \mathcal{S}$ make i-th decisions D_1 and D_2, respectively, then $D_1 = D_2$ (they make the same i-th decisions); and
(3) whenever a non-faulty replica learns that a decision D needs to be made, then it can force consensus on D.

Resilient systems operate in rounds, and in each round consensus is used to decide on and replicate a single transaction (or a set of transactions if batching is used [26]). The round in which a transaction is replicated also determines a linearizable *execution order*. Hence, replication of a transaction and agreeing on an execution order (steps 2 and 3 above) are *a single consensus step*. In practical deployments of resilient systems, reaching consensus on a decision is costly and takes a rather long time. We illustrate this next.

Remark 2.2. Consider a deployment of the Pbft consensus protocol [11, 26, 28]. To maximize resilience and to deal with disruptions at any location, individual replicas need to be spread out over a wide-area network. E.g., spread-out in North America. Due to the spread-out nature of the system, the message delay between replicas is high, and a message delay of $\delta = 10$ ms is at the low end [15, 27].

Under normal conditions, Pbft operates via a *primary-backup* design in which a designated replica (the primary) is responsible for proposing decisions to all other replicas (the backups). The primary does so via a PrePrepare message. Next, all replicas exchange their

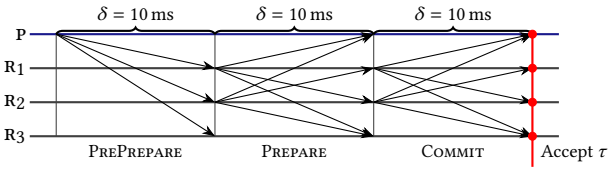

Figure 2.1: A schematic representation of the normal-case of Pbft: the primary P proposes transaction τ to all replicas via a PrePrepare message. Next, replicas commit to τ via a two-phase all-to-all message exchange. In this example, replica R3 is faulty and does not participate.

local state to determine whether the primary properly proposed a decision. To do so, all replicas participate in two phases of all-to-all communication (via Prepare and Commit messages). Hence, if the message delay is δ, then it will take at least 3δ (first the PrePrepare phase, then the Prepare phase, and, finally, the Commit phase) before a proposed decision is accepted by all replicas. E.g., with $\delta = 10$ ms, it will take at least $3\delta = 30$ ms for Pbft to decide on a transaction after the primary received that transaction. In Figure 2.1, we have illustrated this basic working of Pbft.

In a naive implementation of Pbft, the message delay ultimately limits the transaction throughput: if the $(\rho + 1)$-th consensus decision will be made *sequentially after* the ρ-th decision, then the resulting throughput will be at-most $1/(3\delta) \approx 33$ txn/s in the sketched environment. To increase performance, Pbft implementations can use *out-of-order processing* in which the replicas can work on several consensus rounds at the same time [11, 15, 26, 28]. E.g., if individual replicas have sufficient network bandwidth and memory buffers available, then a fine-tuned out-of-order Pbft can easily reach 1000 txn/s. Furthermore, batching can be used such that each consensus decision itself represents many transactions, resulting in systems that can reach even higher throughputs. The high cost of consensus is not specific to Pbft and is shared by all other popular consensus protocols. E.g., in HotStuff [59], each consensus decision will take at least $7\delta = 70$ ms in the sketched environment.

To assure that all non-faulty replicas have *the same state*, transactions are executed in the linearizable order determined via consensus and must be *deterministic* in the sense that execution must always produce exactly the same results given identical inputs:

Example 2.3. Consider a banking system in which each transaction changes the balance of one or more accounts. The *current state* is the balance of each account and can be obtained from the initial state by executing each transaction in-order. Consider the first four transactions

τ_1 = "add \$500 to *Ana*";
τ_2 = "add \$200 to *Bo* and \$300 to *Elisa*";
τ_3 = "move \$30 from *Ana* to *Elisa*";
τ_4 = "remove \$70 from *Elisa*"

(in which the balance of each account is referred to by the account holder). After execution of these transactions, the current state evolves as illustrated in Figure 2.2.

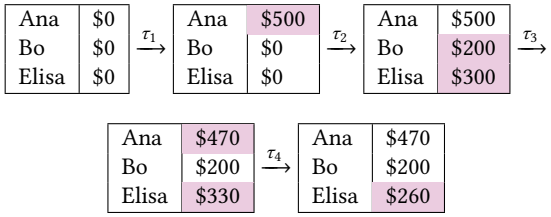

Figure 2.2: Evolution of the *current state* while executing the transactions of Example 2.3.

As all replicas maintain exactly the same (fully-replicated) state and, using consensus, replicate exactly the same transactions and determine exactly the same execution order, each replica can *execute* each transaction and *update* their current state fully independent (without any further need to exchange information). Hence, in a resilient system, transaction processing can be reduced to the *single* problem of ordered transaction replication, which is solved by off-the-shelf consensus protocols [11, 39, 59] (independent of the data and transaction model supported by the system).

Here, we assume that transactions are always replicated and executed as a whole. To deal with *non-applicable transactions*, e.g., that violate constraints, we can include *abort* as a legitimate execution outcome (that does not affect the current state). This assumption is essential to reliably deal with Byzantine behavior: all decisions—including the decision that a transaction is not-applicable—need to be made by *all non-faulty replicas* (via consensus), this to ensure that Byzantine replicas cannot force such a decision or interfere with reliably making such a decision.

Example 2.4. Consider the banking system of Example 2.3. After execution of τ_1, τ_2, τ_3, and τ_4, Ana has a balance of \$470. Now consider transaction τ_5 = "move \$500 from *Ana* to *Bo*". If the system prevents negative account balances, then τ_5 cannot be successfully executed after τ_4. Hence, if τ_5 is replicated and scheduled for execution right after τ_4, then the transaction must be *aborted* at all replicas, and the client needs to be informed of this abort.

3 TOWARDS SHARDED RESILIENT SYSTEMS

In the previous section, we detailed the operations of traditional non-sharded resilient systems: we outlined *five* steps resilient systems perform to process transactions in a *Byzantine environment* and showed that all necessary coordination and communication between replicas in such a system is restricted to a single ordered replication step, which is handled via consensus.

The step from a non-sharded to a sharded resilient system complicates the processing of transactions significantly. To illustrate this, we revisit the five steps for processing a transaction in a resilient system. Consider a multi-shard transaction τ processed by a resilient system and assume we know which shards are involved in processing τ. First, the transaction τ needs to be replicated to all replicas of all shards involved in executing τ. After this, the replicas need to agree an *execution order* for τ. In fully-replicated systems, both steps are solved at once using system-wide consensus, as the replication order determines a linearizable execution order. In a sharded system, per-shard replication of τ only yields

a local linearizable replication order within that shard, however. As distinct shards can replicate transactions locally in different orders, the local replication order does not necessary determine a conflict-free execution order for τ across shards (e.g., serializable execution [5, 6, 29]). Hence, determining an execution order of τ across shards—necessary to maintain data consistency across shards—requires further coordination between the involved shards.

Besides determining the execution order, also *execution* and *updating* the state of replicas poses a challenge in a sharded environment. Within traditional systems, individual replicas can independently execute transactions and update their state accordingly as each replica holds a full copy of all data. This no longer holds for multi-shard transaction: each replica only holds a copy of the data in its shard. Hence, for the execution of τ, replicas in the involved shards need to exchange any necessary state. This exchange is complicated by the presence of Byzantine replicas in each of the involved shards and, hence, requires additional coordination to assure that all necessary state is reliably exchanged.

Next, we will step-wise address these challenges towards multi-shard transaction processing in sharded resilient systems. First, we introduce the ByShard framework, a formalization of sharded resilient systems. Next, we present the *orchestrate-execute model* (OEM) used by ByShard to process multi-shard transaction. Then, in Section 4, we propose orchestration methods inspired by two-phase commit. Next, in Section 5, we propose execution methods inspired by two-phase locking. Finally, in Section 6, we evaluate the performance of transaction processing via OEM in ByShard.

3.1 ByShard: a resilient sharding framework

Let $\mathcal{R}$ be a set of replicas. We model a *sharded system* as a partitioning of $\mathcal{R}$ into a set of $\mathbf{z}$ shards $\mathfrak{S} = \{\mathcal{S}_1, \ldots, \mathcal{S}_\mathbf{z}\}$. Let $\mathcal{S} \in \mathfrak{S}$ be a shard. We write $\mathbf{n}_\mathcal{S} = |\mathcal{S}|$ to denote the number of replicas in $\mathcal{S}$ and $\mathbf{f}_\mathcal{S} = |\mathcal{S}|$ to denote the Byzantine faulty replicas in $\mathcal{S}$. We assume $\mathbf{n}_\mathcal{S} > 3\mathbf{f}_\mathcal{S}$, a minimal requirement to deal with Byzantine behavior within a single shard in practical settings [17, 18]. Let τ be a transaction. We write $\text{shards}(\tau) \subseteq \mathfrak{S}$ to denote the shards that are affected by τ (the shards that contain data that τ reads or writes). We say that τ is a *single-shard transaction* if $|\text{shards}(\tau)| = 1$ and a *multi-shard transaction* otherwise.

Example 3.1. Consider a banking system similar to that of Example 2.3. This time, however, the system is *sharded* into twenty-six shards $\mathfrak{S} = \{\mathcal{S}_a, \ldots, \mathcal{S}_z\}$, one for each letter of the alphabet, such that the shard $\mathcal{S}_\alpha$, $\alpha \in \{a, \ldots, z\}$, holds accounts whose name starts with α. Now reconsider the transactions of Example 2.3. We have $\text{shards}(\tau_1) = \{\mathcal{S}_a\}$, $\text{shards}(\tau_2) = \{\mathcal{S}_b, \mathcal{S}_e\}$, $\text{shards}(\tau_3) = \{\mathcal{S}_a, \mathcal{S}_e\}$, and $\text{shards}(\tau_4) = \{\mathcal{S}_e\}$. Hence, transactions τ_1 and τ_4 are single-shard transactions, whereas τ_2 and τ_3 are multi-shard transactions.

Within ByShard, we can employ any *consensus protocol* [9, 11, 39, 40] to make decisions within a shard, which allows us to operate shards as if they are a single-replica shard. We assume that consensus protocols in ByShard only make *valid* decisions: each decision made by a shard $\mathcal{S}$ will reflect a single processing step at that shard of some transaction. We also need a Byzantine resilient primitive that enables coordination *between* shards. For this role, we can choose any *cluster-sending protocol* [27, 31] that provides reliable communication between shards:

Definition 3.2. A *cluster-sending protocol* provides reliable communication between resilient clusters $\mathcal{S}_1$ and $\mathcal{S}_2$. To enable $\mathcal{S}_1$ to send a value v to $\mathcal{S}_2$, cluster sending protocols provide the following guarantees:

(1) $\mathcal{S}_1$ is able to send v to $\mathcal{S}_2$ only if there is *agreement* on sending v among the non-faulty replicas in $\mathcal{S}_1$;
(2) all non-faulty replicas in $\mathcal{S}_2$ will *receive* the value v; and
(3) all non-faulty replicas in $\mathcal{S}_1$ obtain *confirmation* of receipt.

In ByShard, cluster-sending steps always follow consensus decision. Hence, agreement on any cluster-sending step will be reached without further consensus overhead.

3.2 The orchestrate-execute model

Consider a multi-shard transaction τ. To process this transaction, we will require *commit steps* to replicate the transaction among all replicas in all involved shards and to reach an atomic decision on whether to commit or abort τ. Furthermore, we will require *locking steps* to provide isolated execution, guaranteeing a consistent execution order among all shards, and *execution steps* that update the state of individual replicas.

At the same time, we also want to *minimize* the number of consensus decisions at each involved shard to implement these commit, locking, and execution steps. To do so, we propose the *orchestrate-execute model* (OEM) that is able to incorporate the necessary commit, locking, and execution steps required for processing a multi-shard transaction in at-most two consensus steps per involved shard. In OEM, processing of a multi-shard transaction τ is modeled via individual *shard-steps* that are performed independently by each shard in $\text{shards}(\tau)$ via consensus. Each shard-step of $\mathcal{S} \in \text{shards}(\tau)$ can inspect local data at $\mathcal{S}$, modify local data at $\mathcal{S}$, and forward execution to other shards via cluster-sending:

Example 3.3. Consider the sharded banking example of Example 3.1 and consider the transaction

τ = "if *Ana* has \$500 and *Bo* has \$200, then
move \$400 from *Ana* to *Elisa*; move \$100 from *Bo* to *Elisa*",

requested by client c. We have $\text{shards}(\tau) = \{\mathcal{S}_a, \mathcal{S}_b, \mathcal{S}_e\}$. Next, we rewrite τ to a processing plan with a minimal number of shard-steps (on success). This plan has four shard-steps, namely:

σ_1 = "if *Ana* has \$500, then remove \$400 from *Ana*; $\Longrightarrow_{\mathcal{S}_b}(\sigma_2)$
else send `failure` to c"
σ_2 = "if *Bo* has \$200, then remove \$100 from *Bo*; $\Longrightarrow_{\mathcal{S}_e}(\sigma_3)$
else $\Longrightarrow_{\mathcal{S}_a}(\sigma_4)$"
σ_3 = "add \$500 to *Elisa* and send `success` to c"
σ_4 = "add \$400 to *Ana* and send `failure` to c"

In which $\Longrightarrow_\mathcal{S}(\sigma)$ represents a cluster-sending step that forwards execution to shard $\mathcal{S}$, which is then instructed to execute shard-step σ. For simplicity, we omitted any locking from this processing plan. Hence, this plan results in a non-isolated execution that can violate *consistency constraints* on the data. Notice that the shards affected by processing τ depend on the *current state*: depending on the current state of $\mathcal{S}_a$ and $\mathcal{S}_b$, either only $\mathcal{S}_a$ is affected, or $\mathcal{S}_a$ and $\mathcal{S}_b$ are affected, or $\mathcal{S}_a$, $\mathcal{S}_b$, and $\mathcal{S}_e$ are affected.

OEM will overlap the operations necessary for providing *atomicity*, *isolation*, and *consistency* [5, 6, 29] to minimize the number of consensus steps. For this overlapped design, OEM utilizes only *three types* of shard-steps per shard:

Vote-step A *vote-step* Vote($\mathcal{S}$) for $\mathcal{S}$ verifies the constraints to determine whether $\mathcal{S}$ votes for either *commit* or *abort*. Furthermore, the vote-step can make local changes, e.g., modify local data or acquire locks. To simplify presentation, we assume that a vote-step yielding an *abort* vote does not have any side-effects.

Commit-step A *commit-step* Commit($\mathcal{S}$) for $\mathcal{S}$ performs necessary operations to finalize τ when τ is committed. E.g., modify data and release locks obtained during a preceding vote-step.

Abort-step An *abort-step* Abort($\mathcal{S}$) for $\mathcal{S}$ performs necessary operations to roll back τ when τ is aborted. E.g., roll back local changes of a preceding vote-step or release locks obtained during a preceding vote-step.

Example 3.4. Consider the processing plan for τ of Example 3.3. The shard-steps σ_1 and σ_2 are *vote-steps* that decide whether τ can commit by checking the balance of Ana and Bo. The shard-step σ_3 is a *commit-step* that finalizes execution. Finally, shard-step σ_4 is an *abort-step* that cancels out the modifications made by vote-step σ_1.

In the following two sections, we will discuss how to process multi-shard transactions using these three shard-steps with minimal cost (in terms of consensus and cluster-sending steps).

4 PROVIDING ORCHESTRATION IN OEM

Let τ be a multi-shard transaction. The first part of processing τ is to orchestrate the replication of τ to the involved shards in shards(τ), assure that all these shards reach an *atomic* decision on whether to commit (and execute τ) or to abort (and cancel execution of τ), and trigger the corresponding commit-steps or abort-steps. As such, orchestration mimics the role of *commit protocols* in traditional sharded data management systems [24, 47, 52]. Next, we introduce the three orchestration methods of ByShard.

4.1 Linear orchestration

First, we propose an orchestration method based on the traditional *linear two-phase commit protocol* (Linear-2PC) [24, 47].

Let $\mathcal{S}_1, \ldots, \mathcal{S}_n$ be an ordering of all shards $\mathcal{S}_1, \ldots, \mathcal{S}_n \in$ shards(τ) with vote-steps. The transaction is orchestrated towards a decision by starting execution of Vote($\mathcal{S}_1$). If execution of Vote($\mathcal{S}_i$), $1 \leq i < n$, results in a *commit* vote, then $\mathcal{S}_i$ forwards execution of τ to $\mathcal{S}_{i+1}$, after which $\mathcal{S}_{i+1}$ will start execution of Vote($\mathcal{S}_{i+1}$). If execution of Vote($\mathcal{S}_n$) results in a *commit* vote, then τ will be committed. To do so, $\mathcal{S}_n$ forwards execution of τ to all shards $\mathcal{S} \in$ shards(τ) with a commit-step Commit($\mathcal{S}$), after which each such shard will execute Commit($\mathcal{S}$) *in parallel*. Finally, if execution of Vote($\mathcal{S}_i$), $1 \leq i \leq n$, results in an *abort* vote, then τ will immediately be aborted without further vote-steps (*fast-abort*). To do so, $\mathcal{S}_i$ forwards execution of τ to all shards $\mathcal{S} \in \{\mathcal{S}_1, \ldots, \mathcal{S}_{i-1}\}$ with an abort-step Abort($\mathcal{S}$), after which each such shard will execute Abort($\mathcal{S}$) *in parallel*. We illustrated linear orchestration in Figure 4.1, *left*.

Theorem 4.1. *Let τ be a transaction with n_v vote-steps, n_c commit-steps, and n_a abort-step. Using* linear orchestration*, τ can be committed (aborted) in $n_v + 1$ (in at-most $n_v + 1$) consecutive consensus steps using $n_v + n_c$ (using at-most $n_v + n_a$) consensus-steps and using $n_v + n_c - 1$ (using at-most $n_v + n_a - 1$) cluster-sending steps.*

Proof. Assume that τ is committed. In this case, the n_v vote-steps are performed in sequence, after which all n_c commit-steps are performed in parallel. Hence, we use $n_v + n_c$ consensus steps, of which n_v+1 need to be consecutive. To forward execution, n_v+n_c-1 cluster-sending steps are performed. The case in which τ is aborted is analogous. □

The main strengths of linear orchestration are its simplicity, the flexibility in the order in which vote-steps are processed, and its ability to *abort-fast*. As linear orchestration will only perform abort-steps at previously-voted shards, one can minimize the number of abort-steps by first processing vote-steps of shards with *only vote-steps*, and only after that the shards with both vote- and abort-steps. Furthermore, if heuristics are available, then linear orchestration can prioritize vote-steps with high likelihood of constraint failure in an attempt to quickly arrive at abort. Finally, we can eliminate the commit-step or abort-step for $\mathcal{S}_n$, as these steps can be processed at the same time as the vote-step of $\mathcal{S}_n$.

4.2 Centralized orchestration

As we have seen, linear orchestration is simple and, due to its ability to abort-fast, can minimize the number of shard-steps performed to process τ. This approach comes at the cost of *consecutively* visiting each shard that has applicable vote-steps. Hence, linear orchestration takes at worst |shards(τ)| + 1 consecutive consensus steps for the execution of a transaction τ. As an alternative, we can consider *parallelized orchestration* by processing all vote-steps at the same time (in parallel). Next, we propose such orchestration based on the traditional *centralized two-phase commit protocol* (Centralized-2PC) [47]. First, we present the core idea of such centralized orchestration. Then, we detail on how to efficiently collect and process the votes resulting from all vote-steps in a Byzantine environment.

Let $\mathcal{S}_r, \mathcal{S}_1, \ldots, \mathcal{S}_n$ be an ordering of all shards $\mathcal{S}_r, \mathcal{S}_1, \ldots, \mathcal{S}_n \in$ shards(τ) with vote-steps. We refer to $\mathcal{S}_r$ as the *root* for τ, which will coordinate the orchestration of τ. To assure that the role of the *root* is distributed over all shards, centralized orchestration does not depend on any particular choice of $\mathcal{S}_r$. Hence, any $\mathcal{S}_r \in$ shards(τ) will do. The root of τ starts by executing Vote($\mathcal{S}_r$). If Vote($\mathcal{S}_r$) results in a *commit* vote, then $\mathcal{S}_r$ forwards execution of τ to *all* shards $\mathcal{S}_1, \ldots, \mathcal{S}_n$, after which each shard $\mathcal{S}_i$, $1 \leq i \leq n$, executes Vote($\mathcal{S}$) *in parallel*. After forwarding, $\mathcal{S}_r$ can proceed with shard-steps of other transactions. Let $\mathcal{S}_i$, $1 \leq i \leq n$, be a shard. If Vote($\mathcal{S}_i$) results in a *commit* vote, then $\mathcal{S}_i$ sends a *commit vote* via cluster-sending to $\mathcal{S}_r$. Otherwise, if Vote($\mathcal{S}_i$) results in an *abort* vote, then $\mathcal{S}_i$ sends an *abort vote* via cluster-sending to $\mathcal{S}_r$. After sending a vote to $\mathcal{S}_r$, $\mathcal{S}_i$ can proceed with shard-steps of other transactions.

If $\mathcal{S}_r$ receives *commit* votes from each shard $\mathcal{S}_1, \ldots, \mathcal{S}_n$, then τ will be committed. To do so, $\mathcal{S}_r$ forwards a *global commit vote* via cluster-sending to all shards $\mathcal{S} \in$ shards(τ) with a commit-step Commit($\mathcal{S}$), after which each such shard executes Commit($\mathcal{S}$) *in parallel*. If $\mathcal{S}_r$ receives a single *abort* vote, then τ will be aborted.

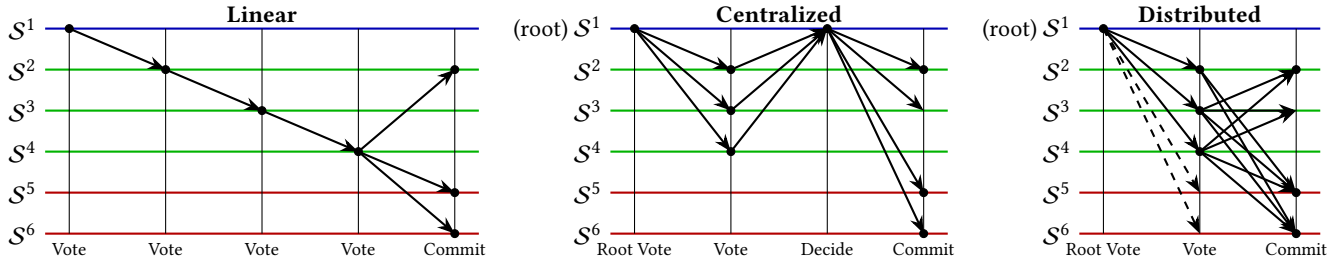

Figure 4.1: Two-phase commit-based orchestration of a transaction τ with $\text{shards}(\tau) = \{\mathcal{S}^1, \ldots, \mathcal{S}^6\}$, in which $\mathcal{S}^1$, $\mathcal{S}^2$, $\mathcal{S}^3$, and $\mathcal{S}^4$ have vote-steps, $\mathcal{S}^2$, $\mathcal{S}^5$, and $\mathcal{S}^6$ have commit steps, and $\mathcal{S}^3$ has an abort-step. Every dot represents a single consensus step, every arrow a single cluster-sending step, and every dashed arrows a cluster-sending step used to set up distributed waiting.

To do so, $\mathcal{S}_r$ forwards a *global abort vote* via cluster-sending to all shards $\mathcal{S} \in \{\mathcal{S}_1, \ldots, \mathcal{S}_n\}$ with an abort-step $\text{Abort}(\mathcal{S})$. All shards $\mathcal{S}$ that receive a global abort vote and voted *abort*, can ignore this vote. All shards $\mathcal{S}$ that receive a global abort vote and voted *commit*, execute $\text{Abort}(\mathcal{S})$ *in parallel*. Finally, we can eliminate the commit-step or abort-step for $\mathcal{S}_r$, as these steps can be processed at the same time as the global vote. We illustrated centralized orchestration in Figure 4.1, *middle*.

We notice that, in the worst case, the root $\mathcal{S}_r$ will receive $n = |\text{shards}(\tau)| - 1$ votes. For efficiency, we cannot use separate *consecutive consensus steps* at $\mathcal{S}_r$ to process each of these incoming votes: if we would use consecutive consensus steps, then receiving these n votes will take worst-case almost-as-long as the steps taken by linear orchestration to perform vote-steps at n shards in sequence. Next, we shall show that we can process these at-most $|\text{shards}(\tau)| - 1$ votes using only *a single consensus decision* at $\mathcal{S}_r$:

Lemma 4.2. *Let τ be a transaction and let shard $\mathcal{S}_r$ be the root that receives commit and abort votes of n other shards. Shard $\mathcal{S}_r$ will receive votes via n cluster-sending steps and can reach a commit or abort decision in at-most a single consensus step at $\mathcal{S}_r$.*

Proof. Consider $\mathcal{S}_r$ receiving votes $v_1, \ldots, v_n$ and let $\text{R}_1, \text{R}_2 \in \mathcal{S}_r$. Both replicas receive votes via cluster-sending and register them in some, possibly distinct, order. Independent of the order in which R_1 and R_2 receive votes, they will both receive the set of votes $\{v_1, \ldots, v_n\}$, receive n_a abort votes, and n_c commit votes, $n_a + n_c = n$. Hence, eventually, R_1 and R_2 can derive the same global commit or abort decision for τ: we *do not* need to enforce a particular ordering in which votes are processed by replicas in $\mathcal{S}_r$ to agree on this decision. We still need to enforce that all replicas in $\mathcal{S}_r$ process this global abort or commit decision for τ in the same order, however. To do so, each replica in $\mathcal{S}_r$ waits until it receives all votes, after which it will use the mechanisms provided by the consensus protocol to trigger a *single consensus step* (e.g., in Pbft by forcing the primary to initiate such step) that reaches agreement on a round in which $\mathcal{S}_r$ continues processing τ (resulting in the global abort or commit decision being shared with other shards). □

In a similar way, shards can process global abort votes with at-most one consensus step. Let $\mathcal{S}_i$, $1 \leq i \leq n$, be a shard. If $\mathcal{S}_i$ voted *abort*, then every replica in $\mathcal{S}_i$ is aware of this vote and can ignore the incoming global abort vote. If $\mathcal{S}_i$ voted *commit*, then every replica in $\mathcal{S}_i$ can use the mechanisms provided by the consensus protocol to reach agreement on a round in which $\mathcal{S}_i$ can execute $\text{Abort}(\mathcal{S}_i)$. Finally, if a shard $\mathcal{S} \in \text{shards}(\tau)$ receives a global commit vote, then every replica in $\mathcal{S}$ can use the mechanisms provided by the consensus protocol to reach agreement on a round in which $\mathcal{S}_i$ can execute $\text{Commit}(\mathcal{S}_i)$. We conclude the following:

Theorem 4.3. *Let τ be a transaction with n_v vote-steps, n_c commit-steps, and n_a abort-steps. Using* centralized orchestration, *τ can be committed (aborted) in exactly four consecutive consensus steps using $n_v + n_c + 1$ (using $n_v + n_a + 1$) consensus-steps and using $2(n_v - 1) + n_c$ (using $2(n_v - 1) + n_a$)) cluster-sending steps.*

Proof. Assume that τ is committed. In this case, the root $\mathcal{S}_r$ first performs its vote-step. Then, all $n_v - 1$ other vote-steps are performed in parallel, resulting in $n_v - 1$ commit votes sent to $\mathcal{S}_r$. Next, using Lemma 4.2, these commit-votes are processed by $\mathcal{S}_r$ using one consensus step. Finally, as the fourth consecutive step, all n_c commit-steps are performed in parallel. Hence, we use $n_v + n_c + 1$ consensus steps, we use $(n_v - 1) + n_c$ cluster-sending steps to forward execution, and $n_v - 1$ cluster-sending steps to send commit votes. The case in which τ is aborted is analogous. □

4.3 Distributed orchestration

Centralized orchestration requires *four* consecutive consensus steps. Next, we propose a method for parallelized orchestration based on the traditional *distributed two-phase commit protocol* (Distributed-2PC) [47] that only requires *three* consecutive consensus steps. We do so by instructing every shard to not just send its vote for *commit* or *abort* to the root, but instead broadcast this vote to *all* shards with either commit-steps or abort-steps.

Let $W \subseteq \text{shards}(\tau)$ be all shards with either a commit-step or an abort-step. Let $\mathcal{S}_i$, $1 \leq i \leq n$, be a shard. Instead of sending the *commit* or *abort* vote resulting from $\text{Vote}(\mathcal{S}_i)$ to $\mathcal{S}_r$, $\mathcal{S}_i$ sends the resulting vote to *all* other shards in W. If $\mathcal{S} \in (W \cap \{\mathcal{S}_1, \ldots, \mathcal{S}_n\})$ voted *abort*, then it can ignore all votes. Let $\mathcal{S}' \in W$ be a shard that did not vote abort. If $\mathcal{S}'$ has a commit-step, then it proceeds with executing $\text{Commit}(\mathcal{S}')$ after it receives n *commit* votes. If $\mathcal{S}'$ has an abort-step, then it proceeds with executing $\text{Abort}(\mathcal{S}')$ after it receives a single *abort* vote. In all other cases, $\mathcal{S}'$ can ignore the votes. We illustrated distributed orchestration in Figure 4.1, *right*.

To assure that each shard in W knows what to do with the votes it receives for τ, the root of τ will not only forward execution to $\mathcal{S}_1, \ldots, \mathcal{S}_n$ with the instruction to vote, but also to all shards in

W with the instruction to wait for votes of shards $\mathcal{S}_1, \ldots, \mathcal{S}_n$ (the wait instructions also implicitly represent the commit vote of the root itself). As with the processing of votes, no consensus step is necessary at the shards in W to process these wait instructions. We conclude the following:

Theorem 4.4. *Let τ be a transaction with n_v vote-steps, n_c commit-steps, and n_a abort-steps. Using* distributed orchestration*, τ can be committed (aborted) in exactly three consecutive consensus steps using n_v+n_c (using n_v+n_a) consensus-steps and using $n_v(n_a+n_c)+(n_v-1)$ (using $n_v(n_a+n_c)+(n_v-1)$) cluster-sending steps.*

Proof. Assume that τ is committed. In this case, the root $\mathcal{S}_r$ first performs its vote-step and sends its commit vote to n_a+n_c shards. Next, all n_v-1 other vote-steps are performed in parallel, resulting in n_v-1 commit votes sent to n_a+n_c shards ($(n_v-1)(n_a+n_c)$ commit votes in total). Finally, as the third consecutive step, each shard with a commit-step can use the techniques of the proof of Lemma 4.2 to process the incoming n_v commit votes and the resulting commit-step using one consensus step. Likewise, each shard with only an abort-step can ignore the commit votes without any consensus steps. Hence, we use n_v+n_c consensus steps, we use n_v-1 cluster-sending steps to forward execution, and $n_v(n_a+n_c)$ cluster-sending steps to send commit votes. The case in which τ is aborted is analogous. □

Remark 4.5. By relying on the client that requested transaction τ to send τ to all shards in shards(τ), we can eliminate the role of the root and reduce distributed orchestration to *two* consecutive consensus steps, this similar to the working of Chainspace [1] and PCerberus [30]. For this to work, we need reliable clients or recovery mechanisms to deal with faulty client behavior, however. As these recovery mechanisms have similar complexity to the *three-step* distributed orchestration we present here, we do not separately investigate such a two-step design.

5 PROVIDING EXECUTION IN OEM

Let τ be a multi-shard transaction. The second part of processing τ is to execute τ by updating any data affected by τ at the shards in shards(τ). As part of execution, one can incorporate steps to assure an *isolated* execution of τ, which makes it easier to maintain *data consistency*. Notice that single-shard steps are ordered via consensus and executed sequentially at the level of a shard. Hence, individual reads and writes always happen in full isolation, guaranteeing *write uncommitted execution* (degree 0 isolation) [5, 6]. As multi-shard transactions can have several shard-steps, the processing of several multi-shard transactions can result in interleaved execution of these transactions. Hence, if further isolation is necessary for the application, then the execution method needs to incorporate some form of concurrency control. To provide concurrency control, we will describe how two-phase locking can be expressed in OEM, this without introducing additional consensus or cluster-sending steps. Using two-phase locking, ByShard provides execution with various degrees of *isolation*, e.g., *serializable execution* (degree 3), *read committed execution* (degree 2), and *read uncommitted execution* (degree 1) [5, 6, 29]. As a baseline, we also describe two basic lock-free execution methods that only provide degree 0 isolation.

To illustrate execution, we formalize the *account-transfer* data and transaction model of preceding examples. For this purpose, we assume that each transaction τ is a pair (C, M) in which C is a set of constraints of the form

$$\textsc{Con}(\text{X}, \text{y}) = \text{“the balance of } X \text{ is at least } y\text{”}$$

and M a set of modifications of the form

$$\textsc{Mod}(\text{X}, \text{y}) = \text{“add } y \text{ to the balance of } X\text{”}.$$

We write $C(\mathcal{S})$ and $M(\mathcal{S})$ to denote the constraints and modifications in C and M, respectively, that affect accounts maintained by $\mathcal{S}$. Semantically, a system *commits* to τ only if all constraints in C hold, in which case all modifications in M are applied to the system. Notice that these minimalistic account-transfer transactions are sufficient to represent all transactions in preceding examples. In Section 7, we discuss why this minimalistic account-transfer data and transaction model is representative for general-purpose workloads for resilient data management systems.

5.1 Isolation-free direct execution

First, we propose a basic execution method with minimal isolation by formalizing the *isolation-free execution method* employed in the linearly orchestrated processing plan of Example 3.3.

Let $\tau = (C, M)$ be a transaction and let $\mathcal{S} \in$ shards(τ). Shard $\mathcal{S}$ needs a vote-step whenever constraints need to be checked at $\mathcal{S}$ ($C(\mathcal{S}) \neq \emptyset$). This vote-step σ checks whether all constraints in $C(\mathcal{S})$ hold. If these constraints hold, then σ makes a commit vote. Otherwise, σ makes an abort vote. To avoid a separate commit-step for τ at $\mathcal{S}$, we optimistically assume that τ will not abort and let the vote-step σ perform all modifications in $M(\mathcal{S})$ after it voted commit. When the transaction gets aborted, we need to roll back any modifications made by σ. Hence, if $M(\mathcal{S}) \neq \emptyset$, we also construct an abort-step Abort($\mathcal{S}$) that rolls back all modifications in $M(\mathcal{S})$ by performing the modifications $\{\textsc{Mod}(\text{X}, \text{-y}) \mid \textsc{Mod}(\text{X}, \text{y}) \in M(\mathcal{S})\}$.

If $\mathcal{S}$ only has modifications ($C(\mathcal{S}) = \emptyset$), then $\mathcal{S}$ only needs a commit-step that performs all modifications in $M(\mathcal{S})$.

The main strength of isolation-free execution is the minimal amount of shard-steps it produces: if a transaction is committed, then each shard will only execute a single shard-step (a vote-step if there are constraints, a commit-step otherwise). Unfortunately, isolation-free execution provides only degree 0 isolation, which can lead to violations of constraints on the data in many applications:

Example 5.1. Consider the sharded banking example of Example 3.1. Assume that the system does not allow negative accounts balances and consider transactions

$\tau_1 = \textsc{Con}(\text{A}, 100), \textsc{Con}(\text{B}, 700), \textsc{Mod}(\text{A}, 400), \textsc{Mod}(\text{B}, \text{-}400)$;

$\tau_2 = \textsc{Con}(\text{A}, 500), \textsc{Mod}(\text{A}, \text{-}300), \textsc{Mod}(\text{E}, 300)$,

and their isolation-free linearly orchestrated execution illustrated in Figure 5.1. As one can see, the balance of A will become negative, breaking the constraint put in place. This is caused by operation $\textsc{Con}(\text{A}, 500)$ of τ_2, which performs a so-called *dirty read* [6, 47].

As isolation-free execution only provides minimal isolation, it is unable to prevent phenomena such as dirty reads that can lead to data inconsistencies. Isolation-free execution does provide *atomicity*, however: either *all* or *none* of the modifications of a transaction are permanent. One way to deal with constraint violations such as in Example 5.1 is by assuring that roll backs do not invalidate

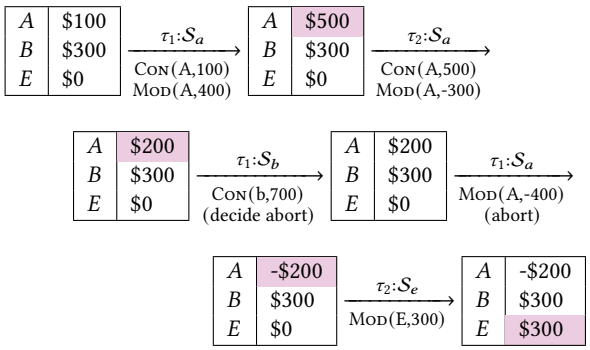

Figure 5.1: Evolution of the *current state* while step-wise executing the transactions of Example 5.1.

constraints in a *domain-specific* manner. To illustrate this, assume we want to assure that accounts never have negative balances.

On the one hand, rolling back Mod(X, y) with $y \leq 0$ (a removal) will *increase* the balance of X and, hence, will never make the balance of X negative. Consequently, these modifications are *safe*. Furthermore, notice that if Con(X, -y) and Mod(X, y), $y \leq 0$, are part of a single vote-step, then they are executed in isolation as a single unit and, hence, the modification will never make the balance negative (this pattern of constraint checking and removing of balance can be seen as a *lock on available resources*, whereas rolling back the removal is a *release of unused resources*).

On the other hand, rolling back Mod(X, y) with $y \geq 0$ (an addition) will *decrease* the balance of X and, hence, can make the balance of X negative. Consequently, these modifications are *unsafe*. To assure that unsafe modification do not invalidate constraints, one can perform these modifications during *commit* (via a commit-step). This means that, in the worst case, every affected shard must execute *two* shard-steps when committing: the vote-step checks constraints and performs safe modifications (which, on abort, are rolled back via the abort-step) and the commit-step performs unsafe modifications. We refer to this execution method, in which safe modifications are part of vote-steps and unsafe modifications are part of commit-steps, as *safe isolation-free execution*.

5.2 Lock-based execution

Although safe isolation-free execution is able to maintain some data consistency, it does so in an domain-specific manner that cannot be applied to all situations. As a more general-purpose method towards maintaining data consistency, we can enforce higher isolation levels for transaction processing, e.g., *degree 3* (serializable execution). The standard way to do so in a multi-shard environment is by using *two-phase locking* [5, 47]. First, we describe the working of two-phase locking. Then, we discuss how to implement two-phase locking with minimal coordination in a Byzantine environment.

Consider a multi-shard transaction τ. When executing τ, τ needs to obtain a *read lock* on each data item D before it reads D and a *write lock* on each data item D before it writes D. Several transactions can hold a read lock on D at the same time, while write locks on D are exclusive: if τ', $\tau' \neq \tau$, holds a write lock on D, then τ cannot obtain any locks on D, and if τ', $\tau' \neq \tau$, holds a read lock on D, then τ cannot obtain a write lock on D, but can obtain a read lock on D. When τ cannot obtain a lock on D that it needs, it simply waits until previous transactions finish and release their locks on D. To provide *serializability*, τ is barred from obtaining new locks after releasing any locks: this assures that there is a point in time where τ is the only transaction that holds all write locks on data items affected by τ, at which point τ can make any changes to these data items in an indivisible atomic manner.

To avoid deadlocks, we enforce that each transaction locks data items in exactly the same order [47]. To minimize the number of shard-steps, we assume a fixed order on shards and on data items within shards, and obtain all locks in that order. Consequently, two-phase locking will require linear orchestration.

Example 5.2. Consider the sharded banking example of Example 3.1 and the transaction

$$\tau = \text{“if } \textit{Ana} \text{ has \$500 and } \textit{Bo} \text{ has \$300 then move \$200 from } \textit{Ana} \text{ to } \textit{Ben}\text{”}.$$

Assume that shards are ordered as $\mathcal{S}_a, \ldots, \mathcal{S}_z$ and that accounts are ordered on account holder name. To execute this transaction, we first obtain a write lock on the account of Ana in $\mathcal{S}_a$, then a write lock on the account of Ben in $\mathcal{S}_b$, and, finally, a read lock on the account of Bo in $\mathcal{S}_b$.

Let $\tau = (C, M)$ be a transaction, let $\mathcal{S} \in \text{shards}(\tau)$, and let

$$\textsc{Accounts}(\mathcal{S}) = \{X \mid \textsc{Con}(\text{X}, \text{y}) \in C(\mathcal{S}) \vee \textsc{Mod}(\text{X}, \text{y}) \in M(\mathcal{S})\}$$

be the set of accounts affected at $\mathcal{S}$. During the vote-step $\textsc{Vote}(\mathcal{S})$, we acquire a lock $\textsc{Lock}(X)$ for every account $X \in \textsc{Accounts}(\mathcal{S})$ in some predetermined order. If there is a $\textsc{Mod}(\text{X}, \text{y}) \in M(\mathcal{S})$, then we acquire a write lock for X. Otherwise, we acquire a read lock. After acquiring the lock on X, we check any constraint $\textsc{Con}(\text{X}, \text{y}) \in C(\mathcal{S})$. If a constraint does not hold, then $\textsc{Vote}(\mathcal{S})$ votes abort and releases all locks already acquired in $\mathcal{S}$. We purposely check these constraints as soon as possible to minimize the amount of time locks are held. Otherwise, if all constraints hold, then $\textsc{Vote}(\mathcal{S})$ votes commit. Next, the commit-step $\textsc{Commit}(\mathcal{S})$ performs all modifications $M(\mathcal{S})$, after which it performs $\textsc{Release}(X)$, for all accounts $X \in \textsc{Accounts}(\mathcal{S})$, to release all locks in $\mathcal{S}$. Finally, the abort-step $\textsc{Abort}(\mathcal{S})$ performs $\textsc{Release}(X)$, for all accounts $X \in \textsc{Accounts}(\mathcal{S})$, to release all locks in $\mathcal{S}$. We have the following:

Theorem 5.3. *Let τ be a transaction with $n = |\text{shards}(\tau)|$. To process τ using two-phase locking, we need n vote-steps to obtain all locks, followed by $n-1$ commit-steps or abort-steps to release all locks. Hence, τ can be processed using $2n-1$ consensus steps and $2n-2$ cluster-sending steps.*

Proof. To prove the theorem, we only need to prove that the vote-step $\textsc{Vote}(\mathcal{S})$ of each shard $\mathcal{S} \in \text{shards}(\tau)$ can obtain all its locks using only a single consensus step at $\mathcal{S}$. Execution of $\textsc{Vote}(\mathcal{S})$ starts after $\mathcal{S}$ reached consensus on this step, and we will prove that no further consensus steps for $\textsc{Vote}(\mathcal{S})$ are required. Let $\textsc{Vote}(\mathcal{S}) = \{\ldots, \textsc{Lock}(X), \ldots\}$. During execution of $\textsc{Vote}(\mathcal{S})$, we distinguish two possible cases:

(1) The lock on X *can* be obtained, in which case execution of $\textsc{Vote}(\mathcal{S})$ continuous.

(2) The lock on X *cannot* be obtained. In this case, execution of $\text{VOTE}(\mathcal{S})$ needs to wait until the lock on X can be obtained. To do so, as part of the execution of $\text{VOTE}(\mathcal{S})$, every replica $\text{R} \in \mathcal{S}$ puts $(\tau, \text{VOTE}(\mathcal{S}))$ on a wait-queue $Q_{\text{R}}(X)$.

Let $\text{R}_1, \text{R}_2 \in \mathcal{S}$. We assume that wait-queues $Q_{\text{R}_1}(X)$ and $Q_{\text{R}_2}(X)$ operate *deterministic*: if the the same sequence of operations is applied to $Q_{\text{R}_1}(X)$ and $Q_{\text{R}_2}(X)$, then the queues always yield the same results. Now consider the case in which the lock X cannot be obtained. Let τ', $\tau \neq \tau'$, be the transaction that is holding the lock on X and let $\sigma = \{\ldots, \text{RELEASE}(X), \ldots\}$ be the commit-step or abort-step of τ' for shard $\mathcal{S}$. During execution, shard-step σ will release the lock on X. When doing so, each replica $\text{R} \in \mathcal{S}$ wakes up transactions in $Q_{\text{R}}(X)$ for execution directly after shard-step σ. We distinguish two cases:

(1) The next transaction in $Q_{\text{R}}(X)$ wants to obtain a read lock, while τ' held a write lock. In this case, wake up all transactions in $Q_{\text{R}}(X)$ that want to obtain a read lock (all these transactions can hold the read lock at the same time).
(2) The next transaction in $Q_{\text{R}}(X)$ wants to obtain a write lock. If τ' was the last transaction holding any lock on X, then we wake up the next transaction (as it will be the only one that can hold an exclusive write lock).

This wake up step is part of the deterministic execution of σ and wake-up queues operate deterministic. Hence, *no* consensus steps are necessary to determine which transactions need to be executed next and to initiate execution of these next transactions. □

We notice that we cannot always minimize the number of affected shards while processing τ via two-phase locking:

Example 5.4. Consider the sharded banking example of Example 5.2 and the transaction τ = "if *Bo* has \$500, then move \$200 from *Bo* to *Ana*". Due to the ordering on shards and accounts used, we *always first* need to obtain a write lock on the account of Ana (in shard $\mathcal{S}_a$) before we can inspect the balance of Bo (in shard $\mathcal{S}_b$), even if Bo does not have sufficient balance. This is in contrast with the isolation-free execution methods, as these methods can *first* inspect the balance of Bo and directly abort execution.

The strength of two-phase locking is that it provides serializability. The downside is that it can cause large transaction processing latencies whenever *contention* is high:

Example 5.5. Consider a system in which consensus steps take $t = 30$ ms each, while all other steps take negligible time (see Remark 2.2). We consider transactions τ_1 and τ_2 such that τ_1 writes to data items $D_1, \ldots, D_{10}$ that are held in shards $\mathcal{S}_1, \ldots, \mathcal{S}_{10}$, respectively, while τ_2 only writes to data item D_1. Transaction τ_1 executes first at $\mathcal{S}_1$ and obtains the write lock on D_1. Next, τ_2 executes at $\mathcal{S}_1$, cannot obtain the write lock on D_1, and has to wait until τ_1 finishes execution and releases the lock on D_1. To do so, τ_1 has to first obtain locks in $m - 1$ shards, after which it can return to $\mathcal{S}_1$ to release the lock on D_1. Hence, τ_1 has to perform m consecutive consensus steps. Even if τ_1 can obtain the locks on $D_2, \ldots, D_{10}$ immediately, it will take at least $10t = 300$ ms before τ_2 can resume execution, even though the actual execution of τ_2 would only take $t = 30$ ms.

One way to partially deal with Example 5.5 is by not imposing degree 3 isolation (serializable execution). The lock primitives we propose to provide degree 3 isolation can easily be used to provide execution methods with other levels of isolation [5, 6, 29]. E.g.,

(1) in *read uncommitted execution* (degree 1 isolation), no read locks are obtained on any data item (while write locks are used in the usual way), thereby reducing lock contention sharply for read-heavy workloads; and
(2) in *read committed execution* (degree 2 isolation), read locks on each data item D are released directly after reading D (while write locks are used in the usual way), thereby minimizing the time read locks are held.

Using lower isolation levels only partially mitigates the issues illustrated in Example 5.5. To further deal with this, one can opt to replace *waiting* by *failing*: whenever a lock cannot be obtained by a transaction τ, τ aborts. This approach guarantees that processing latencies of transactions and resource utilization at the replicas are kept in check in periods of high contention, this at the cost of aborted transactions that could otherwise be successfully executed. As these *non-blocking locks* will never cause deadlocks, these locks can be obtained in any order, enabling their usage in combination with *all* orchestration methods.

6 PERFORMANCE EVALUATION OF BYSHARD

In the previous sections, we introduced BYSHARD as a framework for sharded resilient systems. We also presented several general-purpose methods by which BYSHARD can effectively orchestrate and execute multi-shard transactions. Combining these methods results in *eighteen* multi-shard transaction processing protocols. See the legend of Figure 6.1 for details on how these eighteen protocols are obtained from the presented orchestration and execution methods.

Remark 6.1. In practical deployments of BYSHARD, end-users only need to use one of these eighteen multi-shard transaction processing protocols. In our experiments, we use such single-protocol deployments, as we are interested in the differences between the protocols. This does not rule out deployments of BYSHARD that use several protocols simultaneously. Indeed, BYSHARD does support the usage of several protocols at the same time such that users can select the appropriate isolation level for individual transactions.

Furthermore, multi-shard transaction protocols used by sharded resilient systems such as AHL [15] and CHAINSPACE [1] can also easily be expressed within the orchestrate-execute model of BYSHARD.

To gain further insight in the performance attainable by sharded resilient systems, we implemented the BYSHARD framework, the orchestrate-execution model, and the eighteen multi-shard transaction processing protocols obtained from the presented orchestration an execution methods.

For comparison, we also implemented the multi-shard transaction protocol of AHL [15], which has a novel design that is most similar to the design of our *Centralized, Serializable, non-blocking* protocol CSNB: the main difference being that AHL uses a dedicated *reference committee* to coordinate processing of all multi-shard transactions, whereas in CSNB each transaction is coordinated by a root-shard chosen from the set of shards affected by that transaction. Our implementation of AHL is granted a dedicated extra shard for use as the reference committee. Finally, we note that the design of our *Distributed, Serializable, non-blocking* protocol DSNB

is a generalization of the novel design of Chainspace [1]. We refer to Remark 4.5 for further details on the relationship between the three-step design of DSnb and the two-step design of Chainspace.

Next, we deployed our implementation on a simulated sharded resilient system. In specific, we abstract the operations of *consensus* and *cluster-sending*, while deploying full shards that execute all replica-specific operations necessary for transaction orchestration and execution. This deployment provides detailed control over consensus and cluster-sending costs, enables fine-grained measurements of performance metrics, and allows us to deploy on hundreds of shards.[1]

6.1 Experimental Setup

In each experiment, we run a workload of 5000 transactions. Unless specified otherwise, each transaction affects 16 distinct accounts by putting constraints on 8 accounts (read operations), removing balance from 4 accounts (write operations), and adding balance to 4 accounts (write operations). The accounts affected by these operations are chosen uniformly at random from a set of active accounts. Each account on each shard starts with an initial balance of 2000 and transactions add or remove 500 balance per modification (on average, these are chosen via a binomial distribution with $n = 1000$ and $p = 0.5$). We run experiments with 64 shards and 8192 active accounts (128 active accounts per shard). Finally, the experiments are set up such that the message delay is 10 ms, consensus decisions take 30 ms to make, and each shard can make 1000 decisions/s. In each experiment, we collected *five* core measurements on the performance of the system:

- The *total runtime* represents the elapsed real time.
- The *cumulative duration* represents the sum of the transaction duration (the elapsed real time to process that transaction) of each transaction in the workload, which includes waiting times.
- The *average throughput* represents the average number of transactions processed per second.
- The *average committed throughput* represents the average number of transactions committed (and executed) per second.
- The *median shard-steps* represent the median number of shard-steps performed per shard (each shard-step involves a single consensus step).

We performed *three experiments*, the results of which can be found in Figure 6.1.

The Scalability Experiment. In our first experiment, we inspect the impact of *sharding* on the behavior of ByShard. To do so, we measured the behavior of the system as a function of the *number of shards* while keeping all other parameters the same (including the workload and the initial dataset). By increasing the number of shards, we increase the available parallel processing power. At the same time, we decrease the number of accounts per shard and, hence, we increase the number of multi-shard transactions and the average number of shards affected by each transaction in our workload.

[1]The full C++ implementation of these experiments and the raw measurements are available at https://www.jhellings.nl/projects/byshard/.

The Contention Experiment. In our second experiment, we inspect the impact of *contention* on the behavior of ByShard. To do so, we measured the behavior of the system as a function of the *number of active accounts per shard*. For each case, we generate appropriate workloads as a function of the the overall number of active accounts in the system. Increasing the number of active accounts will decrease the probability that two transactions affect the same account and, hence, decrease contention.

The Factor-Scalability Experiment. In our third experiment, we inspect the impact of *scaling the system* on the behavior of ByShard. To do so, we measured the behavior of the system as a function of the *number of shards* and, as we keep the number of active accounts per shard constant, the *number of active accounts*. For each case, we generate appropriate workloads as a function of the the overall number of accounts in the system. As in the *scalability experiment*, increasing the number of shards increases the available parallel processing power. Furthermore, as in the *contention experiment*, increasing the number of accounts decreases contention.

6.2 Experimental Results

From the presented measurements, we conclude that the eighteen multi-shard transaction protocols we propose have excellent scalability: when the number of shards is increased, the median amount of work per shard (consensus steps and vote processing steps) decreases rapidly. A closer look at the protocols shows that the main difference between them is, on the one hand, their runtime, per-transaction duration, and throughput and, on the other hand, the average committed throughput.

As expected, we see that the protocols that provide lower degrees of isolation have the lowest runtime and lowest average durations, and, consequently, the highest throughput. Furthermore, we see that centralized and distributed orchestration have comparable performance, while outperforming linear orchestration.

When focusing on lock-based execution, we see that using blocking locks result in systems with good runtimes, high commit rates, and scalable performance whenever contention is low. As Example 5.5 already illustrated, the performance of these protocols is sharply affected by contention, however. At the same time, using non-blocking locks results in systems with all-around low runtimes and low average durations. These systems are affected by contention, however, as higher contention results in higher rates of aborted transactions due to lock failures. Still, due to the much lower runtimes, using non-blocking locks typically results in much higher throughputs than using blocking locks.

Finally, we see that protocols that use either centralized or distributed orchestration have much lower runtimes than their linear orchestrated counterparts. At the same time, the parallel processing of shard-steps increases the negative impact of contention on the committed throughput (e.g., due to higher rates of constraint failures). Still, the rate at which transactions are committed is highest using distributed orchestration, as the high throughput achieved by distributed orchestration offsets the negative impact of contention.

When looking at AHL, we see that the novel design of AHL provides excellent scalability for workloads with high ratios of single-shard transactions. When comparing AHL with similar protocols in ByShard, we see that the reliance on a reference committee in

	Isolation-Free execution (write uncommitted)		Lock-based execution							
			Read Uncommitted		Read Committed		Serializable			
	unsafe	*safe*	*blocking*	*non-blocking*	*blocking*	*non-blocking*	*blocking*	*non-blocking*		
Linear	LIFu	LIFs	LRUb	LRUnb	LRCb	LRCnb	LSb	LSnb	AHL	(reference committee)
Centralized	CIFu	CIFs		CRUnb		CRCnb		CSnb		
Distributed	DIFu	DIFs		DRUnb		DRCnb		DSnb		

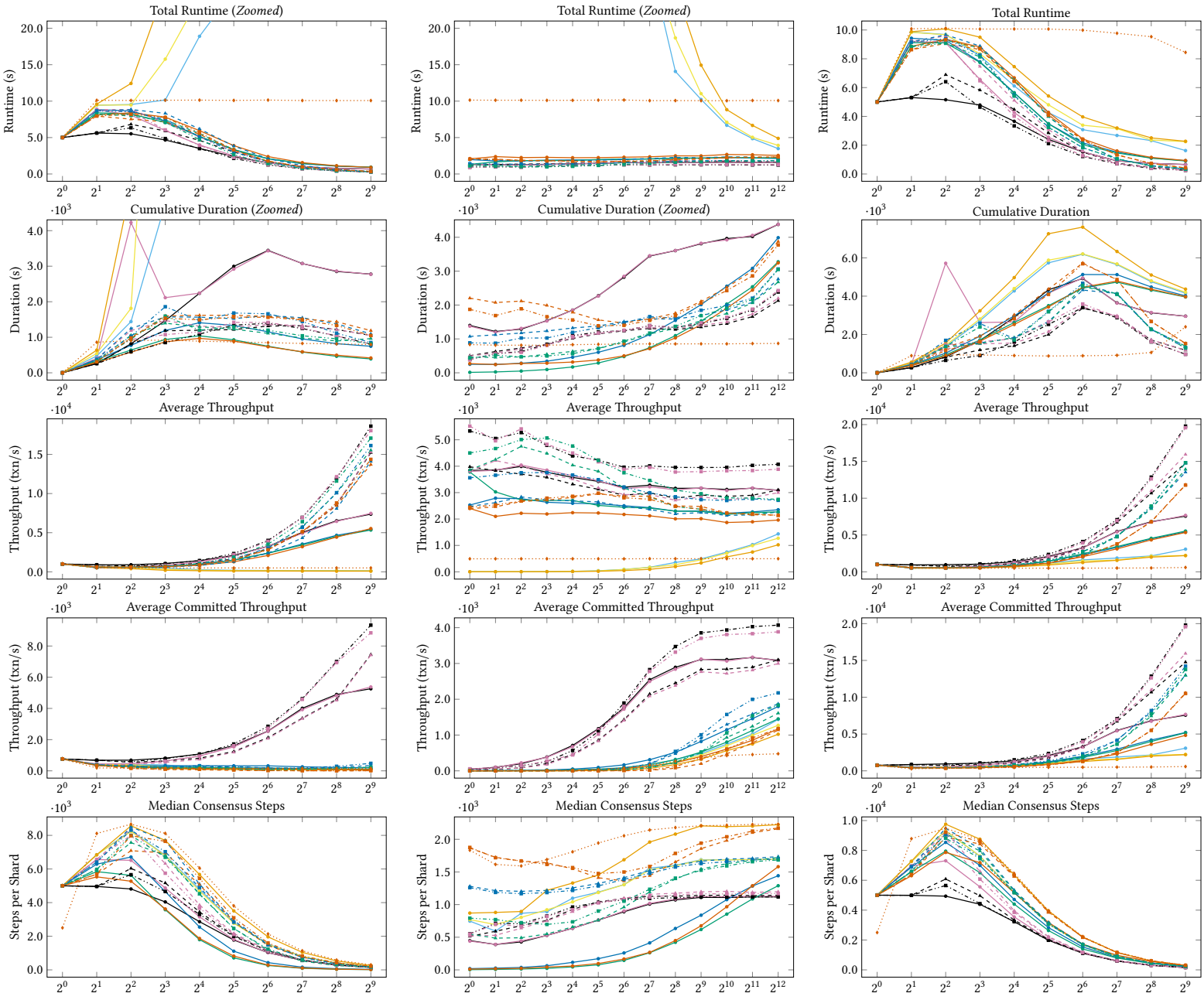

Figure 6.1: Performance characteristics of processing a workload of 5000 transactions. *Left*, the results for the *scalability experiment* in which we measured the behavior of the system when processing a fixed workload as a function of the *number of shards* . *Middle*, the results for the *contention experiment* in which we measured the behavior of a system with 64 shards as a function of the *number of accounts per shard*. *Right*, the results for the *factor-scalability experiment* in which we measured the behavior of the system as a function of the *scalability-factor* by which the number of accounts and shards grow. See Section 6.1 for a detailed description of each experiment and each measurement type.

AHL to orchestrate all multi-shard transactions reduces contention. Unfortunately, this comes at the cost of overall performance: in line with the original evaluation of AHL [15, Section 7.3], we see that the reference committee is a bottleneck when dealing with workloads with high ratios of multi-shard transactions.

7 DISCUSSION AND RELATED WORK

There is abundant literature on the design and implementation of distributed systems, distributed databases, and sharding (e.g., [47, 54, 55]). Furthermore, there is also abundant literature on resilient systems and consensus (e.g., [7, 9, 14, 16, 40, 58]). Next, we shall focus on the design decisions made by BySHARD and compare them to the few works that deal with sharding in resilient systems.

Workloads in resilient systems. The *account-transfer* data and transaction model we used throughout this paper is simple. Still, all principles outlined in this paper can be applied to *any* data and transaction model in which transactions are *one-shot transactions*. We have not considered *interactive transactions* that require back-and-forth steps by clients and the system. Although such interactive transactions are supported by some traditional data management systems, we believe that they are ill-suited for resilient systems, as interactive transaction processing would be a costly and unresponsive process due to the high cost of the individual consensus steps required to process each back-and-forth step.

The general-purpose data and transaction model of BySHARD is in contrast with CHAINSPACE and CERBERUS, which both opt to utilize an UTXO-based data model to their advantage to provide consistent transaction execution when dealing with contention and Byzantine behavior.

BySHARD *and decentralized sharding.* The designs of the eighteen multi-transaction protocols of BySHARD are *decentralized* in the sense that there is no central coordinator that is assigned the task to coordinate execution of all multi-shard transactions. This is in contrast with systems such as AHL [15] that use a reference committee to coordinate execution of all multi-shard transactions. This difference between BySHARD and AHL is not fundamental, however, as the multi-shard transaction protocol of AHL can easily be expressed within the orchestrate-execute model of BySHARD.

As shown in Section 6.2, the usage of a central coordinator (e.g., reference committees) can significantly reduce contention while providing excellent scalability for single-shard workloads. At the same time, the usage of a central coordinator introduces bottlenecks when processing workloads with many multi-shard transactions.

BySHARD *and the usage of Byzantine primitives.* To maximize the throughput of a sharded resilient system, we have to assure that standard performance-enhancing techniques can be applied at the single-shard level. This is especially true for *out-of-order processing* [11, 15, 26], which can increase consensus throughput in consensus-based systems by several orders of magnitudes (see Remark 2.2). In BySHARD, we have assured that such performance-enhancing techniques are easily applicable by utilizing standard Byzantine primitives as basic building blocks.

This is in contrast with recent systems such as CAPER [2] and SHARPER [3] that try to minimize the *duration* of multi-shard transaction processing. To do so, these systems process each multi-shard transactions via a *single* transaction-specific multi-shard-aware consensus step (thereby reducing the number of consecutive consensus steps to an absolute minimum). Unfortunately, such designs have difficulties dealing with contention, while making it nontrivial to apply standard performance-enhancing techniques such as out-of-order processing.

As BySHARD relies on standard Byzantine primitives, the design of BySHARD is highly flexible and can easily be tuned towards specific applications. E.g., by providing only crash-fault tolerance by using the PAXOS consensus protocol, by minimizing communication costs by using the HOTSTUFF consensus protocol, and so on.

Sharding in permissionless blockchains. In parallel to the development of traditional resilient systems and permissioned blockchains, there has been promising work on sharding in permissionless blockchains such as BITCOIN [41] and ETHEREUM [56]. Examples include techniques for enabling reliable cross-chain coordination via sidechains, blockchain relays, atomic swaps, atomic commitment, and cross-chain deals [20, 21, 32, 34, 38, 57, 60]. Unfortunately, these permissionless techniques are several orders of magnitudes slower than comparable techniques for traditional resilient systems, making them unsuitable for systems that aim at high-performance data management.

8 CONCLUSION

In this paper, we introduced BySHARD, a general-purpose framework for sharded resilient data management systems. Additionally, we introduced the *orchestrate-execute model* (OEM) for processing multi-shard transactions in BySHARD. Next, we showed that OEM can incorporate the necessary commit, locking, and execution steps required for processing a multi-shard transaction in at-most two consensus steps per involved shard. Furthermore, we showed that common multi-shard transaction processing based on two-phase commit protocols and two-phase locking can be expressed efficiently in OEM.

Our flexible design allows for many distinct approaches towards multi-shard transaction processing, each striking its own trade-off between *throughput*, *isolation level*, *latency*, and *abort rate*. To illustrate this, we performed an in-depth comparison of the *eighteen* multi-shard transaction processing protocols of BySHARD. Our results show that each protocol supports high transaction throughput and provides scalability. Hence, we believe that the BySHARD framework is a promising step towards flexible general-purpose ACID-compliant scalable resilient multi-shard data and transaction processing capabilities.

REFERENCES

[1] Mustafa Al-Bassam, Alberto Sonnino, Shehar Bano, Dave Hrycyszyn, and George Danezis. 2017. Chainspace: A Sharded Smart Contracts Platform. http://arxiv.org/abs/1708.03778

[2] 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

[3] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. 2020. SharPer: Sharding Permissioned Blockchains Over Network Clusters. https://arxiv.org/abs/1910.00765v2

[4] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. 2018. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. In *Proceedings of the Thirteenth EuroSys Conference*. ACM, 30:1–30:15. https://doi.org/10.1145/3190508.3190538

[5] Vijayalakshmi Atluri, Elisa Bertino, and Sushil Jajodia. 1997. A theoretical formulation for degrees of isolation in databases. *Inform. Software Tech.* 39, 1 (1997), 47–53. https://doi.org/10.1016/0950-5849(96)01109-3

[6] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. 1995. A Critique of ANSI SQL Isolation Levels. *SIGMOD Rec.* 24, 2 (1995), 1–10. https://doi.org/10.1145/568271.223785

[7] Christian Berger and Hans P. Reiser. 2018. Scaling Byzantine Consensus: A Broad Analysis. In *Proceedings of the 2nd Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers*. ACM, 13–18. https://doi.org/10.1145/3284764.3284767

[8] Burkhard Blechschmidt. 2018. *Blockchain in Europe: Closing the Strategy Gap.* Technical Report. Cognizant Consulting. https://www.cognizant.com/whitepapers/blockchain-in-europe-closing-the-strategy-gap-codex3320.pdf

[9] Christian Cachin and Marko Vukolic. 2017. Blockchain Consensus Protocols in the Wild (Keynote Talk). In *31st International Symposium on Distributed Computing*, Vol. 91. Schloss Dagstuhl, 1:1–1:16. https://doi.org/10.4230/LIPIcs.DISC.2017.1

[10] Michael Casey, Jonah Crane, Gary Gensler, Simon Johnson, and Neha Narula. 2018. *The Impact of Blockchain Technology on Finance: A Catalyst for Change.* Technical Report. International Center for Monetary and Banking Studies. https://www.cimb.ch/uploads/1/1/5/4/115414161/geneva21_1.pdf

[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] Christie's. 2018. Major Collection of the Fall Auction Season to be Recorded with Blockchain Technology. https://www.christies.com/presscenter/pdf/9160/RELEASE_ChristiesxArtoryxEbsworth_9160_1.pdf

[13] Cindy Compert, Maurizio Luinetti, and Bertrand Portier. 2018. *Blockchain and GDPR: How blockchain could address five areas associated with GDPR compliance.* Technical Report. IBM Security. https://public.dhe.ibm.com/common/ssi/ecm/61/en/61014461usen/security-ibm-security-solutions-wg-white-paper-external-61014461usen-20180319.pdf

[14] Miguel Correia, Giuliana Santos Veronese, Nuno Ferreira Neves, and Paulo Verissimo. 2011. Byzantine Consensus in Asynchronous Message-Passing Systems: A Survey. *Int. J. Crit. Comput.-Based Syst.* 2, 2 (2011), 141–161.

[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] Tien Tuan Anh Dinh, Rui Liu, Meihui Zhang, Gang Chen, Beng Chin Ooi, and Ji Wang. 2018. Untangling Blockchain: A Data Processing View of Blockchain Systems. *IEEE Trans. Knowl. Data Eng.* 30, 7 (2018), 1366–1385. https://doi.org/10.1109/TKDE.2017.2781227

[17] D. Dolev. 1981. Unanimity in an unknown and unreliable environment. In *22nd Annual Symposium on Foundations of Computer Science*. IEEE, 159–168. https://doi.org/10.1109/SFCS.1981.53

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

[19] Wayne W. Eckerson. 2002. *Data quality and the bottom line: Achieving Business Success through a Commitment to High Quality Data.* Technical Report. The Data Warehousing Institute, 101communications LLC.

[20] Muhammad El-Hindi, Carsten Binnig, Arvind Arasu, Donald Kossmann, and Ravi Ramamurthy. 2019. BlockchainDB: A Shared Database on Blockchains. *Proc. VLDB Endow.* 12, 11 (2019), 1597–1609. https://doi.org/10.14778/3342263.3342636

[21] Ethereum Foundation. 2017. BTC Relay: A bridge between the Bitcoin blockchain & Ethereum smart contracts. http://btcrelay.org

[22] Lan Ge, Christopher Brewster, Jacco Spek, Anton Smeenk, and Jan Top. 2017. *Blockchain for agriculture and food: Findings from the pilot study.* Technical Report. Wageningen University. https://www.wur.nl/nl/Publicatie-details.htm?publicationId=publication-way-353330323634

[23] William J. Gordon and Christian Catalini. 2018. Blockchain Technology for Healthcare: Facilitating the Transition to Patient-Driven Interoperability. *Comput. Struct. Biotechnol. J.* 16 (2018), 224–230. https://doi.org/10.1016/j.csbj.2018.06.003

[24] Jim Gray. 1978. Notes on Data Base Operating Systems. In *Operating Systems, An Advanced Course*. Springer-Verlag, 393–481. https://doi.org/10.1007/3-540-08755-9_9

[25] GSM Association. 2017. Blockchain for Development: Emerging Opportunities for Mobile, Identity and Aid. https://www.gsma.com/mobilefordevelopment/wp-content/uploads/2017/12/Blockchain-for-Development.pdf

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

[27] 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

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

[29] Theo Haerder and Andreas Reuter. 1983. Principles of Transaction-Oriented Database Recovery. *ACM Comput. Surv.* 15, 4 (1983), 287–317. https://doi.org/10.1145/289.291

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

[31] Jelle Hellings and Mohammad Sadoghi. 2019. Brief Announcement: The Fault-Tolerant Cluster-Sending Problem. In *33rd International Symposium on Distributed Computing (DISC 2019)*. Schloss Dagstuhl, 45:1–45:3. https://doi.org/10.4230/LIPIcs.DISC.2019.45

[32] Maurice Herlihy. 2018. Atomic Cross-Chain Swaps. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*. ACM, 245–254. https://doi.org/10.1145/3212734.3212736

[33] Maurice Herlihy. 2019. Blockchains from a Distributed Computing Perspective. *Commun. ACM* 62, 2 (2019), 78–85. https://doi.org/10.1145/3209623

[34] Maurice Herlihy, Barbara Liskov, and Liuba Shrira. 2019. Cross-Chain Deals and Adversarial Commerce. *Proc. VLDB Endow.* 13, 2 (2019), 100–113. https://doi.org/10.14778/3364324.3364326

[35] Thomas N. Herzog, Fritz J. Scheuren, and William E. Winkler. 2007. *Data Quality and Record Linkage Techniques.* Springer. https://doi.org/10.1007/0-387-69505-2

[36] Matt Higginson, Johannes-Tobias Lorenz, Björn Münstermann, and Peter Braad Olesen. 2017. *The promise of blockchain.* Technical Report. McKinsey&Company. https://www.mckinsey.com/industries/financial-services/our-insights/the-promise-of-blockchain

[37] Maged N. Kamel Boulos, James T. Wilson, and Kevin A. Clauson. 2018. Geospatial blockchain: promises, challenges, and scenarios in health and healthcare. *Int. J. Health. Geogr* 17, 1 (2018), 1211–1220. https://doi.org/10.1186/s12942-018-0144-x

[38] Jae Kwon and Ethan Buchman. 2019. Cosmos Whitepaper: A Network of Distributed Ledgers. https://cosmos.network/cosmos-whitepaper.pdf

[39] Leslie Lamport. 2001. Paxos Made Simple. *ACM SIGACT News* 32, 4 (2001), 51–58. https://doi.org/10.1145/568425.568433 Distributed Computing Column 5.

[40] Laphou Lao, Zecheng Li, Songlin Hou, Bin Xiao, Songtao Guo, and Yuanyuan Yang. 2020. A Survey of IoT Applications in Blockchain Systems: Architecture, Consensus, and Traffic Modeling. *ACM Comput. Surv.* 53, 1, Article 18 (2020), 32 pages. https://doi.org/10.1145/3372136

[41] Satoshi Nakamoto. [n.d.]. Bitcoin: A Peer-to-Peer Electronic Cash System. https://bitcoin.org/en/bitcoin-paper

[42] Arvind Narayanan and Jeremy Clark. 2017. Bitcoin's Academic Pedigree. *Commun. ACM* 60, 12 (2017), 36–45. https://doi.org/10.1145/3132259

[43] Senthil Nathan, Chander Govindarajan, Adarsh Saraf, Manish Sethi, and Praveen Jayachandran. 2019. Blockchain Meets Database: Design and Implementation of a Blockchain Relational Database. *Proc. VLDB Endow.* 12, 11 (2019), 1539–1552. https://doi.org/10.14778/3342263.3342632

[44] Faisal Nawab and Mohammad Sadoghi. 2019. Blockplane: A Global-Scale Byzantizing Middleware. In *35th International Conference on Data Engineering (ICDE)*. IEEE, 124–135. https://doi.org/10.1109/ICDE.2019.00020

[45] Dick O'Brie. 2017. *Internet Security Threat Report: Ransomware 2017, An ISTR Special Report.* Technical Report. Symantec. https://www.symantec.com/content/dam/symantec/docs/security-center/white-papers/istr-ransomware-2017-en.pdf

[46] 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

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

[48] Michael Pisa and Matt Juden. 2017. *Blockchain and Economic Development: Hype vs. Reality*. Technical Report. Center for Global Development. https://www.cgdev.org/publication/blockchain-and-economic-development-hype-vs-reality
[49] PwC. 2016. Blockchain – an opportunity for energy producers and consumers? https://www.pwc.com/gx/en/industries/energy-utilities-resources/publications/opportunity-for-energy-producers.html
[50] Thomas C. Redman. 1998. The Impact of Poor Data Quality on the Typical Enterprise. *Commun. ACM* 41, 2 (1998), 79–82. https://doi.org/10.1145/269012.269025
[51] David Reinsel, John Gantz, and John Rydning. 2018. *Data Age 2025: The Digitization of the World, From Edge to Core*. Technical Report. IDC. https://www.seagate.com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf
[52] Dale Skeen. 1982. *A Quorum-Based Commit Protocol*. Technical Report. Cornell University.
[53] Symantec. 2018. Internet Security Threat Report, Volume 32. https://www.symantec.com/content/dam/symantec/docs/reports/istr-23-2018-en.pdf
[54] Gerard Tel. 2001. *Introduction to Distributed Algorithms* (2nd ed.). Cambridge University Press.
[55] Maarten van Steen and Andrew S. Tanenbaum. 2017. *Distributed Systems* (3th ed.). Maarten van Steen. https://www.distributed-systems.net/
[56] Gavin Wood. [n.d.]. Ethereum: a secure decentralised generalised transaction ledger. https://gavwood.com/paper.pdf EIP-150 revision.
[57] Gavin Wood. 2016. Polkadot: vision for a heterogeneous multi-chain framework. https://polkadot.network/PolkaDotPaper.pdf
[58] Yang Xiao, Ning Zhang, Wenjing Lou, and Y. Thomas Hou. 2020. A Survey of Distributed Consensus Protocols for Blockchain Networks. *IEEE Commun. Surv. Tutor* 22, 2 (2020), 1432–1465. https://doi.org/10.1109/COMST.2020.2969706
[59] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. 2019. HotStuff: BFT Consensus with Linearity and Responsiveness. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*. ACM, 347–356. https://doi.org/10.1145/3293611.3331591
[60] Victor Zakhary, Divyakant Agrawal, and Amr El Abbadi. 2020. Atomic Commitment across Blockchains. *Proc. VLDB Endow.* 13, 9 (2020), 1319–1331. https://doi.org/10.14778/3397230.3397231