PROBLEM: SHARDED RESILIENT TRANSACTION PROCESSING WITH MINIMAL COSTS

Anonymous authors
Paper under double-blind review

Abstract

To enable scalable resilient blockchain systems, several powerful general-purpose approaches toward sharding such systems have been demonstrated. Unfortunately, these approaches all come with substantial costs for ordering and execution of multi-shard transactions.

In this work, we ask whether one can achieve significant cost reductions for processing multi-shard transactions by limiting the type of workloads supported. To initiate the study of this problem, we propose core-CHIMERA (CCHIMERA). CCHIMERA uses *strict UTXO-based environmental requirements* to enable powerful multi-shard transaction processing with an absolute minimum amount of coordination between shards. In the environment we designed CCHIMERA for, CCHIMERA will operate *perfectly* with respect to all transactions proposed and approved by well-behaved clients, but does not provide any other guarantees.

To illustrate that CCHIMERA-like protocols can also be of use in environments with faulty clients, we also demonstrate *two* generalizations of CCHIMERA, *optimistic*-CHIMERA and *resilient*-CHIMERA, that make different tradeoffs in complexity and costs when dealing with faulty behavior and attacks. Finally, we compare these three protocols and show their potential scalability and performance benefits over state-of-the-art general-purpose systems. These results underline the importance of the study of specialized approaches toward sharding in resilient systems.

1 Introduction

The advent of blockchain applications and technology has rejuvenated interest of companies, governments, and developers in resilient distributed fully-replicated systems and the distributed ledger technology (DLT) that powers them. Indeed, in the last decade we have seen a surge of interest in reimagining systems and build them using DLT networks. Examples can be found in the financial and banking sector [15, 40, 54], IoT [46], health care [29, 41], supply chain tracking, advertising, and in databases [5,23,31,32,51,52]. This wide interest is easily explained, as blokchains promise to improve resilience against both failures and malicious behavior, while enabling the federated management of data by many participants.

To illustrate this, we look at the financial sector. Current traditional banking infrastructure is often rigid, slow, and creates substantial frictional costs. It is estimated that the yearly cost of transactional friction alone is $71 billion [8] in the financial sector, creating a strong desire for alternatives. This sector is a perfect match for DLT, as it enables systems that manage digital assets and financial transactions in more flexible, fast, and open federated infrastructures that eliminate the friction caused by individual private databases maintained by banks and financial services providers. Consequently, it is expected that a large part of the financial sector will move towards DLT [18].

At the core of DLT is the *replicated state* maintained by the network in the form of a ledger of transactions. In traditional blockchains, this ledger is fully replicated among all participants using consensus protocols [14, 31, 38, 46, 50]. For many practical use-cases, one can choose to use either permissionless consensus solutions that are operated via economic self-incentivization through cryptocurrencies (e.g., Nakamoto consensus [49, 59]), or permissioned consensus solutions that require vetted participation (e.g, PBFT, POE, and HOTSTUFF [16, 33, 61]). Unfortunately, the design of consensus protocols are severely limited in their ability to provide the *high transaction throughput* that is needed to address practical needs, e.g., in the financial sector. Indeed, on the one hand, we see that permissionless solutions can easily scale to thousands of participants, but are severely limited in their transaction processing throughput. E.g., in Ethereum, a popular public permissionless DLT platform, the rapid growth of decentralized finance applications [12] causes its network fees to rise precipitously as participants bid for limited network capacity [7], while Bitcoin can only process a few transactions per second [54]. On the other hand, solutions that use permissioned consensus protocols such as PBFT can reach much higher throughput. Typically, these solutions are still fully-replicated resilient systems, however. Hence, the speed by which individual replicas can process transactions provides an upper-bound on the performance of these solutions, ruling out scalability. Furthermore, adding replicas will actively decrease performance of these solutions, as full replication among more replicas increases the cost of full replication (e.g., via consensus). As such, these solutions lack the scalability required by many modern data-based applications.

Recently, several general-purpose consensus-based systems have been proposed that use *sharding* to combat the limitations of fully-replicated consensus-based systems [1, 3, 4, 17, 35,55]. In these systems, one partitions the data among several

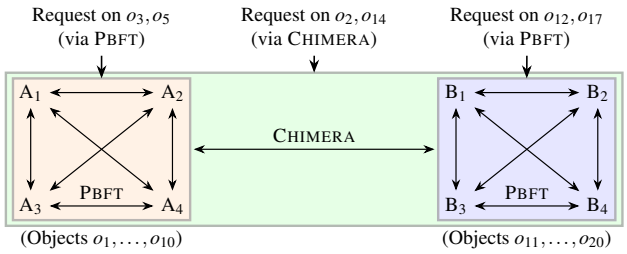

Figure 1: A *sharded* design in which two resilient blockchains each hold only a part of the data. Local decisions within a cluster are made via *traditional* PBFT *consensus*, whereas multi-shard transactions are processed via CHIMERA (proposed in this work).

shards that each can potentially operate mostly-independent on their data, while only requiring inter-shard coordination to process multi-shard transactions that affect data on several shards (see Figure 1).

The choice of protocol for such multi-shard transaction processing determines greatly the scalability benefits of sharding and the overhead costs incurred by sharding, however [1, 3, 4, 17, 35, 55]. In practice, existing proposals for sharding consensus-based systems have taken a general-purpose approach aiming at serving any workload. Unfortunately, such genericity comes at a cost, and existing proposals either have high coordination costs, incur high latencies, or have severe bottlenecks with multi-shard workloads.

In this work, we ask whether one can improve on the state-of-the-art proposals for sharded consensus-based resilient systems by *limiting* the type of workloads supported by the systems. In specific, we propose the following problem for further study:

> **Problem.** Can one *reduce* the cost of coordination in the design of sharded consensus-based systems by limiting the types of workloads supported?

In this paper, we give a preliminary *positive* answer for the above problem. In specific, we put forward the CHIMERA family of multi-shard transaction processing protocols that can process UTXO-transactions and uses properties of these transactions to *reduce* coordination to a minimum.

To be able to adapt to the needs of specific use-cases, we propose three variants of CHIMERA:

1. In Section 4, we propose Core-CHIMERA (CCHIMERA), a design specialized for processing *UTXO-like transactions*. CCHIMERA uses strict environmental assumptions on UTXO-transactions to its advantage to yield a *minimalistic* design that only requires a *single local consensus step* in affected shards, an absolute minimum. Furthermore, CCHIMERA requires only a single round of information sharing between shards. This information sharing can be implemented either via an all-to-all communication step (favoring latency over bandwidth usage) or via an all-to-one-to-all communication step (favoring bandwidth usage over latency).

   Even with this minimalistic design, CCHIMERA will operate *perfectly* with respect to all transactions proposed and approved by well-behaved clients: CCHIMERA is able to correctly execute all such transactions, even in the presence of transactions proposed by malicious clients (CCHIMERA may fail to process transactions originating from malicious clients, however).

To also support more general-purpose environments in which clients are malicious or can legitimately approve conflicting transactions, we propose Optimistic-CHIMERA and Resilient-CHIMERA, *two* generalizations of CCHIMERA that each deal with the strict environmental assumptions of CCHIMERA while preserving the minimalistic design of CCHIMERA:

2. In Section 5, we propose Optimistic-CHIMERA. In the design of Optimistic-CHIMERA (OCHIMERA), we assume that malicious behavior is rare and we optimize the normal-case operations. We do so by keeping the normal-case operations as minimalistic as possible by utilizing a *single* multi-shard consensus step to execute multi-shard transactions in the normal case.

   This multi-shard consensus step combines the local consensus steps of CCHIMERA and the information sharing steps of CCHIMERA into a single step. As with CCHIMERA, this step can either favor latency or bandwidth. When compared to CCHIMERA, the multi-shard consensus step does not require any additional coordination phases in the well-behaved optimistic case, while still being able to lift the environmental assumptions of CCHIMERA and lowering the latency of transaction processing in most cases. In doing so, OCHIMERA does require intricate coordination when recovering from attacks, however.

3. In Section 6, we propose Resilient-CHIMERA. In the design of Resilient-CHIMERA, we assume that malicious behavior is common and we add sufficient coordination to the normal-case operations of CCHIMERA to enable a simpler and localized recovery path, allowing RCHIMERA to operate in a general-purpose fault-tolerant environments without significant costs to recover from attacks.

In Section 7, we show that all three variants of CHIMERA provide strong ordering guarantees based on their usage of UTXO-transactions. Finally, in Section 8, we compare the three CHIMERA protocols and show their potential scalability and performance benefits over state-of-the-art general-purpose systems

2 Preliminaries

As permissioned blockchains already have much higher throughputs than permissionless blockchains, we will focus on permissioned blockchains in this paper.

First, we introduce the system model, the sharding model, the data model, the transaction model, and the terminology and notation used throughout this paper.

If S is a set of replicas, then $\mathcal{G}(S)$ denotes the non-faulty *good replicas* in S that always operate as intended, and $\mathcal{F}(S) = S \setminus \mathcal{G}(S)$ denotes the remaining replicas in S that are *faulty* and can act *Byzantine*, deviate from the intended operations, or even operate in coordinated malicious manners. We write $\mathbf{n}_S = |S|$, $\mathbf{g}_S = |\mathcal{G}(S)|$, and $\mathbf{f}_S = |S \setminus \mathcal{G}(S)| = \mathbf{n}_S - \mathbf{g}_S$ to denote the number of replicas in S, good replicas in S, and faulty replicas in S, respectively.

We assume that communication between replicas is *authenticated*: on receipt of a message m from replica $\mathtt{R} \in \mathfrak{R}$, one can determine that $\mathtt{R}$ did sent m if $\mathtt{R} \in \mathcal{G}(\mathfrak{R})$. Hence, faulty replicas are able to impersonate each other, but are not able to impersonate good replicas. To provide authenticated communication under practical assumptions, we can rely on cryptographic primitives such as digital signatures and threshold signatures [42, 56].

Let $\mathfrak{R}$ be a set of replicas. In a *sharded fault-tolerant system* over $\mathfrak{R}$, the replicas are partitioned into sets $\mathtt{shards}(\mathfrak{R}) = \{\mathcal{S}_0, \ldots, \mathcal{S}_{\mathbf{z}}\}$ such that the replicas in $\mathcal{S}_i$, $0 \leq i \leq \mathbf{z}$, operate as an independent Byzantine fault-tolerant system. As each $\mathcal{S}_i$ operates as an independent Byzantine fault-tolerant system, we require $\mathbf{n}_{\mathcal{S}_i} > 3\mathbf{f}_{\mathcal{S}_i}$, a minimal requirement to enable Byzantine fault-tolerance in an asynchronous environment [20, 21]. We assume that every shard $\mathcal{S} \in \mathtt{shards}(\mathfrak{R})$ has a unique identifier $\mathtt{id}(\mathcal{S})$.

Assumption 2.1. We assume *coordinating adversaries* that can, at will, choose and control any replica $\mathtt{R} \in \mathcal{S}$ in any shard $\mathcal{S} \in \mathtt{shards}(\mathfrak{R})$ in the sharded fault-tolerant system as long as, for each shard $\mathcal{S}' \in \mathtt{shards}(\mathfrak{R})$, the adversaries only control up to $\mathbf{f}_{\mathcal{S}'}$ replicas in $\mathcal{S}'$.

We use the *object-dataset model* in which data is modeled as a collection of *objects*. Each object o has a unique *identifier* $\mathtt{id}(o)$ and a unique *owner* $\mathtt{owner}(o)$.[1] In the following, we assume that all owners are *clients* of the system that manages these objects. The only operations that one can perform on an object are *construction* and *destruction*. An object cannot be recreated, as the attempted recreation of an object o will result in a new object o' with a distinct identifier ($\mathtt{id}(o) \neq \mathtt{id}(o')$).

Changes to object-dataset data are made via transactions requested by clients. We write $\langle \tau \rangle_c$ to denote a transaction τ requested by a client c. We assume that all transactions are *UTXO-like transactions*: a transaction τ first produces resources by destructing a set of *input objects* and then consumes these resources in the construction of a set of *output objects*. We do not rely on the exact rules regarding the production and consumption of resources, as they are highly application-specific. Given a transaction τ, we write $\mathtt{Inputs}(\tau)$ and $\mathtt{Outputs}(\tau)$ to denote the input objects and output objects of τ, respectively, and we write $\mathtt{Objects}(\tau) = \mathtt{Inputs}(\tau) \cup \mathtt{Outputs}(\tau)$.[2]

Assumption 2.2. Given a transaction τ, we assume that one can determine $\mathtt{Inputs}(\tau)$ and $\mathtt{Outputs}(\tau)$ a-priori.

To simplify presentation, we assume throughout this paper that every transactions has inputs. Hence, $|\mathtt{Inputs}(\tau)| \geq 1$. Owners of objects o can *express their approval* for transactions τ that have o as their input. To provide this functionality, we can rely on digital signatures [42].

Assumption 2.3. If an owner is well-behaved, then an expression of approval cannot be forged or provided by any other party.[3] Furthermore, a well-behaved owner of o will only express its approval for *a single* transaction τ with $o \in \mathtt{Inputs}(\tau)$, as only one transaction can consume the object o, and the owner will only do so after the construction of o.

Let o be an object. We assume that there is a well-defined function $\mathtt{shard}(o)$ that maps object o to the single shard $\mathcal{S} \in \mathtt{shards}(\mathfrak{R})$ that is responsible for maintaining o. Given a transaction τ, we write $\mathtt{shards}(\tau) = \{\mathtt{shard}(o) \mid o \in \mathtt{Objects}(\tau)\}$ to denote the shards that are affected by τ. Note that the shards $\mathtt{shards}(\tau)$ affected by transaction τ are all shards that hold either input or output objects of transaction τ. We say that τ is a *single-shard transaction* if $|\mathtt{shards}(\tau)| = 1$ and is a *multi-shard transaction* otherwise.

3 Multi-Shard Transaction Processing

Before we introduce CHIMERA, we put forward the correctness requirements we want to maintain in a multi-shard transaction system in which each shard is itself a set of replicas operated as a Byzantine fault-tolerant system. We say

[1] In this setting, the owner of an object typically is some client that can authenticate transactions involving that object. For example, if objects represent monetary tokens, then the owner can approve transfers of these tokens.

[2] UTXO-like transactions are mainly known for their usage in Bitcoin and other cryptocurrencies [49]. They form an abstract model that can easily be used for many other types of workloads. For example, a versioned file system can be represented via UTXO-like transactions: each object is a file (owned by some user), transactions represent one or more changes to a file, and each such changes turns a file into a object (typically owned by the same user, but ownership can be transfered). In such a versioned file system, new files can be created, existing files can be removed, files can be duplicated, and files can be merged, due to which the inputs and outputs of a transaction do not simply map a single file unto a newer version.

[3] Earlier, we assumed a unique owner that can approve transactions and prove object ownership in a unique and non-ambiguous way. This does not preclude shared ownership in which multiple participants own an object, however. In that case, we simply require that such a group of participants can approve transactions via their own agreement process to determine which transactions to support (e.g., via multiple signatures, via threshold signatures, or via other mechanisms).

that a shard $\mathcal{S}$ performs an action if every good replica in $\mathcal{G}(\mathcal{S})$ performs that action. Hence, any processing decision or execution step performed by $\mathcal{S}$ requires the usage of a *consensus protocol* [14, 16, 31, 45, 46] that coordinates the operations of individual replicas in the system, e.g., a Byzantine fault-tolerant system driven by PBFT [16], POE [33], or HOTSTUFF [61], or a crash fault-tolerant system driven by PAXOS [45]. As these systems are fully-replicated, each replica will eventually execute the same *sequence of transactions* and, hence, will observe the same evolution of the data held by the system. This *sequence of transactions* is often referred to as a ledger or a journal and is agreed upon via consensus:

Definition 3.1. A *consensus protocol* coordinate decision making among the replicas of a resilient cluster $\mathcal{S}$ by providing a reliable ordered replication of *decisions* (e.g., the decision to execute a given transaction next). To do so, consensus protocols provide the following guarantees:

1. If good replica $\mathrm{R} \in \mathcal{S}$ makes a ρ-th decision, then all good replicas $\mathrm{R}' \in \mathcal{S}$ will make a ρ-th decision (whenever communication becomes reliable).

2. If good replicas $\mathrm{R}, \mathrm{Q} \in \mathcal{S}$ make ρ-th decisions, then they make the same decisions.

3. Whenever a good replica learns that a decision D needs to be made, then it can force consensus on D.

Many definitions of consensus include a requirement of *non-triviality* instead of the last requirement in Definition 3.1. To simplify presentation, we focus on the usage of consensus for operating services that processes transactions requested by clients. In such services, each consensus decision represents a transaction requested by a client and non-triviality is provided by assuring that any client can get their requests processed. Next, we illustrate how.

Assume that consensus is provided via a primary-backup consensus protocol such as PBFT. These primary-backup protocols operate in *views* and within a single view a single replica, the *primary*, coordinates consensus decisions among all other replicas. To deal with primary failures, these primary-backup consensus protocols perform a *view-change* during which all preceding consensus decisions are secured, after which the system moves to the *next* view. This next view will be coordinated by *another* primary (typically the next replica assuming some ordering on replicas). This view-change mechanism is also used to force primaries to propose specific decisions: first, clients can send their requests D to all good replicas; next, all good replicas can suggest D to the primary; finally, either the primary successfully coordinates a consensus decision on D or all good replicas eventually decide that the primary failed to do so and initiate view-changes to replace the primary (until a view is reached in which a good primary proposes the suggested request D).

Let τ be a transaction processed by a sharded fault-tolerant system. Processing of τ does not imply execution: the transaction could be invalid (e.g., the owners of affected objects did not express their approval) or the transaction could have inputs that no longer exists.

We say that the system *commits* to transaction τ if the system decides to apply the modifications prescribed by τ, and we say that the system *aborts* τ if it decides to not do so (after processing it). Finally, we say that a transaction received by the system is *discarded* if it will never be processed (never results in a commit or abort).

Using this terminology, we put forward the following requirements for any sharded fault-tolerant system:

R1 *Validity*. The system must only commit or abort valid transactions and discard all invalid transactions. In specific, the system must only commit or abort transaction τ if, for every input object $o \in \texttt{Inputs}(\tau)$ with a well-behaved owner $\texttt{owner}(o)$, the owner $\texttt{owner}(o)$ approves the transaction.[4]

R2 *Shard-involvement*. The shard $\mathcal{S}$ only processes transaction τ if $\mathcal{S} \in \texttt{shards}(\tau)$.

R3 *Shard-applicability*. Let $D(\mathcal{S})$ be the dataset maintained by shard $\mathcal{S}$ at time t. The shards $\texttt{shards}(\tau)$ can only commit transaction τ at time t if τ consumes only objects that exist at time t. Hence, $\texttt{Inputs}(\tau) \subseteq \bigcup\{D(\mathcal{S}) \mid \mathcal{S} \in \texttt{shards}(\tau)\}$.

R4 *Cross-shard-consistency*. If a good replica R partaking in processing transaction τ concludes that transaction τ was committed (aborted), then all good replicas in all shards $\mathcal{S}' \in \texttt{shards}(\tau)$ will eventually reach the same conclusion as replica R.

R5 *Service*. If client c is well-behaved and wants to request a valid transaction τ, then the sharded system will eventually *process* $\langle\tau\rangle_c$. If τ is shard-applicable, then the sharded system will eventually *execute* $\langle\tau\rangle_c$.

R6 *Confirmation*. If the system processes $\langle\tau\rangle_c$ and c is well-behaved, then c will eventually learn whether τ is committed or aborted.

The *validity* of transactions is a *local requirement*: whether a transaction τ is valid can be determined by checking whether all owners of inputs of τ support that transaction. Typically, ownership is expressed via digital signatures, which can be verified deterministically by any replica in any shard independently. Hence, all replicas in all affected shards will make the same conclusion on whether τ is valid. Likewise, also shard-involvement is a *local requirement*, as individual shards

[4] Determining validity of a transaction can include application-level requirements that should hold in a transaction. If, for example, the objects represent monetary balances, then transactions that produce *more output* than they consume *input* can be considered invalid.

can determine whether they need to process a given transaction. In the same sense, shard-applicability and cross-shard-consistency are *global* requirements, as assuring these requirements requires coordination between the shards affected by a transaction.

In the above and throughout this paper, we will speak of transaction *processing* whenever we look at the steps the system takes after receiving a request (eventually leading to discarding the request when it is invalid, a commit decision, or an abort decision). We will speak of transaction *execution* to refer to transactions that finished processing with either a commit decision or an abort decision.

4 Core-CHIMERA: Simple Yet Efficient Transaction Processing

The core idea of CHIMERA is to minimize the coordination necessary for multi-shard ordering and execution of transactions. To do so, CHIMERA combines the semantics of transactions in the object-dataset model with the minimal coordination required to assure shard-applicability and cross-shard consistency. This combination results in the following high-level three-step approach towards processing any transaction τ:

1. *Local inputs*. First, every affected shard $\mathcal{S} \in \texttt{shards}(\tau)$ locally determines whether it has all inputs from $\mathcal{S}$ that are necessary to process τ.

2. *Cross-shard exchange*. Then, every affected shard $\mathcal{S}$ exchanges these inputs to all other shards in $\texttt{shards}(\tau)$, thereby pledging to use their local inputs when executing of τ.

3. *Decide outcome*. Finally, every affected shard $\mathcal{S}$ decides to commit τ if all affected shards were able to provide all local inputs necessary for execution of τ.

Next, we describe how these three high-level steps are incorporated by CHIMERA into normal consensus steps at each shards. Let shard $\mathcal{S} \in \texttt{shards}(\mathfrak{R})$ receive client request $\langle\tau\rangle_c$. The good replicas in $\mathcal{S}$ will first determine whether τ is valid and whether τ can be applicable with respect to those inputs $\texttt{Inputs}(\tau)$ maintained by shard $\mathcal{S}$.

If τ is not valid or $\mathcal{S} \notin \texttt{shards}(\tau)$, then the good replicas discard τ. Otherwise, if τ is valid and $\mathcal{S} \in \texttt{shards}(\tau)$, then the good replicas utilize *consensus* to force the shard $\mathcal{S}$ to propose a consensus decision in some consensus round ρ on the message $m(\mathcal{S},\tau)_\rho = (\langle\tau\rangle_c, I(\mathcal{S},\tau), D(\mathcal{S},\tau))$, in which $I(\mathcal{S},\tau) = \{o \in \texttt{Inputs}(\tau) \mid \mathcal{S} = \texttt{shard}(o)\}$ is the set of objects maintained by $\mathcal{S}$ that are input to τ and $D(\mathcal{S},\tau) \subseteq I(\mathcal{S},\tau)$ is the set of currently-available inputs at $\mathcal{S}$. Transaction τ can only be applicable with respect to those inputs $\texttt{Inputs}(\tau)$ maintained by shard $\mathcal{S}$ if $I(\mathcal{S},\tau) = D(\mathcal{S},\tau)$. Hence, only if $I(\mathcal{S},\tau) = D(\mathcal{S},\tau)$ will $\mathcal{S}$ pledge to use the local inputs $I(\mathcal{S},\tau)$ in the execution of τ.

We use *consensus* during the *local inputs* step as it provides an ordered agreement among seqeuences of transactions. This ordered agreement is necessary to acquire a consistent results among all replicas in a shard: all replicas of a shard need to process all transactions they process in the same order, as otherwise they cannot agree on which of the inputs of a transaction τ are available to τ in the presence of other transactions with the same inputs.

The acceptance of $m(\mathcal{S},\tau)_\rho$ in round ρ by all good replicas completes the *local inputs* step. Next, during processing of τ, the *cross-shard exchange* and *decide outcome* steps are performed. First, the *cross-shard exchange* step. In this step, $\mathcal{S}$ broadcasts $m(\mathcal{S},\tau)_\rho$ to all other shards in $\texttt{shards}(\tau)$. To assure that the broadcast arrives, we rely on a reliable primitive for *cross-shard exchange* that guarantees that only approved-upon values can be exchanged. Recently, such primitives have been formalized as cluster-sending [34, 36]:

Definition 4.1. Let $\mathcal{S}_1, \mathcal{S}_2$ be two shards. The *cluster-sending problem* is the problem of sending a value v from $\mathcal{S}_1$ to $\mathcal{S}_2$ such that:

1. all good replicas in $\mathcal{S}_2$ *receive* the value v;

2. all good replicas in $\mathcal{S}_1$ receive *confirmation* that the value v was received by all good replicas in $\mathcal{S}_2$; and

3. good replicas in $\mathcal{S}_2$ can only receive a value v if all good replicas in $\mathcal{S}_1$ *agreed* upon sending v.

After $\mathcal{S}$ broadcasts $m(\mathcal{S},\tau)_\rho$ to all other shards in $\texttt{shards}(\tau)$, the replicas in $\mathcal{S}$ wait until they receive messages $m(\mathcal{S}',\tau)_{\rho'} = (\langle\tau\rangle_c, I(\mathcal{S}',\tau), D(\mathcal{S}',\tau))$ from all other shards $\mathcal{S}' \in \texttt{shards}(\tau)$.

After cross-shard exchange comes the final *decide outcome* step. After $\mathcal{S}$ receives $m(\mathcal{S}',\tau)_{\rho'}$ from all shards $\mathcal{S}' \in \texttt{shards}(\tau)$, it decides to *commit* whenever $I(\mathcal{S}',\tau) = D(\mathcal{S}',\tau)$ for all $\mathcal{S}' \in \texttt{shards}(\tau)$. Otherwise, it decides *abort*. If $\mathcal{S}$ decides commit, then all good replicas in $\mathcal{S}$ destruct all objects in $D(\mathcal{S},\tau)$ and construct all objects $o \in \texttt{Outputs}(\tau)$ with $\mathcal{S} = \texttt{shard}(o)$. Finally, each good replica informs c of the outcome of execution. If c receives, from every shard $\mathcal{S}'' \in \texttt{shards}(\tau)$, identical outcomes from $\mathbf{g}_{\mathcal{S}''} - \mathbf{f}_{\mathcal{S}''}$ distinct replicas in $\mathcal{S}''$, then it considers τ to be successfully executed. In Figure 2, we sketched the working of CCHIMERA.

The *cross-shard exchange* step of CCHIMERA at $\mathcal{S}$ involves waiting for other shards $\mathcal{S}'$. Hence, there is the danger of deadlocks if the other shards $\mathcal{S}'$ never perform their cross-shard exchange steps. To assure that such situations do not lead to a deadlock, we employ two techniques.

1. *Internal propagation*. As shard $\mathcal{S}$ has decided *not* to discard τ, the transaction must be valid. Hence, other shards will not discard τ whenever they receive a request to

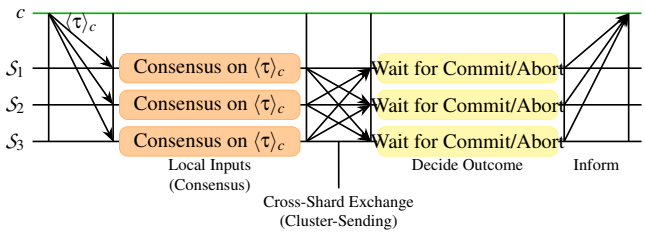

Figure 2: The message flow of CCHIMERA for a 3-shard client request $\langle\tau\rangle_c$ that is committed.

process τ. To deal with situations in which some shards $\mathcal{S}' \in \texttt{shards}(\tau)$ did not receive $\langle\tau\rangle_c$ (e.g., due to network failure or due to a faulty client that fails to send $\langle\tau\rangle_c$ to $\mathcal{S}$), we allow each shard to learn τ from any other shard. In specific, to assure that all shards $\mathcal{S}' \in \texttt{shards}(\tau)$ will receive a request to process τ, all shards $\mathcal{S}'$ will start consensus on $\langle\tau\rangle_c$ after receiving *cross-shard exchange* related to $\langle\tau\rangle_c$. As $\mathcal{S}$ uses a resilient primitive to provide cross-shard exchange, we can assure that all shards $\mathcal{S}' \in \texttt{shards}(\tau)$ will receive cross-shard exchange and eventually start processing τ.

2. *Concurrent resolution*. To deal with concurrent transactions that contend for the same objects, we allow each shard to accept and process transactions for different rounds concurrently. To assure that concurrent resolution does not lead to inconsistent state updates, each replica implements the following *first-pledge* and *ordered-commit* rules. Let τ be a transaction with $\mathcal{S} \in \texttt{shards}(\tau)$ and $\text{R} \in \mathcal{S}$. The *first-pledge* rule states that $\mathcal{S}$ pledges o, constructed in round ρ, to transaction τ only if τ is the first transaction proposed after round ρ with $o \in \texttt{Inputs}(\tau)$. The *ordered-commit* rule states that $\mathcal{S}$ can abort τ in any order, but will only commit τ that is accepted in round ρ after previous rounds finished execution.

The above first-pledge and ordered-commit rules do not need to be enforced or guaranteed, as they specify deterministic behavior for all good replicas. Next, we illustrate the usage of these rules.

Example 4.2. Consider two shards $\mathcal{S}_1$ and $\mathcal{S}_2$ affected by transactions τ_1 and τ_2 that each require objects o_1 and o_2 residing on shards $\mathcal{S}_1$ and $\mathcal{S}_2$, respectively. Now consider the case in which shard $\mathcal{S}_1$ first processes τ_1 and then τ_2, while shard $\mathcal{S}_2$ first processes τ_2 and then τ_1. In this case, shard $\mathcal{S}_1$ will pledge o_1 to τ_1 and shard $\mathcal{S}_2$ will pledge o_2 to τ_2. Hence, both τ_1 and τ_2 miss inputs and will fail to complete execution. As both transactions will abort, the order in which they abort does not matter.

In this situation, which will only happen if two transactions have the same inputs in violation of Assumption 2.3, will result in an abort for the two transactions τ_1 and τ_2. Transactions that have unique inputs (in line with Assumption 2.3), will always be able to be committed.

Abort decisions at shard $\mathcal{S}$ on a transaction τ can often be made without waiting for all shards $\mathcal{S}' \in \texttt{shards}(\tau)$: shard $\mathcal{S}$ can decide abort after it detects $I(\mathcal{S},\tau) \neq D(\mathcal{S},\tau)$ or after it receives the first message $(\langle\tau\rangle_c, I(\mathcal{S}'',\tau), D(\mathcal{S}'',\tau))$ with $I(\mathcal{S}'',\tau) \neq D(\mathcal{S}'',\tau)$, $\mathcal{S}'' \in \texttt{shards}(\tau)$. For efficiency, we allow $\mathcal{S}$ to abort in these cases.

Theorem 4.3. *If, for all shards $\mathcal{S}^*$, $\mathbf{g}_{\mathcal{S}^*} > 2\mathbf{f}_{\mathcal{S}^*}$, and Assumptions 2.1, 2.2, and 2.3 hold, then Core-*CHIMERA *satisfies Requirements R1–R6 with respect to all transactions that are not requested by malicious clients and do not involve objects with malicious owners.*

Proof. Let τ be a transaction. As good replicas in $\mathcal{S}$ discard τ if it is invalid or if $\mathcal{S} \notin \texttt{shards}(\tau)$, CCHIMERA provides *validity* and *shard-involvement*. Next, *shard-applicability* follow directly from the decide outcome step.

If a shard $\mathcal{S}$ commits or aborts transaction τ, then it must have completed the decide outcome and cross-shard exchange steps. Hence, all shards $\mathcal{S}' \in \texttt{shards}(\tau)$ must have exchanged the necessary information to $\mathcal{S}$. By relying on cluster-sending for cross-shard exchange, $\mathcal{S}'$ requires cooperation of all good replicas in $\mathcal{S}'$ to exchange the necessary information to $\mathcal{S}$. Hence, we have the guarantee that these good replicas will also perform cross-shard exchange to any other shard $\mathcal{S}'' \in \texttt{shards}(\tau)$. As such, every shard $\mathcal{S}'' \in \texttt{shards}(\tau)$ will receive the same information as $\mathcal{S}$, complete cross-shard exchange, and make the same decision during the decide outcome step, providing *cross-shard consistency*.

Due to internal propagation and concurrent resolution, every valid transaction τ will be processed by CCHIMERA as soon as it is send to any shard $\mathcal{S} \in \texttt{shards}(\tau)$. Hence, every shard in $\texttt{shards}(\tau)$ will perform the necessary steps to eventually inform the client. As all good replicas $\text{R} \in \mathcal{S}$, $\mathcal{S} \in \texttt{shards}(\tau)$, will inform the client of the outcome for τ, the majority of these inform-messages come from good replicas, enabling the client to reliably derive the true outcome. Hence, CCHIMERA provides *service* and *confirmation*. □

Notice that in the object-dataset model in which we operate, each object can be constructed once and destructed once. Hence, each object o can be part of at-most two committed transactions: the first of which will construct o as an output, and the second of which has o as an input and will consume and destruct o. This is independent of any other operations on other objects. As such these two transactions *cannot* happen concurrently. Consequently, we only have concurrent transactions on o if the owner $\texttt{owner}(o)$ expresses approval for several transactions that have o as an input. By Assumption 2.3, the owner $\texttt{owner}(o)$ must be malicious in that case. As such, transactions of well-behaved clients and owners will *never abort*.

In the design of CCHIMERA, we take *full* advantage of the above observation: CCHIMERA effectively *eliminates all coordination* when deciding the order in which individual replicas and shards process both single-shard and multi-shard transactions. Indeed, the order in which replicas will process a multi-shard transaction is decided *before* any coordination and, hence, independent of whichever order the other shards choose to process transactions. Only the outcome of processing transactions are influenced by cross-shard exchange steps. Due to this, CCHIMERA allows all involved shards to process a transaction *independently* with a single consensus step: all communication between shards in CCHIMERA is dedicated to exchange execution state *after* individual shards reach consensus. We can do so as any *aborts*, which could have been prevented with additional coordination, are always due to malicious behavior by clients and owners of objects. Due to this, CCHIMERA will not undo any pledges of objects to the execution of any transactions. This implies that objects that are involved in malicious transactions can get lost for future usage, while not affecting any transactions of well-behaved clients.

Finally, we remark that CCHIMERA depends on underlying consensus and cluster-sending protocols. The level to which CCHIMERA can deal with asynchronous behavior depends on the particular choices of these protocols.

5 Optimistic-CHIMERA: Robust Transaction Processing

In the previous section, we introduced CCHIMERA, a minimalistic multi-shard transaction processing protocol that relies on properties of UTXO-like transactions to maximize performance. Although the design of CCHIMERA is simple yet effective, we see two shortcomings that limits its use. First, CCHIMERA operates under Assumption 2.3, the assumption that any issues arising from concurrent transactions is due to malicious behavior of clients. As such, CCHIMERA chooses to lock out objects affected by such malicious behavior for any future usage. Second, CCHIMERA requires consecutive consensus and cluster-sending steps, which increases its transaction processing latencies. Next, we investigate how to deal with these weaknesses of CCHIMERA *without giving up* on the minimalistic nature of CCHIMERA.

To do so, we propose Optimistic-CHIMERA (OCHIMERA), which is optimized for the *optimistic* case in which we have no concurrent transactions, while providing a recovery path that can recover from concurrent transactions without locking out objects (and without requiring Assumption 2.3). At the core of OCHIMERA is assuring that any issues due to malicious behavior, e.g., concurrent transactions, are *detected* in such a way that individual replicas can recover. At the same time, we want to minimize transaction processing latencies. To bridge between these two objectives, we integrate detection and cross-shard coordination within a single consensus round that runs at each affected shard.

OCHIMERA does not rely on underlying consensus and cluster-sending protocols. For the design of OCHIMERA, we assume *asynchronous communication*: messages can get lost, arrive with arbitrary delays, and in arbitrary order. Consequently, it is impossible to distinguish between, on the one hand, a replica that is malicious and does not send out messages, and, on the other hand, a replica that does send out proposals that get lost in the network. It is well-known that in such an environment, consensus cannot be provided [25, 27]. As such, OCHIMERA is designed to operate in an asynchronous environment in which it will *never cause* data inconsistency and only guarantees *progress* (service and confirmation) eventually when communication is reliable for a sufficiently-long period of time. This is the same model of partial asynchronous communication as used by PBFT.

Assume consensus decision are made with a PBFT-like primary-backup consensus protocol. Let $\langle\tau\rangle_c$ be a multi-shard transaction, let $\mathcal{S} \in \texttt{shards}(\tau)$ be an affected shard with primary $\mathcal{P}(\mathcal{S})$, and let $m(\mathcal{S},\tau)_{v,\rho} = (\langle\tau\rangle_c, I(\mathcal{S},\tau), D(\mathcal{S},\tau))$ be the round-ρ proposal of $\mathcal{P}(\mathcal{S})$ of view v of $\mathcal{S}$. To enable detection of concurrent transactions, OCHIMERA modifies the consensus-steps of the underlying consensus protocol by applying the following high-level idea:

> A replica $\mathrm{R} \in \mathcal{S}$, $\mathcal{S} \in \texttt{shards}(\tau)$, only accepts proposal $m(\mathcal{S},\tau)_{v,\rho}$ for transaction τ if it gets confirmation that replicas in each other shard $\mathcal{S}' \in \texttt{shards}(\tau)$ are also accepting proposals for τ. Otherwise, replica R detects failure.

To simplify presentation, we will use a traditional design that uses all-to-all communication between all replicas in all affected shards akin to the design of PBFT [16]. Traditionally, PBFT has a *commit* phase that is distinct from the commit decision made during multi-shard transaction processing. To disambiguate the commit phase of PBFT and the commit decision made during multi-shard transaction processing, we have renamed the commit phase of PBFT into the *finalize* phase. To minimize inter-shard communication (at the cost of latency) one can also utilize threshold signatures to implement all-to-one-to-all communication akin to the design of HOTSTUFF [61] to carry over local prepare and finalize certificates between shards via a few constant-sized messages.[5]

Next, we illustrate how to integrate the above idea in the three-phase design of PBFT, thereby turning PBFT into a multi-shard aware consensus protocol:

[5] Such a design will require a single replica responsible for receiving all messages, aggregating them into a single certificate, and broadcasting this message back. To deal with failures of this replica, we can simply choose a primary (whose failures are already dealt with by the normal recovery mechanisms that we shall describe). Alternatively, one can use the approach taken by SBFT [28] and choose any other replica as the aggregator (thereby offloading the primary), but this will require additional recovery mechanisms to deal with failures of that replica.

1. *Global preprepare*. Primary $\mathcal{P}(\mathcal{S})$ must send $m(\mathcal{S},\tau)_{v,\rho}$ to all replicas $\mathrm{R}' \in \mathcal{S}'$, $\mathcal{S}' \in \mathtt{shards}(\tau)$. Replica $\mathrm{R} \in \mathcal{S}$ only finishes the global preprepare phase after it receives a *global preprepare certificate* consisting of a set $M = \{m(\mathcal{S}'',\tau)_{v'',\rho''} \mid \mathcal{S}'' \in \mathtt{shards}(\tau)\}$ of preprepare messages from all primaries of shards affected by τ.

2. *Global prepare*. After $\mathrm{R} \in \mathcal{S}$, $\mathcal{S} \in \mathtt{shards}(\tau)$, finishes the global preprepare phase, it sends prepare messages for M to all other replicas in $\mathrm{R}' \in \mathcal{S}'$, $\mathcal{S}' \in \mathtt{shards}(\tau)$. Replica $\mathrm{R} \in \mathcal{S}$ only finishes the global prepare phase for M after, for every shard $\mathcal{S}' \in \mathtt{shards}(\tau)$, it receives a *local prepare certificate* consisting of a set $P(\mathcal{S}')$ of prepare messages for M from $\mathbf{g}_{\mathcal{S}'}$ distinct replicas in $\mathcal{S}'$. We call the set $\{P(\mathcal{S}'') \mid \mathcal{S}'' \in \mathtt{shards}(\tau)\}$ a *global prepare certificate*.

3. *Global finalize*. After replica $\mathrm{R} \in \mathcal{S}$, $\mathcal{S} \in \mathtt{shards}(\tau)$, finishes the global prepare phase, it sends finalize messages for M to all other replicas in $\mathrm{R}' \in \mathcal{S}'$, $\mathcal{S}' \in \mathtt{shards}(\tau)$. Replica $\mathrm{R} \in \mathcal{S}$ only finishes the global finalize phase for M after, for every shard $\mathcal{S}' \in \mathtt{shards}(\tau)$, it receives a *local finalize certificate* consisting of a set $C(\mathcal{S}')$ of finalize messages for M from $\mathbf{g}_{\mathcal{S}'}$ distinct replicas in $\mathcal{S}'$. We call the set $\{P(\mathcal{S}'') \mid \mathcal{S}'' \in \mathtt{shards}(\tau)\}$ a *global finalize certificate*.

The above three-phase *global*-PBFT protocol already takes care of the *local input* and *cross-shard exchange* steps. Indeed, a replica $\mathrm{R} \in \mathcal{S}$ that finishes the global finalize phase has accepted global preprepare certificate M, which contains all information of other shards to proceed with processing. At the same time, R also has confirmation that M is prepared by a majority of all good replicas in each shard $\mathcal{S}' \in \mathtt{shards}(\tau)$ (which will eventually be followed by execution of τ within $\mathcal{S}'$). With these ingredients in place, only the *decide outcome* step remains.

The decide outcome step at shard $\mathcal{S}$ is entirely determined by the global preprepare certificate M. Shard $\mathcal{S}$ decides to *commit* whenever $I(\mathcal{S}',\tau) = D(\mathcal{S}',\tau)$ for all $(\langle\tau\rangle_c, I(\mathcal{S}',\tau), D(\mathcal{S}',\tau)) \in M$. Otherwise, it decides *abort*. If $\mathcal{S}$ decides commit, then all good replicas in $\mathcal{S}$ destruct all objects in $D(\mathcal{S},\tau)$ and construct all objects $o \in \mathtt{Outputs}(\tau)$ with $\mathcal{S} = \mathtt{shard}(o)$. Finally, each good replica informs c of the outcome of execution. If c receives, from every shard $\mathcal{S}' \in \mathtt{shards}(\tau)$, identical outcomes from $\mathbf{g}_{\mathcal{S}'} - \mathbf{f}_{\mathcal{S}'}$ distinct replicas in $\mathcal{S}'$, then it considers τ to be successfully executed. In Figure 3, we sketched the working of OCHIMERA.

We note that the multi-shard aware consensus protocol OCHIMERA is a *multi-shard consensus protocol* that aims to make a single consensus decision among all replicas in all shards affected by a multi-shard transaction. OCHIMERA is not the only multi-shard consensus protocol recently proposed (e.g., [3, 4]). What sets OCHIMERA apart is how it guarantees correctness *in all environments*, which is determined by how OCHIMERA deals with *non-optimistic cases* in

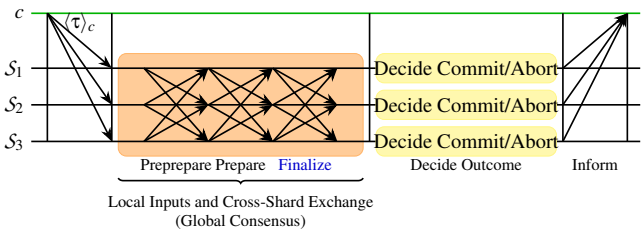

Figure 3: The message flow of OCHIMERA for a 3-shard client request $\langle\tau\rangle_c$ that is committed.

which failure is detected and recovery is necessary. We will detail recovery next. As a first step, we illustrate the ways in which the normal-case of OCHIMERA can fail (e.g., due to malicious behavior of clients, failing replicas, or unreliable communication).

Example 5.1. Consider a transaction τ proposed by client c and affecting shard $\mathcal{S} \in \mathtt{shards}(\tau)$. First, we consider the case in which $\mathcal{P}(\mathcal{S})$ is malicious and tries to set up a coordinated attack. To have maximum control over the steps of OCHIMERA, the primary sends the message $m(\mathcal{S},\tau)_{v,\rho}$ to only $\mathbf{g}_{\mathcal{S}''} - \mathbf{f}_{\mathcal{S}''}$ good replicas in each shard $\mathcal{S}'' \in \mathtt{shards}(\tau)$. By doing so, $\mathcal{P}(\mathcal{S})$ can coordinate the faulty replicas in each shard to assure failure of any phase at any replica $\mathrm{R}' \in \mathcal{S}'$, $\mathcal{S}' \in \tau$:

1. To prevent R' from finishing the global preprepare phase (and start the global prepare phase) for an M with $m(\mathcal{S}',\tau)_{v',\rho'} \in M$, $\mathcal{P}(\mathcal{S})$ simply does not send $m(\mathcal{S},\tau)_{v,\rho}$ to R'.

2. To prevent R' from finishing the global prepare phase (and start the global finalize phase) for M, $\mathcal{P}(\mathcal{S})$ instructs the faulty replicas in $\mathcal{F}(\mathcal{S})$ to not send prepare messages for M to R'. Hence, R' will receive at-most $\mathbf{g}_{\mathcal{S}} - \mathbf{f}_{\mathcal{S}}$ prepare messages for M from replicas in $\mathcal{S}$, assuring that it will not receive a local prepare certificate $P(\mathcal{S})$ and will not finish the global prepare phase for M.

3. Likewise, to prevent R' from finishing the global finalize phase (and start execution) for M, $\mathcal{P}(\mathcal{S})$ instructs the faulty replicas in $\mathcal{F}(\mathcal{S})$ to not send finalize messages to R'. Hence, R' will receive at-most $\mathbf{g}_{\mathcal{S}} - \mathbf{f}_{\mathcal{S}}$ finalize messages for M from replicas in $\mathcal{S}$, assuring that it will not receive a local finalize certificate $C(\mathcal{S})$ and will not finish the global finalize phase for M.

None of the above attacks can be attributed to faulty behavior of $\mathcal{P}(\mathcal{S})$ as unreliable communication can result in the same outcomes for R'. Furthermore, even if communication is reliable and $\mathcal{P}(\mathcal{S})$ is good, replica R' can see the same outcomes due to malicious behavior of the client or of primaries of other shards in $\mathtt{shards}(\tau)$:

1. The client c can be malicious and not send τ to $\mathcal{S}$. At the same time, all other primaries $\mathcal{P}(\mathcal{S}'')$ of shards $\mathcal{S}'' \in$

shards(τ) can be malicious and not send anything to $\mathcal{S}$ either. In this case, $\mathcal{P}(\mathcal{S})$ will never be able to send any message $m(\mathcal{S},\tau)_{v,\rho}$ to R′, as no replica in $\mathcal{S}$ is aware of τ.

2. If any primary $\mathcal{P}(\mathcal{S}'')$ of $\mathcal{S}'' \in$ shards(τ) is malicious, then it can prevent some replicas in $\mathcal{S}$ from starting the global prepare phase, thereby preventing these replicas to send prepare messages to R′. If $\mathcal{P}(\mathcal{S}'')$ prevents sufficient replicas in $\mathcal{S}$ from starting the global prepare phase, R′ will be unable to finish the global prepare phase.

3. Likewise, any malicious primary $\mathcal{P}(\mathcal{S}'')$ of $\mathcal{S}'' \in$ shards(τ) can prevent replicas in $\mathcal{S}$ from starting the global finalize phase, thereby assuring that R′ will be unable to finish the global finalize phase.

To deal with malicious behavior, OCHIMERA needs a robust recovery mechanism. Indeed, the main difference of the multi-shard consensus of OCHIMERA and the single-shard consensus PBFT is that OCHIMERA will use a single primary *per* shard whereas PBFT only has a single primary. This difference affects the capability of individual replicas to detect the root cause of disruptions of the normal-case operations (as several primaries could be the root cause of such disruptions). As such, we cannot simply build the robust recovery mechanism on top of traditional view-change approaches: these traditional view-change approaches require that one can identify a single source of failure (when communication is reliable), namely the current primary. To remedy this, the recovery mechanisms of OCHIMERA has components that perform *local view-change* and that perform *global state recovery*.

Next, we will detail the working of the recovery mechanisms of OCHIMERA. To simplify presentation, we will focus on the recovery of a single transaction. The techniques presented are straightforward to generalize to any history of zero-or-more transactions. The pseudo-code for the recovery protocol can be found in Figure 4. Next, we describe the working of this recovery protocol in detail.

Let R $\in \mathcal{S}$ be a replica that determines that it cannot finish a round ρ of view v. First, R determines whether it already has a *guarantee* on which transaction it has to process in round ρ. This is the case when the following conditions are met: R finished the global prepare phase for M with $m(\mathcal{S},\tau)_{v,\rho} \in M$ and has received a local finalize certificate $C(\mathcal{S}'')$ for M from some shard $\mathcal{S}'' \in$ shards(τ). In this case, R can simply request all missing local finalize certificates directly, as $C(\mathcal{S}'')$ can be used to prove to any involved replica R′ $\in \mathcal{S}'$, $\mathcal{S}' \in$ shards(τ), that R′ also needs to finalize M. To request such missing finalize certificates of $\mathcal{S}'$, replica R sends out VCGlobalSCR messages to all replicas in $\mathcal{S}'$ (Line 7 of Figure 4). Any replica R′ that receives such a VCGlobalSCR message can use the information in that message to reach the global finalize phase for M and, hence, provide R with the requested finalize messages (Line 11 of Figure 4).

1: **event** R $\in \mathcal{S}$ is unable to finish round ρ of view v **do**
2: **if** R finished in round ρ the global prepare phase for M,
but is unable to finish the global finalize phase **then**
3: Let P be the global prepare certificate of R for M.
4: **if** R has a local finalize certificate $C(\mathcal{S}'')$ for M **then**
5: **for** $\mathcal{S}' \in$ shards(τ) **do**
6: **if** R did not yet receive a local finalize certificate $C(\mathcal{S}')$ **then**
7: Broadcast $\langle$VCGlobalSCR : $M, P, C(\mathcal{S}'')\rangle$ to all replicas in $\mathcal{S}'$.
8: **else** Detect the need for local state recovery of round ρ of view v (Figure 5).
9: **else** Detect the need for local state recovery of round ρ of view v (Figure 5).
10: (Eventually repeat this event if R remains unable to finish round ρ.)

11: **event** R′ $\in \mathcal{S}'$ receives message $\langle$VCGlobalSCR : $M, P, C(\mathcal{S}'')\rangle$ from R $\in \mathcal{S}$ **do**
12: **if** R′ did not reach the global finalize phase for M **then**
13: Use M, P, and $C(\mathcal{S}'')$ to reach the global finalize phase for M.
14: **else** Send a finalize message for M to R.

Figure 4: The view-change *global short-cut recovery path* that determines whether R already has the assurance that the current transaction will be finalized. If this is the case, then R requests only the missing information to proceed with execution. Otherwise, R requires at-least local recovery (Figure 5).

If R does not have a *guarantee* itself on which transaction it has to process in round ρ, then it needs to determine whether any other replica (either in its own shard or in any other shard) has already received and acted upon such a guarantee. To initiate such local and global state recovery, R simply detects the current view as faulty. To do so, R broadcasts a VCRecoveryRQ message to all other replicas in $\mathcal{S}$ that contains all information R collected on round ρ in view v (Line 4 of Figure 5). Other replicas Q $\in \mathcal{S}$ that already have *guarantees* for round ρ can help R by providing all missing information (Line 6 of Figure 5). On receipt of this information, R can proceed with the round (Line 7 of Figure 5). If no replicas can provide the missing information, then eventually all good replicas will detect the need for local recovery, this either by themselves (Line 1 of Figure 5) or after receiving VCRecoveryRQ messages of at-least $\mathbf{f}_{\mathcal{S}}+1$ distinct replicas in $\mathcal{S}$, of which at-least a single replica must be good (Line 10 of Figure 5).

Finally, if a replica R receives $\mathbf{g}_{\mathcal{S}}$ VCRecoveryRQ messages, then it has the guarantee that at least $\mathbf{g}_{\mathcal{S}} - \mathbf{f}_{\mathcal{S}} \geq \mathbf{f}_{\mathcal{S}}+1$ of these messages come from good replicas in $\mathcal{S}$. Hence, due to Line 10 of Figure 5, all $\mathbf{g}_{\mathcal{S}}$ good replicas in $\mathcal{S}$ will send VCRecoveryRQ, and, when communication is reliable, also receive these messages. Consequently, at this point, R can start the new view by electing a new primary and awaiting the NewView proposal of this new primary (Line 12 of Figure 5). If R is the new primary, then it starts the new view by proposing a NewView. As other shards *could* have already made final decisions depending on local prepare or finalize certificates of $\mathcal{S}$ for round ρ, we need to assure that such certificates are not invalidated. To figure out whether such final decisions have been made, the new primary will query other shards $\mathcal{S}'$ for their state whenever the NewView message contains global preprepare certificates for transactions τ, $\mathcal{S}' \in$ shards(τ), but not a local finalize certificate to *guarantee* execution of τ

1: **event** R $\in \mathcal{S}$ detects the need for local state recovery of round ρ of view v **do**
2: Let M be any latest global preprepare certificate accepted for round ρ by R.
3: Let S be M and any prepare and finalize certificates for M collected by R.
4: Broadcast $\langle \texttt{VCRecoveryRQ} : v, \rho, S \rangle$.

5: **event** Q $\in \mathcal{S}$ receives messages $\langle \texttt{VCRecoveryRQ} : v, \rho, S \rangle$ of R $\in \mathcal{S}$ and Q has
1. started the global prepare phase for M with $m(\mathcal{S}, \tau)_{w,\rho} \in M$;
2. a global prepare certificate for M;
3. a local finalize certificate $C(\mathcal{S}'')$ for M

do
6: Send $\langle \texttt{VCLocalSCR} : M, P, C(\mathcal{S}'') \rangle$ to R $\in \mathcal{S}$.

7: **event** R $\in \mathcal{S}$ receives message $\langle \texttt{VCLocalSCR} : M, P, C(\mathcal{S}'') \rangle$ from Q $\in \mathcal{S}$ **do**
8: **if** R did not reach the global finalize phase for M **then**
9: Use M, P, and C to reach the global finalize phase for M.

10: **event** R $\in \mathcal{S}$ receives messages $\langle \texttt{VCRecoveryRQ} : v_i, \rho, S_i \rangle$, $1 \leq i \leq \mathbf{f}_{\mathcal{S}} + 1$, from $\mathbf{f}_{\mathcal{S}} + 1$ distinct replicas in $\mathcal{S}$ **do**
11: R detects the need for local state recovery of round ρ of view $\min\{v_i \mid 1 \leq i \leq \mathbf{f}_{\mathcal{S}} + 1\}$.

12: **event** R $\in \mathcal{S}$ receives messages $\langle \texttt{VCRecoveryRQ} : v, \rho, S_i \rangle$, $1 \leq i \leq \mathbf{g}_{\mathcal{S}}$, from distinct replicas in $\mathcal{S}$ **do**
13: **if** $\texttt{id}(\text{R}) \neq (v+1) \bmod \mathbf{n}_{\mathcal{S}}$ **then**
14: (R awaits the NewView message of the new primary, Line 14 of Figure 6.)
15: **else**
16: Broadcast $\langle \texttt{NewView} : \langle \texttt{VCRecoveryRQ} : v, \rho, S_i \rangle \mid 1 \leq i \leq \mathbf{g}_{\mathcal{S}} \rangle$ to all replicas in $\mathcal{S}$.
17: **if** there exists a S_i that contains global preprepare certificate M, but no S_j contains a local finalize certificate for M **then**
18: R initiates global state recovery of round ρ (Line 1 of Figure 6).

Figure 5: The view-change *local short-cut recovery path* that determines whether some Q can provide R with the assurance that the current transaction will be finalized. If this is the case, then R only needs this assurance, otherwise $\mathcal{S}$ requires a new view (Figure 6).

(Line 17 of Figure 5).

The new-view process has three stages. First, the new primary P proposes the new-view via a NewView message (Line 12 of Figure 5). If necessary, the new primary P also requests the relevant global state from any relevant shard (Line 1 of Figure 6). The replicas in other shards will respond to this request with their local state (Line 9 of Figure 6). The new primary collects these responses and sends them to all replicas in $\mathcal{S}$ via a NewViewGlobal message. Then, after P sends the NewView message to R $\in \mathcal{S}$, R determines whether the NewView message contains sufficient information to recover round ρ (Line 15 of Figure 6), contains sufficient information to wait for any relevant global state (Line 17 of Figure 6), or to determine that the new primary must propose for round ρ (Line 19 of Figure 6). If R determines it needs to wait for any relevant global state, then R will wait for this state to arrive via a NewViewGlobal message. Based on the received global state, R determines to recover round ρ (Line 21 of Figure 6), or determines that the new primary must propose for round ρ (Line 24 of Figure 6).

Next, we will prove the correctness of the view-change of OCHIMERA. First, using a standard quorum argument, we prove that in a single round of a single view of $\mathcal{S}$, only a single global preprepare message affecting $\mathcal{S}$ can get finalized by any other affected shards:

1: **event** P $\in \mathcal{S}$ initiates global state recovery of round ρ using $\langle \texttt{NewView} : V \rangle$ **do**
2: Let T be the transactions with global preprepare certificates for round ρ of $\mathcal{S}$ in view V.
3: Let S be the shards affected by transactions in T.
4: Broadcast $\langle \texttt{VCGlobalStateRQ} : v, \rho, V \rangle$ to all replicas in $\mathcal{S}' \in S$.
5: **for** $\mathcal{S}' \in S$ **do**
6: Wait for VCGlobalStateRQ messages for V from $\mathbf{g}_{\mathcal{S}'}$ distinct replicas in $\mathcal{S}'$.
7: Let $W(\mathcal{S}')$ be the set of received VCGLOBALSTATERQ messages.
8: Broadcast $\langle \texttt{NewViewGlobal} : V, \{W(\mathcal{S}') \mid \mathcal{S}' \in S\} \rangle$ to all replicas in $\mathcal{S}$.

9: **event** R$' \in \mathcal{S}'$ receives message $\langle \texttt{VCGlobalStateRQ} : v, \rho, V \rangle$ from P $\in \mathcal{S}$ **do**
10: **if** R$'$ has a global preprepare certificate M with $m(\mathcal{S}, \tau)_{w,\rho} \in M$ and reached the global finalize phase for M **then**
11: Let P be the global prepare certificate for M.
12: Send $\langle \texttt{VCGlobalStateR} : v, \rho, V, M, P \rangle$ to P.
13: **else** Send $\langle \texttt{VCGlobalStateR} : v, \rho, V \rangle$ to P.

14: **event** R $\in \mathcal{S}$ receives valid $\langle \texttt{NewView} : V \rangle$ message from replica P **do**
15: **if** there exists a $\langle \texttt{VCRecoveryRQ} : v_i, \rho, S_i \rangle \in V$ that contains
1. a global preprepare certificate M with $m(\mathcal{S}, \tau)_{w,\rho} \in M$;
2. a global prepare certificate P for M; and
3. a local finalize certificate $C(\mathcal{S}'')$ for M

then
16: Use M, P, and C to reach the global finalize phase for M.
17: **else if** there exists a $\langle \texttt{VCRecoveryRQ} : v_i, \rho, S_i \rangle \in V$ that contains
1. a global preprepare certificate M; and
2. no $\langle \texttt{VCRecoveryRQ} : v_j, \rho, S_j \rangle \in V$ contains a local finalize certificate for M

then
18: R detects the need for global state recovery of round ρ (Line 20 of Figure 6).
19: **else** (P must propose for round ρ.)

20: **event** R $\in \mathcal{S}$ receives valid $\langle \texttt{NewViewGlobal} : V, W \rangle$ from P $\in \mathcal{S}$ **do**
21: **if** any message in W is of the form $\langle \texttt{VCGlobalStateR} : v, \rho, V, M, P \rangle$ **then**
22: Select $\langle \texttt{VCGlobalStateR} : v, \rho, V, M, P \rangle \in W$ with latest view w, $m(\mathcal{S}, \tau)_{w,\rho} \in M$.
23: Use M and P to reach the global finalize phase for M.
24: **else** (P must propose for round ρ.)

Figure 6: The view-change *new-view recovery path* that recovers the state of the previous view based on a NewView proposal of the new primary. As part of the new-view recovery path, the new primary can construct a global new-view that contains the necessary information from other shards to reconstruct the local state.

Lemma 5.1. *Let τ_1 and τ_2 be transactions with $\mathcal{S} \in (\texttt{shards}(\tau_1) \cap \texttt{shards}(\tau_2))$. If $\mathbf{g}_{\mathcal{S}} > 2\mathbf{f}_{\mathcal{S}}$ and there exists shards $\mathcal{S}_i \in \texttt{shards}(\tau_i)$, $i \in \{1,2\}$, such that good replicas $\text{R}_i \in \mathcal{G}(\mathcal{S}_i)$ reached the global finalize phase for global preprepare certificate M_i with $m(\mathcal{S}, \tau_i)_{v,\rho} \in M_i$, then $\tau_1 = \tau_2$.*

Proof. We prove this property using contradiction. We assume $\tau_1 \neq \tau_2$. Let $P_i(\mathcal{S})$ be the local prepare certificate provided by $\mathcal{S}$ for M_i and used by R_i to reach the global finalize phase, let $S_i \subseteq \mathcal{S}$ be the $\mathbf{g}_{\mathcal{S}}$ replicas in $\mathcal{S}$ that provided the prepare messages in $P_i(\mathcal{S})$, and let $T_i = S_i \setminus \mathcal{F}(\mathcal{S})$ be the good replicas in S_i. By construction, we have $|T_i| \geq \mathbf{g}_{\mathcal{S}} - \mathbf{f}_{\mathcal{S}}$. As all replicas in $T_1 \cup T_2$ are good, they will only send out a single prepare message per round ρ of view v. Hence, if $\tau_1 \neq \tau_2$, then $T_1 \cap T_2 = \emptyset$, and we must have $2(\mathbf{g}_{\mathcal{S}} - \mathbf{f}_{\mathcal{S}}) \leq |T_1 \cup T_2|$. As all replicas in $T_1 \cup T_2$ are good, we also have $|T_1 \cup T_2| \leq \mathbf{g}_{\mathcal{S}}$. Hence, $2(\mathbf{g}_{\mathcal{S}} - \mathbf{f}_{\mathcal{S}}) \leq \mathbf{g}_{\mathcal{S}}$, which simplifies to $\mathbf{g}_{\mathcal{S}} \leq 2\mathbf{f}_{\mathcal{S}}$, a contradiction. Hence, we conclude $\tau_1 = \tau_2$. □

Next, we use Lemma 5.1 to prove that any global prepre-

pare certificate that *could* have been accepted by any good affected replica is preserved by OCHIMERA:

Proposition 5.1. *Let τ be a transaction and $m(\mathcal{S},\tau)_{v,\rho}$ be a preprepare message. If, for all shards $\mathcal{S}^*$, $\mathbf{g}_{\mathcal{S}^*} > 2\mathbf{f}_{\mathcal{S}^*}$, and there exists a shard $\mathcal{S}' \in \texttt{shards}(\tau)$ such that $\mathbf{g}_{\mathcal{S}'} - \mathbf{f}_{\mathcal{S}'}$ good replicas in $\mathcal{S}'$ reached the global finalize phase for M with $m(\mathcal{S},\tau)_{v,\rho} \in M$, then every successful future view of $\mathcal{S}$ will recover M and assure that the good replicas in $\mathcal{S}$ reach the finalize phase for M.*

Proof. Let $v^* \leq v$ be the first view in which a global prepare certificate M^* with $m(\mathcal{S},\tau^*)_{v^*,\rho} \in M^*$ satisfied the premise of this proposition. Using induction on the number of views after the first view v^*, we will prove the following two properties on M^*:

1. every good replica that participates in view w, $v^* < w$, will recover M^* upon entering view w and reach the finalize phase for M^*; and

2. no replica will be able to construct a local prepare certificate of $\mathcal{S}$ for any global preprepare certificate $M^\dagger \neq M^*$ with $m(\mathcal{S},\tau^\dagger)_{w,\rho} \in M^\dagger$, $v^* < w$.

The base case is view v^*+1. Let $S' \subseteq \mathcal{G}(\mathcal{S}')$ be the set of $\mathbf{g}_{\mathcal{S}'} - \mathbf{f}_{\mathcal{S}'}$ good replicas in $\mathcal{S}'$ that reached the global finalize phase for M^*. Each replica $\mathrm{R}' \in S'$ has a local prepare certificate $P(\mathcal{S})$ consisting of $\mathbf{g}_{\mathcal{S}}$ prepare messages for M^* provided by replicas in $\mathcal{S}$. We write $S(\mathrm{R}') \subseteq \mathcal{G}(\mathcal{S})$ to denote the at-least $\mathbf{g}_{\mathcal{S}} - \mathbf{f}_{\mathcal{S}}$ good replicas in $\mathcal{S}$ that provided such a prepare message to R'.

Consider any valid new-view proposal $\langle \texttt{NewView} : V \rangle$ for view v^*+1. If the conditions of Line 15 of Figure 6 hold for global preprepare certificate $M^\dagger$ with $m(\mathcal{S},\tau^\ddagger)_{w,\rho} \in M^\ddagger$, then we recover $M^\ddagger$. As there is a local finalize certificate for $M^\ddagger$ in this case, the premise of this proposition holds on $M^\ddagger$. As v^* is the first view in which the premise of this proposition hold, we can use Lemma 5.1 to conclude that $w = v^*$, $M^\ddagger = M^*$, and, hence, that the base case holds if the conditions of Line 15 of Figure 6 hold. Next, we assume that the conditions of Line 15 of Figure 6 do not hold, in which case M^* can only be recovered via global state recovery. As the first step in global state recovery is proving that the condition of Line 17 of Figure 6 holds. Let $T \subseteq \mathcal{G}(\mathcal{S})$ be the set of at-least $\mathbf{g}_{\mathcal{S}} - \mathbf{f}_{\mathcal{S}}$ good replicas in $\mathcal{S}$ whose `VCRecoveryRQ` message is in V and let $\mathrm{R}' \in S'$. We have $|S(\mathrm{R}')| \geq \mathbf{g}_{\mathcal{S}} - \mathbf{f}_{\mathcal{S}}$ and $|T| \geq \mathbf{g}_{\mathcal{S}} - \mathbf{f}_{\mathcal{S}}$. Hence, by a standard quorum argument, we conclude $S(\mathrm{R}') \cap T \neq \emptyset$. Let $\mathrm{Q} \in (S(\mathrm{R}') \cap T)$. As Q is good and send prepare messages for M^*, it must have reached the global prepare phase for M^*. Consequently, the condition of Line 17 of Figure 6 holds and to complete the proof, we only need to prove that any well-formed `NewViewGlobal` message will recover M^*.

Let $\langle \texttt{NewViewGlobal} : V, W \rangle$ be any valid global new-view proposal for view v^*+1. As Q reached the global prepare phase for M^*, any valid global new-view proposal must include messages from $\mathcal{S}' \in \texttt{shards}(\tau)$. Let $U' \subseteq \mathcal{S}'$ be the replicas in $\mathcal{S}'$ of whom messages `VCGlobalStateR` are included in W. Let $V' = U' \setminus \mathcal{F}(\mathcal{S}')$. We have $|S'| \geq \mathbf{g}_{\mathcal{S}'} - \mathbf{f}_{\mathcal{S}'}$ and $|V'| \geq \mathbf{g}_{\mathcal{S}'} - \mathbf{f}_{\mathcal{S}'}$. Hence, by a standard quorum argument, we conclude $S' \cap V' \neq \emptyset'$. Let $\mathrm{Q}' \in (S' \cap V')$. As Q' reached the global finalize phase for M^*, it will meet the conditions of Line 23 of Figure 6 and provide both M^* and a global prepare certificate for M^*. Let $M^\ddagger$ be any other global preprepare certificate in W accompanied by a global prepare certificate. Due to Line 22 of Figure 6, the global preprepare certificate for the newest view of $\mathcal{S}$ will be recovered. As v^* is the newest view of $\mathcal{S}$, $M^\ddagger$ will only prevent recovery of M^* if it is also a global preprepare certificate for view v^* of $\mathcal{S}$. In this case, Lemma 5.1 guarantees that $M^\ddagger = M^*$. Hence, any replica R will recover M^* upon receiving $\langle \texttt{NewViewGlobal} : V, W \rangle$.

Now assume that the induction hypothesis holds for all views j, $v^* < j \leq i$. We will prove that the induction hypothesis holds for view $i+1$. Consider any valid new-view proposal $\langle \texttt{NewView} : V \rangle$ for view $i+1$ and let $M^\ddagger$ with $m(\mathcal{S},\tau^\ddagger)_{w,\rho} \in M^\ddagger$ be any global preprepare certificate that is recovered due to the new-view proposal $\langle \texttt{NewView} : V \rangle$. Hence, $M^\ddagger$ is recovered via either Line 16 of Figure 6 or Line 23 of Figure 6. In both cases, there must exist a global prepare certificate P for $M^\ddagger$. As $\langle \texttt{NewView} : V \rangle$ is valid, we must have $w \leq i$. Hence, we can apply the second property of the induction hypothesis to conclude that $w \leq v^*$. If $w = v^*$, then we can use Lemma 5.1 to conclude that $M^\ddagger = M^*$. Hence, to complete the proof, we must show that $w = v^*$. First, the case in which $M^\ddagger$ is recovered via Line 16 of Figure 6. Due to the existence of a global finalize certificate C for $M^\ddagger$, $M^\ddagger$ satisfies the premise of this proposition. By assumption, v^* is the first view for which the premise of this proposition holds. Hence, $w \geq v^*$, in which case we conclude $M^\ddagger = M^*$. Last, the case in which $M^\ddagger$ is recovered via Line 23 of Figure 6. In this case, $M^\ddagger$ is recovered via some message $\langle \texttt{NewViewGlobal} : V, W \rangle$. Analogous to the proof for the base case, V will contain a message `VCRecoveryRQ` from some replica $\mathrm{Q} \in S(\mathrm{R}')$. Due to Line 2 of Figure 5, Q will provide information on M^*. Consequently, a prepare certificate for M^* will be obtained via global state recovery, and we also conclude $M^\ddagger = M^*$. □

Lemma 5.1 and Proposition 5.1 assure that no transaction that could-be-finalized by any replica will ever get lost by the system. Next, we bootstrap these technical properties to prove that all good replicas can always recover such could-be-finalized transactions.

Proposition 5.2. *Let τ be a transaction and $m(\mathcal{S},\tau)_{v,\rho}$ be a preprepare message. If, for all shards $\mathcal{S}^*$, $\mathbf{g}_{\mathcal{S}^*} > 2\mathbf{f}_{\mathcal{S}^*}$, and there exists a shard $\mathcal{S}' \in \texttt{shards}(\tau)$ such that $\mathbf{g}_{\mathcal{S}'} - \mathbf{f}_{\mathcal{S}'}$ good replicas in $\mathcal{S}'$ reached the global finalize phase for M with $m(\mathcal{S},\tau)_{v,\rho} \in M$, then every good replica in $\mathcal{S}$ will accept M whenever communication becomes reliable.*

Proof. Let R $\in \mathcal{S}$ be a good replica that is unable to accept M. At some point, communication becomes reliable, after which R will eventually trigger Line 1 of Figure 4. We have the following cases:

1. If R meets the conditions of Line 4 of Figure 4, then R has a local finalize certificate $C(\mathcal{S}'')$, $\mathcal{S}'' \in \texttt{shards}(\tau)$. This local finalize certificate certifies that at least $\mathbf{g}_{\mathcal{S}''} - \mathbf{f}_{\mathcal{S}''}$ good replicas in $\mathcal{S}''$ finished the global prepare phase for M. Hence, the conditions for Proposition 5.1 are met for M and, hence, any shard in $\texttt{shards}(\tau)$ will maintain or recover M. Replica R can use $C(\mathcal{S}'')$ to prove this situation to other replicas, forcing them to finalize M, and provide any finalize messages R is missing (Line 11 of Figure 4).

2. If R does not meet the conditions of Line 4 of Figure 4, but some other good replica Q $\in \mathcal{S}$ does, then Q can provide all missing information to R (Line 6 of Figure 5). Next, R uses this information (Line 7 of Figure 5), after which it meets the conditions of Line 4 of Figure 4.

3. Otherwise, if the above two cases do not hold, then all $\mathbf{g}_{\mathcal{S}}$ good replicas in $\mathcal{S}$ are unable to finish the finalize phase. Hence, they perform a view-change. Due to Proposition 5.1, this view-change will succeed and put every replica in $\mathcal{S}$ into the finalize phase for M. As all good replicas in $\mathcal{S}$ are in the finalize phase, each good replica in $\mathcal{S}$ will be able to make a local finalize certificate $C(\mathcal{S})$ for M, after which they meet the conditions of Line 4 of Figure 4. □

Finally, we use Proposition 5.2 to prove *cross-shard-consistency*.

Theorem 5.2. *Optimistic-*CHIMERA *maintains cross-shard consistency.*

Proof. Assume a single good replica R $\in \mathcal{S}$ executes a transaction τ (by committing or aborting). Hence, it accepted some global preprepare certificate M with $m(\mathcal{S},\tau)_{v,\rho} \in M$. Consequently, R has local finalize certificates $C(\mathcal{S}')$ for M of every $\mathcal{S}' \in \texttt{shards}(\tau)$. Hence, at least $\mathbf{g}_{\mathcal{S}'} - \mathbf{f}_{\mathcal{S}'}$ good replicas in $\mathcal{S}'$ reached the global finalize phase for M, and we can apply Proposition 5.2 to conclude that any good replica R$'' \in \mathcal{S}''$, $\mathcal{S}'' \in \texttt{shards}(\tau)$ will accept M. As R$''$ bases its execution decision for τ on the same global prepare certificate M as R, they will both make the same decision, completing the proof. □

Due to the similarity between OCHIMERA and CCHIMERA, one can use the details of Theorem 4.3 to prove that OCHIMERA provides *validity*, *shard-involvement*, and *shard-applicability*. Via Theorem 5.2, we proved *cross-shard-consistency*. We cannot prove *service* and *confirmation*, however. The reason for this is simple: even though OCHIMERA can detect and recover from accidental faulty behavior and accidental concurrent transactions, OCHIMERA is not designed to gracefully handle targeted attacks: OCHIMERA is optimistic in the sense that it is optimized for the situation in which faulty behavior (including concurrent transactions that contend for the same objects) is rare. Still, in all cases, OCHIMERA maintains cross-shard consistency, however. Moreover, in the optimistic case, progress is guaranteed:

Proposition 5.3. *If, for all shards $\mathcal{S}^*$, $\mathbf{g}_{\mathcal{S}^*} > 2\mathbf{f}_{\mathcal{S}^*}$, and Assumptions 2.1, 2.2, and 2.3 hold, then Optimistic-*CHIMERA *satisfies Requirements R1–R6 in the optimistic case (whenever communication is reliable, shards have good primaries, and no concurrent transactions exist).*

OCHIMERA cannot defend against denial-of-service attacks targeted at blocking individual replicas and shards from participating. Unfortunately, no existing consensus protocol is able to deal with such attacks. Furthermore, as is the case for other multi-shard consensus protocols, coordinated attempts can prevent OCHIMERA from making progress in periods when the optimistic assumption does not hold. At the core of such attacks is the ability for malicious clients and malicious primaries to corrupt the operations of shards coordinated by good primaries, as already shown in Example 5.1. Due to Theorem 5.2, such attacks will *never* affect consistency in OCHIMERA, however.

To further reduce the impact of targeted attacks, one can make primary election non-deterministic, e.g., by using shard-specific distributed coins to elect new primaries in individual shards [11, 13]. Finally, we remark that we have presented OCHIMERA with a per-round checkpoint and recovery method. In this simplified design, the recovery path only has to recover at-most a single round. Our approach can easily be generalized to a more typical multi-round checkpoint and recovery method, however. Furthermore, we believe that the way in which OCHIMERA extends PBFT can easily be generalized to other consensus protocols, e.g., POE [33] and HOTSTUFF [61].

6 Resilient-CHIMERA: Transaction Processing Under Attack

In the previous section, we introduced OCHIMERA, a general-purpose minimalistic and efficient multi-shard transaction processing protocol. OCHIMERA is designed with the assumption that malicious behavior is rare, due to which it can minimize coordination in the normal-case while requiring intricate coordination when recovering from attacks. As an alternative to the optimistic approach of OCHIMERA, we can apply a *pessimistic* approach to CCHIMERA to gracefully recover from concurrent transactions that is geared towards minimizing the influence of malicious behavior altogether (without requiring Assumption 2.3). Next, we explore such a pessimistic design via *resilient-*CHIMERA (RCHIMERA).

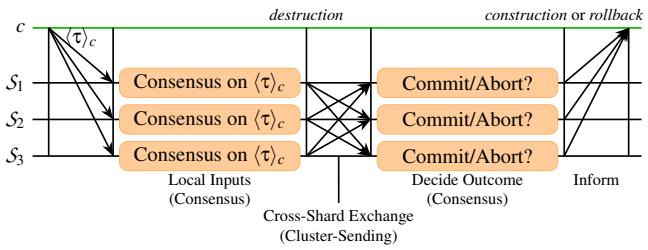

Figure 7: The message flow of RCHIMERA for a 3-shard client request $\langle\tau\rangle_c$ that is committed.

The design of RCHIMERA builds upon the design of CCHIMERA by adding additional coordination to the cross-shard exchange and decide outcome steps. As in CCHIMERA, the acceptance of $m(\mathcal{S},\tau)_\rho$ in round ρ by all good replicas completes the *local inputs* step. Before cross-shard exchange, the replicas in $\mathcal{S}$ destruct the objects in $D(\mathcal{S},\tau)$, thereby fully pledging these objects to τ until the commit or abort decision. Then, $\mathcal{S}$ performs cross-shard exchange by broadcasting $m(\mathcal{S},\tau)_\rho$ to all other shards in $\texttt{shards}(\tau)$, while the replicas in $\mathcal{S}$ wait until they receive messages $m(\mathcal{S}',\tau)_{\rho'}$ from all other shards $\mathcal{S}' \in \texttt{shards}(\tau)$.

After cross-shard exchange comes the final *decide outcome* step. After $\mathcal{S}$ receives $m(\mathcal{S}',\tau)_{\rho'}$ from all shards $\mathcal{S}' \in \texttt{shards}(\tau)$, the replicas force a *second consensus step* that determines the round ρ^* at which $\mathcal{S}$ decides *commit* (whenever $I(\mathcal{S}',\tau) = D(\mathcal{S}',\tau)$ for all $\mathcal{S}' \in \texttt{shards}(\tau)$) or *abort*. Note that the second consensus step in a shard $\mathcal{S}$ starts after all good replicas in $\mathcal{S}$ have received all required information to decide either commit or abort. Hence, the second consensus step is only used to *order* the commit or abort decision among all other decisions made in that shard. If $\mathcal{S}$ uses a primary-backup consensus protocol, then this second consensus step can be initiated by the primary without further input. If the primary is malicious, the good replicas need to force the second consensus step, however (e.g., by replacing the primary).

If $\mathcal{S}$ decides commit, then, in round ρ^*, all good replicas in $\mathcal{S}$ construct all objects $o \in \texttt{Outputs}(\tau)$ with $\mathcal{S} = \texttt{shard}(o)$. If $\mathcal{S}$ decides abort, then, in round ρ^*, all good replicas in $\mathcal{S}$ reconstruct all objects in $D(\mathcal{S},\tau)$ (rollback). Finally, each good replica informs c of the outcome of execution. If c receives, from every shard $\mathcal{S}' \in \texttt{shards}(\tau)$, identical outcomes from $\mathbf{g}_{\mathcal{S}'} - \mathbf{f}_{\mathcal{S}'}$ distinct replicas in $\mathcal{S}'$, then it considers τ to be successfully executed. In Figure 7, we sketched the working of RCHIMERA.

We notice that processing a multi-shard transaction via RCHIMERA requires *two* consensus steps per shard. In some cases, we can eliminate the second step, however. First, if τ is a multi-shard transaction with $\mathcal{S} \in \texttt{shards}(\tau)$ and the replicas in $\mathcal{S}$ accept $(\langle\tau\rangle_c, I(\mathcal{S},\tau), D(\mathcal{S},\tau))$ with $I(\mathcal{S},\tau) \neq D(\mathcal{S},\tau)$, then the replicas can immediately abort whenever they accept $(\langle\tau\rangle_c, I(\mathcal{S},\tau), D(\mathcal{S},\tau))$. Second, if τ is a single-shard transaction with $\texttt{shards}(\tau) = \{\mathcal{S}\}$, then the replicas in $\mathcal{S}$ can immediately decide commit or abort whenever they accept $(\langle\tau\rangle_c, I(\mathcal{S},\tau), D(\mathcal{S},\tau))$. Hence, in both cases, processing of τ at $\mathcal{S}$ only requires a single consensus step at $\mathcal{S}$. Next, we prove the correctness of RCHIMERA:

Theorem 6.1. *If, for all shards $\mathcal{S}^*$, $\mathbf{g}_{\mathcal{S}^*} > 2\mathbf{f}_{\mathcal{S}^*}$, and Assumptions 2.1, 2.2, and 2.3 hold, then Resilient-*CHIMERA *satisfies Requirements R1–R6.*

Proof. Let τ be a transaction. As good replicas in $\mathcal{S}$ discard τ if it is invalid or if $\mathcal{S} \notin \texttt{shards}(\tau)$, RCHIMERA provides *validity* and *shard-involvement*. Next, *shard-applicability* follow directly from the decide outcome step.

If a shard $\mathcal{S}$ commits or aborts transaction τ, then it must have completed the decide outcome and cross-shard exchange steps. Hence, all shards $\mathcal{S}' \in \texttt{shards}(\tau)$ must have exchanged the necessary information to $\mathcal{S}$. By relying on cluster-sending for cross-shard exchange, $\mathcal{S}'$ requires cooperation of all good replicas in $\mathcal{S}'$ to exchange the necessary information to $\mathcal{S}$. Hence, we have the guarantee that these good replicas will also perform cross-shard exchange to any other shard $\mathcal{S}'' \in \texttt{shards}(\tau)$. Consequently, every shard $\mathcal{S}'' \in \texttt{shards}(\tau)$ will receive the same information as $\mathcal{S}$, complete cross-shard exchange, and make the same decision during the decide outcome step, providing *cross-shard consistency*.

A client can force service on a transaction τ by choosing a shard $\mathcal{S} \in \texttt{shards}(\tau)$ and sending τ to all good replicas in $\mathcal{G}(\mathcal{S})$. By doing so, the normal mechanisms of consensus can be used by the good replicas in $\mathcal{G}(\mathcal{S})$ to force acceptance on τ in $\mathcal{S}$ and, hence, bootstrapping acceptance on τ in all shards $\mathcal{S}' \in \texttt{shards}(\tau)$. Due to cross-shard consistency, every shard in $\texttt{shards}(\tau)$ will perform the necessary steps to eventually inform the client. As all good replicas $\mathrm{R} \in \mathcal{S}$, $\mathcal{S} \in \texttt{shards}(\tau)$, will inform the client of the outcome for τ, the majority of these inform-messages come from good replicas, enabling the client to reliably derive the true outcome. Hence, RCHIMERA provides *service* and *confirmation*. □

As with CCHIMERA, RCHIMERA depends on underlying consensus and cluster-sending protocols and the level to which RCHIMERA can deal with asynchronous behavior depends on the particular choices of these protocols.

The step from RCHIMERA to general-purpose workloads is small: one can replace the data model used by RCHIMERA by a general-purpose data model using two-phase locking instead of object construction and destruction to guarantee isolated execution. To guarantee the same level of isolation for transaction execution as RCHIMERA guarantees *without* using the UTXO-like transactions (see Section 7), such an approach will require a costly locking scheme for all pieces of data affected by a transaction (including any objects that would be considered outputs by RCHIMERA). Such a two-phase locking design will end up very similar to that of CHAINSPACE or BYSHARD.

7 The Ordering of Transactions in CHIMERA

Having introduced the three variants of CHIMERA in Sections 4, 5, and 6, we will now analyze the ordering guarantees provided by CHIMERA. We further refer to Section 8 for a detailed comparison of the three variants of CHIMERA. Here, we will show that CHIMERA provides serializable execution [6, 9].

The data model utilized by CCHIMERA, OCHIMERA, and RCHIMERA guarantees that any object o can only be involved in at-most *two* committed transactions: one that *constructs* o and another one that *destructs* o. Assume the existence of such transactions τ_1 and τ_2 with $o \in \texttt{Outputs}(\tau_1)$ and $o \in \texttt{Inputs}(\tau_2)$. Due to *cross-shard-consistency* (Requirement R4), the shard $\texttt{shard}(o)$ will have to execute both τ_1 and τ_2. From these observations, we can derive a serializable order on all committed transactions:

Theorem 7.1. *A sharded fault-tolerant system that uses the object-dataset data model, processes UTXO-like transactions, and satisfies Requirements R1-R5 commits transactions in a serializable order.*

Proof. Assume the existence of transactions τ_1 and τ_2 with $o \in \texttt{Outputs}(\tau_1)$ and $o \in \texttt{Inputs}(\tau_2)$. Due to *shard-applicability* (Requirement R3), shard $\texttt{shard}(o)$ will execute τ_1 strictly before τ_2. Now consider the relation

$$\prec := \{(\tau, \tau') \mid (\text{the system committed to } \tau \text{ and } \tau') \wedge (\texttt{Outputs}(\tau) \cap \texttt{Inputs}(\tau') \neq \emptyset)\}.$$

Obviously, we have $\prec(\tau_1, \tau_2)$. To prove that all committed transactions are executed in a *serializable* ordering, we first prove the following:

> If we interpret transactions as nodes and $\prec$ as an edge relation, then the resulting graph is *acyclic*.

The proof is by contradiction. Let G be the graph-interpretation of $\prec$. We assume that graph G is cyclic. Hence, there exists transactions $\tau_0, \ldots, \tau_{m-1}$ such that $\prec(\tau_i, \tau_{i+1})$, $0 \leq i < m-1$, and $\prec(\tau_{m-1}, \tau_0)$. By the definition of $\prec$, we can choose objects o_i, $0 \leq i < m$, with $o_i \in (\texttt{Outputs}(\tau_i) \cap \texttt{Inputs}(\tau_{(i+1) \bmod m}))$. Due to *cross-shard-consistency* (Requirement R4), the shard $\texttt{shard}(o_i)$, $0 \leq i < m$, executed transactions τ_i and $\tau_{(i+1) \bmod m}$. Consider o_i, $0 \leq i < m$, and let t_i be the time at which shard $\texttt{shard}(o_i)$ executed τ_i and constructed o_i. Due to *shard-applicability* (Requirement R3), we know that shard $\texttt{shard}(o_i)$ executed $\tau_{(i+1) \bmod m}$ strictly after t_i. Moreover, also shard $\texttt{shard}(o_{(i+1) \bmod m})$ must have executed $\tau_{(i+1) \bmod m}$ strictly after t_i and we derive $t_i < t_{(i+1) \bmod m}$. Hence, we must have $t_0 < t_1 < \cdots < t_{m-1} < t_0$, a contradiction. Consequently, G must be acyclic.

To derive a serializable execution order for all committed transactions, we simply construct a directed acyclic graph in which transactions are nodes and $\prec$ is the edge relation. Next, we *topologically sort* the graph to derive the searched-for ordering. □

We notice that CHIMERA only provides serializability for *committed* transactions: concurrent transactions that contend for the same objects will always be aborted and, hence, will not be executed and will not affect the serializable order of execution of transactions. This flexibility is crucial in the design of CCHIMERA: as a consequence of the minimal coordination in CCHIMERA, the order in which individual replicas make abort decisions is not coordinated, due to which replicas can process abort decisions in distinct non-serializable orders, this even replicas within a singular shard. It is this flexibility in dealing with aborted transactions that allows CCHIMERA to operate with minimal coordination while still providing strong isolation for all committed transactions. Both OCHIMERA and RCHIMERA incorporate stronger coordination, due to which all commit and abort decisions within a shard are made in a consistent order among all replicas in that shard. Still, both OCHIMERA and RCHIMERA do not enforce a distinct non-serializable order in which abort decisions are processed across shards, as such enforcement would incur additional coordination costs.

8 Analysis of the Three CHIMERA Variants

In the previous sections, we proposed three variants of CHIMERA and showed their correctness. Next, we analyze the benefits and costs of the three CHIMERA multi-shard transaction processing protocols, compare them with state-of-the-art multi-shard transaction processing protocols, and evaluate the impact of malicious behavior on CHIMERA. A summary of this analysis can be found in Figure 8.

We did not detail the exact message complexity of the three CHIMERA protocols. For CCHIMERA and RCHIMERA we measure the complexity in the number of consensus steps and cluster-sending steps they require. Implementation-wise, one can choose to either implement these steps with all-to-all communication (as in PBFT) or with all-to-one-to-all communication (as in HOTSTUFF) to optimize for either low latency or low bandwidth usage. Similarly, we can implement also OCHIMERA with all-to-one-to-all communication instead of all-to-all communication.

Remark 8.1. A common technique to improve the transaction throughput of consensus-based systems is by processing a batch holding many transactions per consensus decision. To simplify presentation, we have chosen to present the three CHIMERA protocols without such batching. Both CCHIMERA and RCHIMERA can easily be generalized to process transactions in batches: at a per-shard level, they use standard consensus protocols that operate independently of other shards. Hence, instead of one transaction per consensus decision, both can include a batch of transactions in their consensus

decisions (after which they perform the steps related to the transactions in that batch in order).

For OCHIMERA, blocks of transactions are more challenging as OCHIMERA uses a single multi-shard consensus step that includes all replicas of all shards affected by a transaction. Still, OCHIMERA can be generalized to process batches of transactions that affect the same set of shards. Such a generalization requires additional machinery, however: multi-shard batches can lead to several shards proposing distinct batches that include the same transaction, however. It is possible to deal with such issues with existing techniques (e.g., by assigning each transaction to a single batch-proposal shard based on the digest of that transaction) [31].

8.1 A Comparison of CHIMERA Variants

First, Figure 8 provides a high-level comparison of the costs of each of the three CHIMERA protocols to process a single transaction τ that affects $s = |\texttt{shards}(\tau)|$ distinct shards. For the normal-case behavior, we compare the complexity in the number of *sequential communication phases* (which, in the idle case, are the main determinant for client latencies), the number of *consensus steps* per shard and *cross-shard exchange* steps between shards (which together determine the bandwidth costs and put an upper bound on throughput). As one can see, all three protocols have a low number of *phases*, due to which all three can provide low latencies toward clients. On the one hand, we see that OCHIMERA only performs a PBFT-like multi-shard consensus, which takes *three* consecutive phases of communication. On the other hand, CCHIMERA performs both a local consensus step and a cross-shard exchange step. Assuming the local consensus step is performed via PBFT, this results in *four* consecutive phases of communication (three *local communication phases* for PBFT, one for the cross-shard exchange step). Hence, in environments in which cross-shard communication has low latency, OCHIMERA will be able to provide lower latencies than both CCHIMERA and RCHIMERA, as its optimistic design eliminates one phase of communication (at the cost of requiring cross-shard communication in every phase).

Next, we compare how the three protocols deal with malicious behavior by clients and by replicas. If no clients behave malicious, then all transactions will *commit*. In all three protocols, malicious behavior by clients can lead to the existence of concurrent transactions that affect the same object. Upon detection of such concurrent transactions, all three protocols will *abort*. The consequences of such an abort are different in the three protocols.

In CCHIMERA, objects affected by aborted transactions remain pledged and cannot be reused. In practice, this loss of objects can provide an incentive for clients to not behave malicious, but does limit the usability of CCHIMERA in non-incentivized environments. Both OCHIMERA and RCHIMERA deal with concurrent transactions by aborting them via the normal-case of the protocol. The three CHIMERA protocols are resilient against malicious replicas: only malicious primaries can affect the normal-case operations of these protocols. If the behavior of a primary is disrupting the normal-case operations, then in CCHIMERA and RCHIMERA such behavior is dealt with by the recovery mechanisms of the underlying consensus protocol (e.g., in PBFT such disruptions will eventually lead to a view-change whenever communication is reliable), whereas OCHIMERA will utilize the view-change recovery mechanisms outlined in Section 5. In both CCHIMERA and RCHIMERA, dealing with a malicious primary in a shard can be done completely in isolation of all other shards. In OCHIMERA, which is optimized with the assumption that failures are rare, the failure of a primary while processing a transaction τ can lead to view-changes in all shards affected by τ.

In conclusion, we see that the three CHIMERA variants each make their own tradeoff between *normal-case costs* and ability to deal with faulty behavior (by both clients and other replicas), with RCHIMERA being robust against any attack at the cost of 2 consensus decisions per transaction per involved shard.

8.2 Comparison With the State-of-the-Art

Several recent papers have proposed specialized systems that combine sharding with consensus-based resilient systems. Examples include systems such as AHL [17], BYSHARD [35], CAPER [3], CHAINSPACE [1], RINGBFT [55], and SHARPER [4], which all use sharding for data management and transaction processing. Next, we compare the design of CHIMERA in detail with AHL [17], CHAINSPACE [1], RINGBFT [55], and SHARPER [4], and briefly look at BYSHARD [35] and CAPER [3].

AHL [17]. AHL uses a *centralized* commit protocol to order all multi-shard transactions. In specific, AHL [17] uses a reference committee that leads a *centralized two-phase commit protocol* (Centralized 2PC) [30, 53] that is implemented via consensus steps and cluster-sending. Typically, the reference committee is responsible for coordinating *all* multi-shard transactions. Furthermore, AHL uses non-blocking locks to provide transaction isolation due to which valid transactions can be aborted, whereas in CHIMERA only faulty transactions (e.g., by malicious clients) are aborted. By using Centralized 2PC, AHL eliminates any all-to-all communication between shards affected by a transaction in favor of one-to-all communication between the reference committee and the affected shards. Due to this, AHL takes five consecutive consensus rounds, more than twice the number of rounds required by the costliest CHIMERA variants. As reported in the original evaluation of AHL [17, Section 7.3], the central role of the reference committee in coordinating *all* multi-shard transactions will quickly become a bottleneck for performance when

Protocol	Principle Technique	Phases[a] (Cross-Shard)	Consensus Steps. *Total*	*Sequential*	*Type*[b]	Cross-Shard Communication[c]	Transaction Abort Causes	Transaction Concurrency and Ordering	Failure Recovery (method and when)
CCHIMERA	Independent Consensus UTXO Data Model	4 (1)	s	1	LC	1 (CS, A2A)	Faulty Only	Data Model Pledges (Incentive[d])	Local Recovery Local Primary Failure
OCHIMERA	Multi-Shard Consensus UTXO Data Model	3 (3)	s	1	MS	3 (MS, A2A)	Faulty Only	Data Model Aborts	Local and Global Recovery Any Primary Failure
RCHIMERA	Distributed Commit UTXO Data Model	7 (1)	$2s$	2	LC	1 (CS, A2A)	Faulty Only	Data Model Aborts	Local Recovery Local Primary Failure
AHL [17]	Reference Committee Non-Blocking Locks	19 (4)	$2s+2$	5	LC	4 (CS, O2A)	Failed Locks	Reference Committee Locks & Aborts	Local Recovery Local Primary Failure
CHAINSPACE [1]	Distributed Commit locking Locks	11 (2)	$2s$	3	LC	2 (CS, A2A)	Failed Locks	Distributed Commit Locks & Aborts	Local Recovery Local Primary Failure
RINGBFT [55]	Linear Commit Blocking Locks	$8s-5$ ($2s-2$)	$2s-1$	$2s-1$	LC	$2s-2$ (CS, O2O)	Invalid Only	Linear Commit Blocking Locks	Local Recovery Local Primary Failure
SHARPER [4]	Multi-Shard Consensus Shard-Wide Blocking Locks	3 (3)	s	1	MS	3 (MS, A2A)	Failed Locks (Shard-Wide)	Multi-Shard Consensus Shard-Wide Locks & Aborts	Global Recovery Any Primary Failure, Concurrency

[a]Total number of consecutive communication phases. For protocols that use a local consensus protocol, we count three consecutive phases per consensus step (e.g., using PBFT), and we count a single phase per cluster-sending step.

[b]We write *LC* to local per-shard consensus steps that do *not involve* cross-shard communication and we use *MS* to indicate multi-shard consensus steps that require cross-shard communication for each phase of communication performed by the multi-shard consensus protocol.

[c]We write *CS* to indicate *cluster-sending* and *MS* to indicate *multi-shard consensus*; and we write *A2A* to denote all-to-all communication, O2A to denote one-to-all or all-to-one communication, and O2O to denote one-to-one communication between involved shards.

[d]In CCHIMERA, objects affected by aborted transactions remain pledged and cannot be reused. In practice, this loss of objects can provide an incentive for clients to not behave malicious.

Figure 8: Comparison of the three CHIMERA protocols for processing a transaction that affects s shards. We compare the normal-case complexity. the mechanism used to deal with concurrent transactions (due to malicious clients), and the mechanisms used to provide failure recovery.

processing workloads heavy in multi-shard transactions (even if none of these transactions are concurrent), while AHL shows excellent performance when processing single-shard transactions [35].

CHAINSPACE [1]. CHAINSPACE uses a *distributed two-phase commit protocol* (Distributed 2PC) [30, 53], that is implemented via consensus steps and cluster-sending, to order all multi-shard transactions. Furthermore, similar to AHL, CHAINSPACE uses non-blocking locks to provide transaction isolation due to which valid transactions can be aborted. The operations of this commit protocol are similar to the design of RCHIMERA, except that CHAINSPACE does not take advantage of any specific properties of the data model (e.g., to provide isolation). A further minor difference between CHAINSPACE and RCHIMERA is that CHAINSPACE distinguishes between shards that are used as inputs and shards that are used as outputs and only informs output shards after the input shards finish processing a transaction, due to which transaction processing in CHAINSPACE takes one more round as in RCHIMERA.

RINGBFT [55]. RINGBFT uses a *linear two-phase commit protocol* (Linear 2PC) [30, 53], that is implemented via consensus steps and cluster-sending, to order all multi-shard transactions. Due to the usage of Linear 2PC, RINGBFT is able to utilize blocking locks in a deadlock-free manner to provide transaction isolation. Due to this usage of locks, RINGBFT is the only protocol besides CHIMERA that is able to always process valid transactions without spurious aborts. Furthermore, the usage of Linear 2PC minimizes cross-shard communication costs, as all communication is between pairs-of-affected-shards (no all-to-all, one-to-all, or all-to-one communication). The benefits of RINGBFT come at a cost, however, as the linear design imposes a *linear* amount of consecutive consensus and cross-shard communication steps in terms of the shards affected by the transaction, whereas all other proposals require a constant number of consecutive steps.

SHARPER [4]. SHARPER uses a *multi-shard consensus protocol* to order all multi-shard transactions. The operations of this multi-shard consensus protocol are conceptually similar to the design of OCHIMERA, except that SHARPER does not take advantage of any specific properties of the data model (e.g., to provide isolation or to simplify recovery). Furthermore, SHARPER requires that affected shards process their multi-shard transactions in a common processing order, In effect, this imposes a per-shard lock on multi-shard transaction processing, due to which each shard in SHARPER can only engage in at most a single multi-shard transaction at a time, limiting concurrent execution even in the absence of transactions that contend for the same data objects. This is in sharp contrast to the original design of PBFT on which both OCHIMERA and SHARPER are based: the design of PBFT enables implementations with an out-of-order design that al-

lows for the processing of multiple consensus rounds at once, thereby enabling primaries to use all their available bandwidth to continously propose transactions for future consensus rounds *without* waiting for previous consensus rounds to finish. This *out-of-order design* of PBFT enables it to process thousands of consensus rounds per second, even in real-world networks with high message delays [16,31]. OCHIMERA *does* allow for out-of-order processing, however, as the correctness of OCHIMERA does not fundamentally depend on the order in which transactions that affect distinct objects are processed. Finally, the philosophy of SHARPER is to serve as a single unified protocol that can support both PAXOS-style crash fault-tolerance and malicious behavior, and it remains an important research question as to whether SHARPER can be extended to the general-purpose unreliable communication and attack models supported by OCHIMERA. In specific, we believe OCHIMERA improves on the resilience of SHARPER by providing a *robust* local and global view-change mechanism that can deal with per-shard replica failures, per-shard primary failures, and coordinated attacks by replicas and clients to disrupt global consensus steps.

BYSHARD [35] and CAPER [3]. BYSHARD [35] proposes a framework in which one can evaluate many distinct protocols based on the application of two-phase commit and two-phase locking in a consensus-based environment. Specific instances of BYSHARD correspond with the approaches taken by CHAINSPACE and RINGBFT, while AHL can be seen as a restricted case of the BYSHARD protocols that utilize distributed orchestration. The differences between, on the one hand, CHIMERA and, on the other hand, AHL, CHAINSPACE, and RINGBFT, extend to the BYSHARD framework. The design of CAPER [3] shares similarities with the design of SHARPER.

8.3 The Performance Potential of CHIMERA

Next, we modelled the performance benefits of CHIMERA. To do so, we have modeled the maximum throughput of each of these protocols in an environment where each replica has a bandwidth of 1 Gbit/s and the message delay is 15 ms.[6] Furthermore, we test with either seven (of which two can be faulty) or sixty-one replicas (of which twenty can be faulty). We have chosen to optimize CCHIMERA, OCHIMERA, and RCHIMERA to minimize *processing latencies* over minimizing bandwidth usage, as reducing processing latencies is the goal of the design of CHIMERA. In specific, we do *not* use request batching and we do not use *threshold signatures*. To further minimize latencies, we use a primary-initiated one-phase broadcast-based cross-shard exchange step that will succeed in a single communication phase whenever the sending primary is non-faulty and communication is reliable [32]. Although this cross-shard exchange step has low latency and can deal with failures, it does impose a high cost at the primary in the sending shard. In cases when one does not want to optimize for processing latencies and individual replicas have spare computational power, then one can utilize threshold signatures to further boost throughput by a constant factor (at the cost of the per-transaction processing latency).

In Figure 9, we have visualized the maximum attainable throughput (the number of transactions processed per second while processing a workload of 32M transactions) for each of the protocols as a function of the number of shards and as a function of the number of objects affected by each transaction when processing a workload in which 50% of the transactions are single-shard and the remaining transactions affect objects chosen fully-at-random. Hence, when multiple shards are present in the system, the remaining transactions are likely to be multi-shard transactions. As a baseline for comparison, we have also included CHAINSPACE [1] and AHL [17]. For AHL, we used an additional shard as a reference committee (hence, if we use n shards in the experiment, then AHL can use $n+1$).

In general, we see that CCHIMERA has higher throughput than OCHIMERA and OCHIMERA has higher throughput than RCHIMERA. These results are easily explained: all protocols are bottlenecked by the communication at the primaries of shards (as these primaries broadcast proposal messages, which are much larger than any other messages), and in general the shards perform the least amount of work per transaction in CCHIMERA and the most amount of work per transaction under RCHIMERA.

For example, when comparing CCHIMERA with OCHIMERA, we see that the multi-shard consensus of OCHIMERA makes a trade-off in favor of latency at the expense of *higher communication costs* at the primaries of shards than the higher-latency combination of local consensus and cross-shard exchange used by CCHIMERA. This is not always true, however. To minimize latencies for CCHIMERA and RCHIMERA, we used a primary-initiated one-phase broadcast-based cross-shard exchange step that imposes a high cost at the primary in the sending shard. As a consequence, we see that multi-shard transactions that only affect a few shards (e.g., very few shards or very small transactions), result in less communication at the primaries under the multi-shard consensus performed by OCHIMERA than the combined cost of a local consensus and cross-shard exchange step performed by RCHIMERA. Consequently, we see slightly higher throughput for OCHIMERA than for CCHIMERA in these settings.

For RINGBFT, we notice that it is optimized for a completely different design goal: RINGBFT is optimized to minimize communication costs and maximize throughput (by

[6] We have chosen for a bandwidth of 1 Gbit/s and a message delay of 15 ms as this message delay is multiple orders of magnitudes larger than the time it takes to send all bytes of any individual message. Hence, if the protocol has bottlenecks based on message delay, then these bottlenecks will clearly show with the parameters chosen.

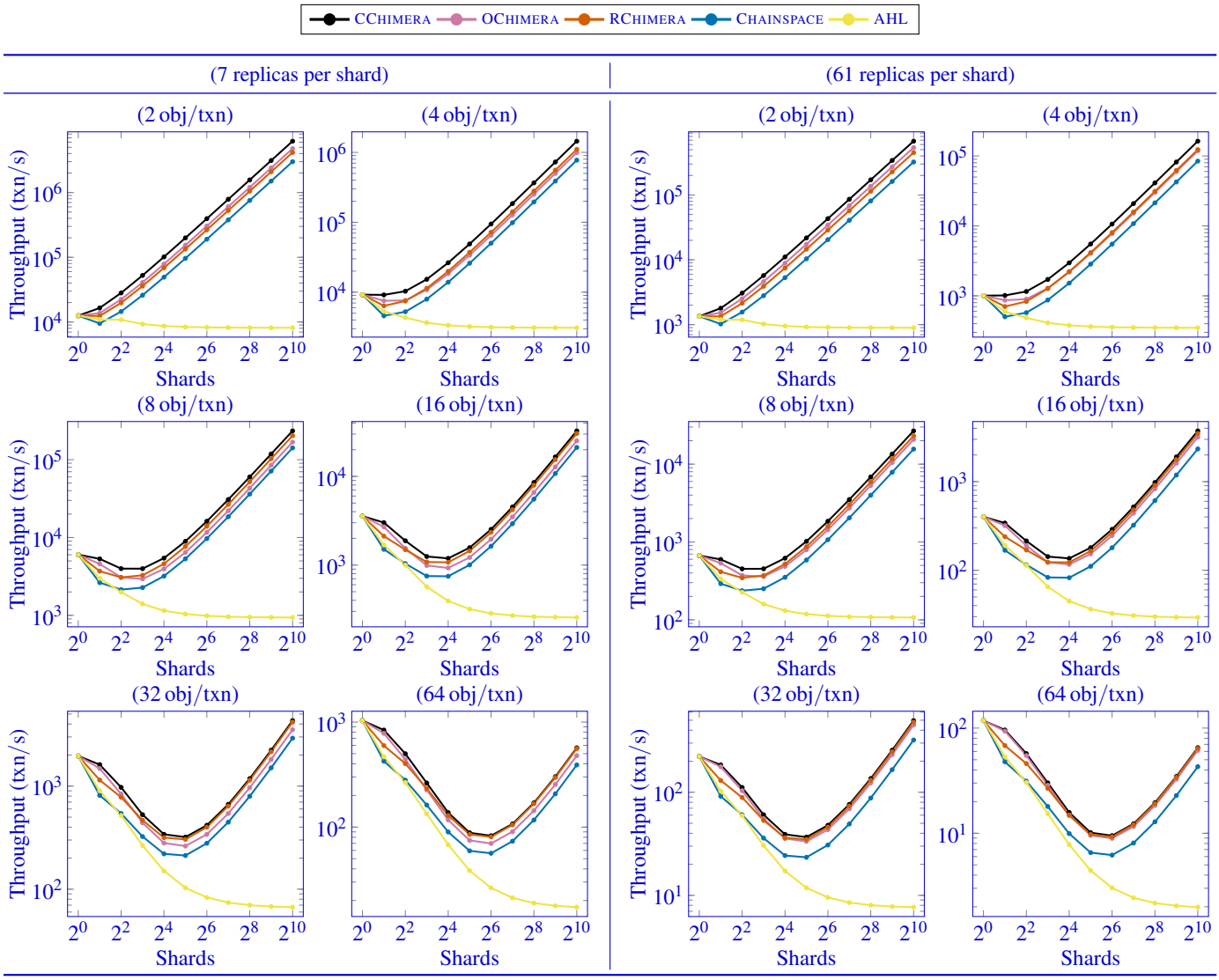

Figure 9: Throughput of the three CHIMERA protocols as a function of the number of shards when processing a workload in which 50% of the transactions are single-shard transactions and all remaining transactions affect objects chosen fully-at-random and, hence, are likely to be multi-shard transactions.

inducing very high latencies for multi-shard transactions), whereas the three CHIMERA protocols aim to maximize throughput while also minimizing latency. To further underline this, we measured the *minimal latency* of processing a multi-shard transaction as a function of the number of affected shards for the three CHIMERA protocols and for RINGBFT, CHAINSPACE, and AHL. We refer to Figure 10 for the results. As is clear from the results, only the latency of RINGBFT is strongly determined by the number of shards. This is not a surprise, as the three CHIMERA protocols, CHAINSPACE, and AHL all operate in a fixed number of consecutive steps, whereas in RINGBFT the number of consecutive steps is a function of the number of affected shards. In specific, CCHIMERA has two consecutive steps (a consensus and a cross-shard exchange step); OCHIMERA has a single step (a multi-shard consensus step); RCHIMERA has three consecutive steps (two consensus steps and a cross-shard exchange step); CHAINSPACE has five consecutive steps (three consensus steps and two cross-shard exchange steps); and AHL has nine consecutive steps (five consensus steps and four cross-shard exchange steps). With regards to behavior, CHAINSPACE is similar to RCHIMERA, except that the construction of outputs is a separate step that happens *after* inputs are destructed. The high number of consecutive steps in AHL is a design choice aimed at reducing *overall* inter-shard communication at the cost of latency by *not* using all-to-all broadcasts between affected shards, but instead steering all communication between shards via an all-to-*reference*

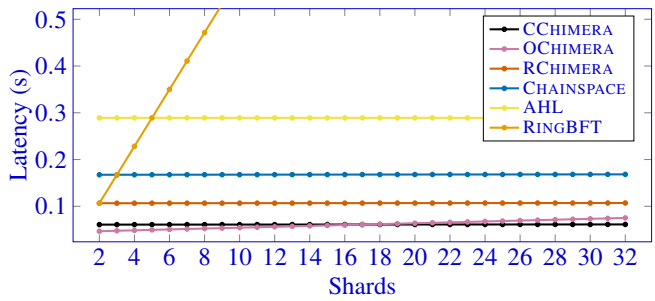

Figure 10: Latency of processing a transaction during multi-shard transaction processing as a function of the number of affected shards, assuming that the transaction affects 64 objects, the network has a bandwidth of 1 Gbit/s, the message delay is 15 ms, and each shard has 7 replicas.

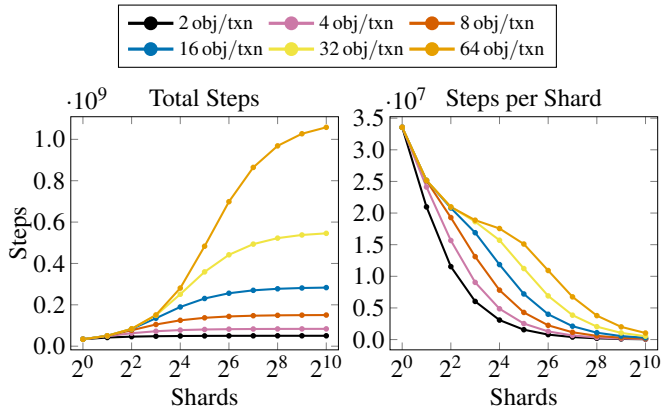

Figure 11: Cumulative number of shards affected by a workload of 32M transactions as a function of the size of transactions and the number of shards. On the *left*, the *total* number of steps. On the *right*, the *average* number of steps per shard. For CCHIMERA and OCHIMERA, the number of steps is equivalent to the number of per-shard consensus steps, for RCHIMERA the number of steps is *half* the number of per-shard consensus steps.

committee-to-all communication pattern, which introduces additional consensus steps.

In our workloads, the *ratio of multi-shard transactions* is high: we want to study how the multi-shard transaction processing protocols we compare differ in their operations and we are especially interested in the performance of the system when dealing with *multi-shard transactions* that require substantial coordination to deal with contention. Indeed, in workloads that mainly consist of *single-shard transactions*, each of the multi-shard transaction protocols we look at will fall back to the same underlying single-shard consensus protocol to efficiently process such single-shard transactions.

In Figure 11, we have visualized the number of per-shard steps performed by the system (for CCHIMERA and OCHIMERA, this is equivalent to the number of per-shard consensus steps, for RCHIMERA this is half the number of per-shard consensus steps). In general, we see that an increase in shards has two effects:

1. Simple single-shard transactions can be dispersed over more shards. Hence, increasing the number of shards will reduce the average number of shard steps each shard has to process with respect to *single-shard transactions*. Furthermore, significantly increasing the number of shards will distribute the *multi-shard transactions* over these shards, reducing the cost of these transactions per shard. Both effects will result in drastically improved performance when scaling to many shards.

2. Large transactions can become more complex when increasing the number of shards, as more shards can hold objects relevant of large transactions. For example, a transaction that affects 16 objects in an environment with four shards can affect at-most four shards, while in an environment with 16 shards it can affect at-most 16 shards). Hence, for large transactions, we only see reductions in the per-shard cost to process these transactions when scaling beyond the number of shards large transactions can affect.

In a general-purpose sharded system without any specific bottlenecks, the above will result in great scalability as soon as the number of shards far outgrows the size of transactions. This behavior is clearly observable for all three CHIMERA protocols. Indeed, all three CHIMERA protocols have excellent scalability: increasing the number of shards will increase the overall throughput of the system. Sharding does come with clear overheads, however, increasing the number of shards also increases the number of shards affected by each transaction, thereby increasing the overall number of consensus steps. This is especially true for very large transactions that affect many objects (that can affect many distinct shards). Hence, as one can see from the results, the benefits of sharding are the strongest when processing mainly single-shard transactions or when scaling beyond the size of individual transactions.

In the comparison between OCHIMERA and RCHIMERA, we see that OCHIMERA (implemented with all-to-all communication) will outperform RCHIMERA whenever transactions involve few shards (due to them involving few objects). In this case, the communication cost of the three cross-shard communication steps that are part of the multi-shard consensus of OCHIMERA is lower than the cost of the second local consensus round in RCHIMERA. When transactions affect many shards, RCHIMERA outperforms OCHIMERA, as RCHIMERA only has a single cross-shard communication step per transaction (and all other communication is local within a shard). In all cases, CCHIMERA will outperform the other protocols with respect to transaction throughput. Furthermore, by design CCHIMERA will always have lower latencies than RCHIMERA (due to the fewer consensus and cluster-sending steps CCHIMERA performs). In environments in

which inter-shard and intra-shard communication have similar (high) message delays, OCHIMERA will typically have lower latencies than CCHIMERA due to the large impact message delays have on the latency of consensus and cluster-sending steps, this even when CCHIMERA has higher throughput than OCHIMERA.

In comparison with CHAINSPACE, we see that this protocol behaves both in theory and in practice similar to RCHIMERA: the main difference being that CHAINSPACE commits in two steps (first, changes to the inputs are committed, then, in a consecutive consensus step, the changes to the outputs are committed). Due to this, CHAINSPACE has slightly higher costs associated with processing multi-shard transactions than RCHIMERA.

In comparison with AHL, we see a large improvement in performance. Due to the high ratio of multi-shard transactions, the performance of AHL for processing multi-shard transactions is bottlenecked by the throughput of the reference committee used by AHL to coordinate processing of *all* multi-shard transactions. These findings are in line with the original evaluation of AHL [17, Section 7.3]. The usage of a reference committee is contrasted by the design of the CHIMERA protocols: the CHIMERA protocols do *not* depend on an external coordinator to process multi-shard transactions. Instead, in the CHIMERA protocols, processing a multi-shard transaction τ is coordinated *only* by the shards affected by τ, due to which the CHIMERA protocol can concurrently process multi-shard transactions that affect distinct sets of shards without introducing bottlenecks. A closer look at the data does reveal *excellent* scalability of AHL with regards to single-shard transactions: although the reference committee has a full load while processing all multi-shard transactions, all shards *except* the reference committee show a very low load (that can be used to process many more single-shard transactions during the experiment).

8.4 CHIMERA and Malicious Behavior

Finally, we have modeled the maximum throughput of each of the three CHIMERA protocols in an environment in which some shards are impacted by malicious replicas. Unless stated otherwise, we use the same setting as in Section 8.3. We have used 64 shards and four objects per shard and we measure the performance of the three CHIMERA protocols as a function of the number of shards that are affected by malicious behavior.

Within a shard, only malicious primaries have a strong impact on the performance of that shard. Furthermore, malicious primaries that completely disrupt normal-case operations will be replaced. Hence, to maximize the malicious impact, we have chosen for malicious primaries that try to *throttle* the performance of the system by slowing down their own operations. In the experiment, we have chosen that these primaries do so by halving the speed by which their shards operate.

In Figure 12, we have visualized the average attainable

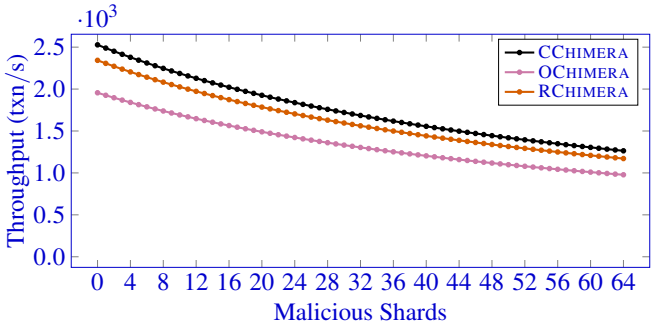

Figure 12: Throughput of the three CHIMERA protocols as a function of the number of shards that are affected by malicious behavior, as measured by the transaction cost at each shard. We have 16 obj/txn for all transactions. In this case, the shards affected by malicious behavior are controlled by malicious primaries that throttle the throughput of that shard to *half* the maximum attainable throughput.

throughput for shards processing workloads in which 50% of the transactions are multi-shard as a function of the number of shards that are affected by malicious behavior (and can affect the speed by which some of the multi-shard transactions processed are processed) for each of the CHIMERA protocols. The shards affected by malicious behavior are set up with malicious primaries that purposely throttle the throughput of the system by slowing down their operations to *half* the speed they could attain. We see that each shard affected by malicious behavior will slow down each CHIMERA protocol with respect to those transactions that are affected by these shards. At the same time, transactions not handled by shards affected by malicious behavior shards are unaffected and will be processed at normal speed due to which the impact of a few affected shards is minimal.

9 Related Work

Distributed systems are typically employed to either increase reliability (e.g., via consensus-based fault-tolerance) or to increase performance (e.g., via sharding). Consequently, there is abundant literature on such distributed systems, distributed databases, and sharding (e.g., [53, 57, 58]) and on consensus-based fault-tolerant systems (e.g., [10, 14, 19, 31, 57]). Furthermore, in Section 8.2, we reviewed related work on multi-shard permissioned consensus-based systems. Next, we focus on other works that deal with sharding in fault-tolerant systems.

In Section 8.2, we have only compared with other sharded resilient systems with similar environmental assumptions. Besides these sharded systems, several other resilient systems such as OMNILEDGER [43] and RAPIDCHAIN [63] have been proposed. These systems make very different environmental assumptions (e.g., different threat and communication models) due to which these systems are incomparable

to the CHIMERA protocols and the systems considered in Section 8.2.

A few fully-replicated consensus-based systems utilize sharding at the level of consensus decision making, this to improve consensus throughput *without* adopting a multi-shard design [2, 22, 26, 32]. In these systems, only a small subset of all replicas, those in a single shard, participate in the consensus on any given transaction, thereby reducing the costs to replicate this transaction without improving storage and processing scalability.

Recently, there has also been promising work on sharding and techniques supporting sharding for permissionless blockchains. Examples include techniques to enable sidechains, blockchain relays, and atomic swaps [23, 24, 37, 39, 44, 60, 62], which each enable various forms of cooperation between blockchains (including simple cross-chain communication and cross-chain transaction coordination). Unfortunately, these permissionless techniques are several orders of magnitudes slower than comparable techniques for traditional fault-tolerant systems, making them incomparable with the design of CHIMERA discussed in this work.

10 Conclusion

In this paper, we took a new look at the problem of multi-shard transaction processing in consensus-based systems. In specific, we proposed the study of *sharded consensus-based systems* that use restrictions on the workloads supported to improve performance over general-purpose methods.

To initiate this study, we introduced Core-CHIMERA, Optimistic-CHIMERA, and Resilient-CHIMERA, three fully distributed approaches towards multi-shard fault-tolerant transaction processing. The design of these approaches is geared towards processing UTXO-like transactions in sharded distributed ledger networks. Due to the usage of UTXO-like transactions, the three CHIMERA variants can minimize cost to an absolute minimum, while maximizing performance, thereby showing the potential of *restricting the types of supported workloads*. This potential is further underlined by our comparison with the state-of-the-art protocols, in which we see that the three CHIMERA variants both have lower costs and complexity.

Although the workloads supported by CHIMERA are minimalistic, we believe that our results can be generalized to more-general settings. In specific, we believe that the combination of sharding and *Conflict-free Replicated Data Types* (CRDTs) [47] has great potential to provide high performance in a consensus-based environment. Another intriguing direction is the investigation of resilient systems that *not only* limit the types of workloads supported, *but also* reduce the resilient guarantees provided by the system, this to further maximize throughput and minimize latency. For example, by no longer using a consensus-based design but instead using small quorum systems [48].

References

[1] Mustafa Al-Bassam, Alberto Sonnino, Shehar Bano, Dave Hrycyszyn, and George Danezis. Chainspace: A sharded smart contracts platform, 2017. URL: http://arxiv.org/abs/1708.03778.

[2] Yair Amir, Claudiu Danilov, Danny Dolev, Jonathan Kirsch, John Lane, Cristina Nita-Rotaru, Josh Olsen, and David Zage. Steward: Scaling byzantine fault-tolerant replication to wide area networks. *IEEE Transactions on Dependable and Secure Computing*, 7(1):80–93, 2010. doi:10.1109/TDSC.2008.53.

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

[4] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. SharPer: Sharding permissioned blockchains over network clusters. In *Proceedings of the 2021 International Conference on Management of Data*, page 76–88. ACM, 2021. doi:10.1145/3448016.3452807.

[5] 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. Hyperledger Fabric: A distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, pages 30:1–30:15. ACM, 2018. doi:10.1145/3190508.3190538.

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

[7] Paddy Baker and Omkar Godbole. Ethereum fees soaring to 2-year high: Coin metrics. *CoinDesk*, 2020. URL: https://www.coindesk.com/defi-hype-has-sent-ethereum-fees-soaring-to-2-year-high-coin-metrics.

[8] Guillaume Bazot. Financial intermediation cost, rents, and productivity: An international comparison. Technical report, European Historical Economics Society, 2018.

[9] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ANSI SQL isolation levels. *SIGMOD Rec.*, 24(2):1–10, 1995. doi:10.1145/568271.223785.

[10] Christian Berger and Hans P. Reiser. Scaling byzantine consensus: A broad analysis. In *Proceedings of the 2nd Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers*, pages 13–18. ACM, 2018. doi:10.1145/3284764.3284767.

[11] Gabi Bracha and Ophir Rachman. Randomized consensus in expected $O((n^2 \log n))$ operations. In *Distributed Algorithms*, pages 143–150. Springer Berlin Heidelberg, 1992. doi:10.1007/BFb0022443.

[12] Christopher Brookins. DeFi boom has saved bitcoin from plummeting. *Forbes*, 2020. URL: https://www.forbes.com/sites/christopherbrookins/2020/07/12/defi-boom-has-saved-bitcoin-from-plummeting/.

[13] Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. In *Advances in Cryptology — CRYPTO 2001*, pages 524–541. Springer Berlin Heidelberg, 2001. doi:10.1007/3-540-44647-8_31.

[14] Christian Cachin and Marko Vukolic. Blockchain consensus protocols in the wild (keynote talk). In *31st International Symposium on Distributed Computing*, volume 91, pages 1:1–1:16. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017. doi:10.4230/LIPIcs.DISC.2017.1.

[15] Michael Casey, Jonah Crane, Gary Gensler, Simon Johnson, and Neha Narula. The impact of blockchain technology on finance: A catalyst for change. Technical report, International Center for Monetary and Banking Studies, 2018. URL: https://www.cimb.ch/uploads/1/1/5/4/115414161/geneva21_1.pdf.

[16] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, 2002. doi:10.1145/571637.571640.

[17] Hung Dang, Tien Tuan Anh Dinh, Dumitrel Loghin, Ee-Chien Chang, Qian Lin, and Beng Chin Ooi. Towards scaling blockchain systems via sharding. In *Proceedings of the 2019 International Conference on Management of Data*, pages 123–140. ACM, 2019. doi:10.1145/3299869.3319889.

[18] Nikhilesh De. CFTC chair: 'a large part' of financial system could end up in blockchain format. *CoinDesk*, 2020. URL: https://www.coindesk.com/cftc-chair-a-large-part-of-financial-system-could-end-up-in-blockchain-format.

[19] Tien Tuan Anh Dinh, Rui Liu, Meihui Zhang, Gang Chen, Beng Chin Ooi, and Ji Wang. Untangling blockchain: A data processing view of blockchain systems. *IEEE Trans. Knowl. Data Eng.*, 30(7):1366–1385, 2018. doi:10.1109/TKDE.2017.2781227.

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

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

[22] Michael Eischer and Tobias Distler. Scalable byzantine fault-tolerant state-machine replication on heterogeneous servers. *Computing*, 101:97–118, 2019. doi:10.1007/s00607-018-0652-3.

[23] Muhammad El-Hindi, Carsten Binnig, Arvind Arasu, Donald Kossmann, and Ravi Ramamurthy. BlockchainDB: A shared database on blockchains. *Proc. VLDB Endow.*, 12(11):1597–1609, 2019. doi:10.14778/3342263.3342636.

[24] Ethereum Foundation. BTC Relay: A bridge between the bitcoin blockchain & ethereum smart contracts, 2017. URL: http://btcrelay.org.

[25] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985. doi:10.1145/3149.214121.

[26] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP, pages 51–68. ACM, 2017. doi:10.1145/3132747.3132757.

[27] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, 2002. doi:10.1145/564585.564601.

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

[29] William J. Gordon and Christian Catalini. Blockchain technology for healthcare: Facilitating the transition to patient-driven interoperability. *Computational and Structural Biotechnology Journal*, 16:224–230, 2018. doi:10.1016/j.csbj.2018.06.003.

[30] Jim Gray. Notes on data base operating systems. In *Operating Systems, An Advanced Course*, pages 393–481. Springer-Verlag, 1978. doi:10.1007/3-540-08755-9_9.

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

[32] Suyash Gupta, Sajjad Rahnama, Jelle Hellings, and Mohammad Sadoghi. ResilientDB: Global scale resilient blockchain fabric. *Proc. VLDB Endow.*, 13(6):868–883, 2020. doi:10.14778/3380750.3380757.

[33] Suyash Gupta, Sajjad Rahnama, Jelle Hellings, and Mohammad Sadoghi. Proof-of-execution: Reaching consensus through fault-tolerant speculation. In *Proceedings of the 24th International Conference on Extending Database Technology (EDBT)*, pages 301–312. OpenProceedings.org, 2021. doi:10.5441/002/edbt.2021.27.

[34] Jelle Hellings and Mohammad Sadoghi. Brief announcement: The fault-tolerant cluster-sending problem. In *33rd International Symposium on Distributed Computing (DISC 2019)*, pages 45:1–45:3. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019. doi:10.4230/LIPIcs.DISC.2019.45.

[35] Jelle Hellings and Mohammad Sadoghi. Byshard: Sharding in a byzantine environment. *Proceedings of the VLDB Endowment*, 14(11):2230–2243, 2021. doi:10.14778/3476249.3476275.

[36] Jelle Hellings and Mohammad Sadoghi. The fault-tolerant cluster-sending problem. In Ivan Varzinczak, editor, *Foundations of Information and Knowledge Systems*, pages 168–186. Springer, 2022. doi:10.1007/978-3-031-11321-5_10.

[37] Maurice Herlihy. Atomic cross-chain swaps. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*, pages 245–254. ACM, 2018. doi:10.1145/3212734.3212736.

[38] Maurice Herlihy. Blockchains from a distributed computing perspective. *Commun. ACM*, 62(2):78–85, 2019. doi:10.1145/3209623.

[39] Maurice Herlihy, Barbara Liskov, and Liuba Shrira. Cross-chain deals and adversarial commerce. *The VLDB Journal*, 2021. doi:10.1007/s00778-021-00686-1.

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

[41] Maged N. Kamel Boulos, James T. Wilson, and Kevin A. Clauson. Geospatial blockchain: promises, challenges, and scenarios in health and healthcare. *International Journal of Health Geographics*, 17(1):1211–1220, 2018. doi:10.1186/s12942-018-0144-x.

[42] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography*. Chapman and Hall/CRC, 2nd edition, 2014.

[43] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. OmniLedger: A secure, scale-out, decentralized ledger via sharding. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 583–598. IEEE, 2018. doi:10.1109/SP.2018.000-5.

[44] Jae Kwon and Ethan Buchman. Cosmos whitepaper: A network of distributed ledgers, 2019. URL: https://cosmos.network/cosmos-whitepaper.pdf.

[45] Leslie Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):51–58, 2001. Distributed Computing Column 5. doi:10.1145/568425.568433.

[46] Laphou Lao, Zecheng Li, Songlin Hou, Bin Xiao, Songtao Guo, and Yuanyuan Yang. A survey of iot applications in blockchain systems: Architecture, consensus, and traffic modeling. *ACM Comput. Surv.*, 53(1), 2020. doi:10.1145/3372136.

[47] Mihai Letia, Nuno Preguiça, and Marc Shapiro. Consistency without concurrency control in large, dynamic systems. *SIGOPS Oper. Syst. Rev.*, 44(2):29–34, 2010. doi:10.1145/1773912.1773921.

[48] J.-P. Martin, L. Alvisi, and M. Dahlin. Small byzantine quorum systems. In *Proceedings International Conference on Dependable Systems and Networks*, pages 374–383. IEEE, 2002. doi:10.1109/DSN.2002.1028922.

[49] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2009. URL: https://bitcoin.org/bitcoin.pdf.

[50] Arvind Narayanan and Jeremy Clark. Bitcoin's academic pedigree. *Commun. ACM*, 60(12):36–45, 2017. doi:10.1145/3132259.

[51] Senthil Nathan, Chander Govindarajan, Adarsh Saraf, Manish Sethi, and Praveen Jayachandran. Blockchain meets database: Design and implementation of a blockchain relational database. *Proc. VLDB Endow.*,

12(11):1539–1552, 2019. doi:10.14778/3342263.3342632.

[52] Faisal Nawab and Mohammad Sadoghi. Blockplane: A global-scale byzantizing middleware. In *35th International Conference on Data Engineering (ICDE)*, pages 124–135. IEEE, 2019. doi:10.1109/ICDE.2019.00020.

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

[54] Michael Pisa and Matt Juden. Blockchain and economic development: Hype vs. reality. Technical report, Center for Global Development, 2017. URL: https://www.cgdev.org/publication/blockchain-and-economic-development-hype-vs-reality.

[55] Sajjad Rahnama, Suyash Gupta, Rohan Sogani, Dhruv Krishnan, and Mohammad Sadoghi. RingBFT: Resilient consensus over sharded ring topology. In *Proceedings of the 25st International Conference on Extending Database Technology*, pages 298–311. OpenProceedings.org, 2022. doi:10.48786/edbt.2022.17.

[56] Victor Shoup. Practical threshold signatures. In *Advances in Cryptology — EUROCRYPT 2000*, pages 207–220. Springer Berlin Heidelberg, 2000. doi:10.1007/3-540-45539-6_15.

[57] Gerard Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, 2nd edition, 2001.

[58] Maarten van Steen and Andrew S. Tanenbaum. *Distributed Systems*. Maarten van Steen, 3th edition, 2017. URL: https://www.distributed-systems.net/.

[59] Gavin Wood. Ethereum: a secure decentralised generalised transaction ledger, 2016. EIP-150 revision. URL: https://gavwood.com/paper.pdf.

[60] Gavin Wood. Polkadot: vision for a heterogeneous multi-chain framework, 2016. URL: https://polkadot.network/PolkaDotPaper.pdf.

[61] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. HotStuff: BFT consensus with linearity and responsiveness. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, pages 347–356. ACM, 2019. doi:10.1145/3293611.3331591.

[62] Victor Zakhary, Divyakant Agrawal, and Amr El Abbadi. Atomic commitment across blockchains. *Proc. VLDB Endow.*, 13(9):1319–1331, 2020. doi:10.14778/3397230.3397231.

[63] Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova. Rapidchain: Scaling blockchain via full sharding. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 931–948. ACM, 2018. doi:10.1145/3243734.3243853.

Response to the Reviewers' comments
"Problem: Sharded Resilient Transaction Processing with Minimal Costs"

Anonymous authors

March 1, 2023

We would like to tank the chairs and the reviewers for providing us the opportunity to revise our work. We have detailed below our rebuttal to reviewers comments and, where applicable, the corresponding changes we made to the paper (highlighted in the paper in blue). Before we continue, we like to thank each of the *three* reviewers for their careful review and their detailed feedback.

Review #1 (reviewer MJ9g)

> **Review:** Assumption 2.2 starts with how the transaction should look like – which I personally think should be a specification, rather than an assumption. It's also non-typical that each transaction needs to have an input. Doesn't that rule out PoW? Or can the input field be a dummy one or a null? This needs to be explained so that readers have a clear picture.

For the presentation in the paper, the main purpose of Assumption 2.2 is to assume that one can determine inputs and outputs of a transaction (e.g., verifying Requirement 1 locally depends on this assumption). The second assumption (non-empty inputs) is only there to simplify presentation and can be worked around. Hence, we have reworded the assumption to clarify this.

> **Review:** Owner of an object is not defined.

What owners are exactly will depend on the application and the specifics are not essential for the operation of any of the protocols in the paper. Typically the owner is an outside party (e.g., client or user of the system) that can authenticate transactions involving specific objects. We have clarified the role of owners in the paper.

> **Review:** It's not clear what it means by "shards that are affected by a transaction." Is an input object affected? If yes, why?

The shards $\texttt{shards}(\tau)$ affected by transaction τ are all shards that hold either input or output objects of transaction τ. We have clarified this in the paper.

Input objects are strongly affected by τ, as execution of τ will destruct inputs (see the paragraph above Assumption 2.2).

> **Review:** It's also strange that the model assumes that "a set of replicas operated as a Byzantine fault-tolerant (consensus) system." What if a shard adopts a weaker abstraction? E.g., a Byzantine atomic register? Or a DAG-based system? They don't exactly use consensus. If the model/solution only works for consensus-based systems, then this should be clearly specified early in the introduction.

As the reviewer correctly remarks, the current focus of the paper is on consensus-based resilient systems. We have made this more explicit in the Introduction. Furthermore, we have further underlined our focus on consensus-based clusters in Section 2.

We remark that the main problem outlined in the Introduction and investigated in this paper can certainly be applied to other non-consensus-based resilient systems that provide even weaker guarantees to further maximize throughput and minimize latency. We believe that such approaches are interesting directions for future work. We have added a note regarding such future work to the conclusion.

> **Review:** Even if a shard is operating on a consensus-based system, the following assumption is not precise: "As these systems are fully-replicated, each replica holds exactly the same data." There could be delay, so it's possible that a replica is straggling and doesn't have the up-to-date data.

We agree with the reviewer. Indeed, technically replicas will only hold a common prefix of all consensus decisions that have been made. We have made the statement more precise.

> **Review:** It's not clear what "reliable ordered replication" means. This doesn't seem to be a standard term. Please define it clearly.

We have reformulated the definition to clarify that we intend the usage of consensus to make a sequence of decisions (one per round).

> **Review:** Definition 3.1 is quite strong. It rules out most randomized consensus systems, e.g., PoW or PoS which may switch on different chains. Therefore, I'm not sure how practical the proposed solution is.

The paper indeed focuses mainly on permissioned systems (e.g. as operated by PBFT-like primary-backup consensus) for which the strong claims of Definition 3.1 are rather normal. See, e.g., early formalizations such as `https://doi.org/10.1145/2455.214112` and `https://doi.org/10.1145/357172.357176` and the formalizations in " Introduction to Distributed Algorithms" (see citations in the paper). Definition 3.1 is satisfied by all permissioned consensus protocols we are aware of, e.g., primary-backup consensus protocols such as PBFT. Furthermore, a slight variation of Definition 3.1 is also satisfied by traditional probabilistic consensus protocols such as the seminal asynchronous consensus protocol by Bracha and Toueg (`https://doi.org/10.1145/4221.214134`) and modern iterations on that protocol (e.g., HoneyBadger and Dumbo, see `https://eprint.iacr.org/2016/199.pdf` and `https://doi.org/10.1145/3372297.3417262`). The reviewer is right that our Definition rules out the usage of permissionless techniques such as Proof-of-Work, as these techniques do strictly speaking never *fully guarantee* convergence to a single consensus decision. To make clear we focus mainly on permissioned systems, we have further clarified the permissioned environment we focus on in the paper.

> **Review:** The correctness condition requires each input object owner to approve the transaction, which seems too expensive or even impossible in practical systems. What if the owner is faulty or slow? In real workload, it's possible that an object is a hotkey, which will penalize the owner, because it needs to approve tons of transactions. I'm not sure this is a realistic or favorable assumption to adopt.
>
> ...
>
> I don't understand why validity is a local property. What if the owners in shard S all approved the transaction, but the owners in another shard S' disapproved the transaction. I thought this transaction should be viewed as invalid, but then each shard alone cannot check for the validity condition.

Ownership, as assumed in the paper, is controlled by parties outside the system (e.g., clients and users). Typically, ownership is implemented via public-key cryptography and proof-of-approvals can be provided via digital signatures attached to the transaction. Hence, this requirement comes at no additional communication cost to the system and validity can be checked purely based on the content of a transaction. We added a footnote in Section 2 to further explain object owners.

> **Review:** It's not clear to me what the difference is between commit and execute. Please define.

Execution can refer to both commiting or aborting a transaction. We have reworked the definition in Section 3 to clarify the meaning of commit and abort (right above the requirements R1–R6). We have kept the definition of execution at the end of Section 3.

> **Review:** R2 is not very clear. Why can't the system off-load processing to other shards or systems?

Technically, heavy computations can be off-loaded. Off-loading non-computational transaction in the sharded environment used in this paper makes little sense, however: involved shards (that hold the objects affected by a transaction) always *need* to update their internal state as a consequence of the execution of a transaction (as they are responsible for storing objects affected by the transaciton), even if any computations for that transaction are offloaded to some other system.

> **Review:** In R3, τ "consumes only existing objects." This requirement is not clear to me at all. What is time t? What if some object does not exist at time t, but appears at time t+1. Should the transaction be committed? What if the object exists at time t, but then later got removed at time t+1? Should the transaction be committed?

The CHIMERA protocols do not attempt to maximize the amount of transactions that can be committed: if a transaction is picked up for processing at any time t, then only at that point will the shards processing the transaction determine whether it can be processed (whether the objects exist at that point). If objects are missing or are not yet created at t, then the transaction will abort. We have reformulated R3 to better express this intent.

> **Review:** I personally couldn't interpret the evaluation section at all. I understand that it's some sort of simulation based on some performance modeling, primarily on communication patterns. However, I don't see many details for me to understand the meaning of the figures. It's stated that the network is 1 Gbit/s, and there is some latency (message delay = 15ms in Figure 10). However, more information is needed. Is this a typical number? Are we looking at only a cluster setup? Why? Wouldn't the wide-area network setting be more reasonable for the targeted case? If the number is an average, why? What is causing fluctuations of the performance? (It's "performance modeling" anyway. Is the delay varying?) What is the size of the transaction? Why 7 and 61 replicas per shard? Are they typical? Why only testing 50

We have chosen a message delay of 15 ms as it is the minimum message delay you will see in practice if not all replicas are deployed within the same data center. Wide-area (cross-continent and inter-continent) deployments can easily reach much higher message delays (e.g., 100 ms–400 ms). We have chosen for 15 ms as it is multiple orders of magnitudes larger than the time it takes to send all bytes of any individual message. Hence, if the protocol has bottlenecks based on message delay, then these bottlenecks will clearly show with the parameters chosen.

Network speeds of up-to-1 Gbit/s are realistic for cross-continent deployments with replicas placed in distinct datacenters, but for inter-continent deployments one typically sees network speeds that are an order of magnitude lower.

Fluctuations in throughput in our evaluation are due to the randomly generated workloads, which are expected to uniformly affect all shards evenly.

The number of replicas per shard are chosen as follows: in permissioned blockchains, the minimum size of a cluster $\mathcal{S}$ is 4 replicas (which can deal with $\mathbf{f}_{\mathcal{S}} = 1$ faulty replica, as clusters require $\mathbf{n}_{\mathcal{S}} = 3\mathbf{f}_{\mathcal{S}} + 1$ replicas). Hence, as a lower bound on the number of replicas we have chosen 7 replicas (which can deal with two faulty replicas per shard, allowing for repairs when a single replica fails). To investigate the impact of the size of clusters, we have also included measurements with large clusters (61 replicas, which can deal with twenty simultaneous replica failures per shard). These cluster sizes are typical for permissioned systems due to the costs of operating even larger systems (both financial and in terms of reduced performance), while the additional resilience obtained by even larger clusters is minimal.

We have addressed the above in the paper. Furthermore, we have improved our analysis.

Review #2 (reviewer yNuG)

> **Review:** The introduction motivates the use of distributed ledger technology (DLT), but it does not motivate the idea of limiting the workload to UTXO-like transactions. However, I think the paper should justify the choice of UTXO-like transactions, discuss some examples of where they are being used, and whether they are prevalent or not.

UTXO-like transactions are mainly known for their usage in Bitcoin and other cryptocurrencies. They form an abstract model that can easily be used for many other types of workloads, e.g., versioned files. We have included an example of such in the paper.

> **Review:** Can CHIMERA protocols be extended to support other types of transactions? If so, the paper would benefit from discussing that.

For CChimera, we believe the natural next step is to further restrict the guarantees the protocol provides, as this would allow the replacement of consensus by lighter resilient primitives. Both OChimera and RChimera are more general protocols as they are more flexible (less limited) with regards to the assumptions on UTXO transactions originating from faulty replicas.

Of these protocols, RChimera is the easiest to further modify to general-purpose workloads: one can replace the UTXO-data model used by RChimera by a general-purpose data model with two-phase locking instead of object construction and destruction to guarantee isolated execution (depending on the type of transactions, this can involve locks at additional shards, however). By doing so, we will end up with a design that is very similar to that of Chainspace or ByShard, however. We have added a clarification along these lines to Section 6.

Another direction of generalization is toward different types of limitations of workloads, e.g., toward CRDTs. We explorer this option in the conclusion.

> **Review:** I felt that that the discussion of RCHIMERA is shallow. For instance, why do replicas have to force a second step of consensus?

In RChimera, the second consensus step in a shard $\mathcal{S}$ starts after all good replicas in that shard have received all required information to decide either commit or abort. Hence, at that point RChimera will perform a consensus step to *order* the commit or abort decision among all other decisions made in that shard. Now assume individual shards use a primary-backup consensus protocol for their operations. If shard $\mathcal{S}$ is coordinated by a good primary, then such a consensus step will occur automatically. Otherwise, the good replicas need to force the consensus step (e.g., by replacing the primary). We have added a further clarification to the paper.

> **Review:** The paper mentions at different places that, for several steps, either all-to-all or an all-to-one-to-all communication can be used, however, it does not state any conditions on the "one" replica in the all-to-one-to-all communication. Does this replica must be good? What if it is malicious?

With respect to OChimera, we have added the following detail: "Such a design will require a single replica responsible for receiving all messages, aggregating them into a single certificate, and broadcasting this message back. To deal with failures of this replica, we can simply choose a primary (whose failures are already dealt with by the normal recovery mechanisms that we shall describe). Alternatively, one can use the approach taken by Sbft and choose any other replica as the aggregator (thereby offloading the primary), but this will require additional recovery mechanisms to deal with failures of that replica."

> **Review:** Page 6: "CCHIMERA effectively eliminates all coordination when deciding to process a multi-shard..". This statement seems inaccurate as "Cross-shard exchange" step has to be performed before reaching a decision about a transaction.

From a per-shard viewpoint, the order in which replicas within a single shard process a multi-shard transaction will be decided *before* any coordination and, hence, independent of whichever order the other shards choose. Only the outcome of processing the transaction are influenced by the cross-shard exchange. We have clarified this viewpoint in the text.

Review: One main concern I have is that the evaluation results are based on modeling and not based on real implementation of the proposed protocols. So, I am not sure to what level the results can be trusted as modeling might omit significant details. Nonetheless, below are my comments about presented results.

In our experience, the main bottleneck in real-world non-locally-deployed resilient systems (with message delays above a few millisecond) for throughput is network bandwidth (as limitations on computational power, e.g., to verify signatures and execute transactions can be dealt with using heavy parallelization at individual replicas). With respect to bandwidth usage and message delays, the modeling is faithful.

Review: Figure 8 compares only the number of consensus steps needed for each protocol during the normal case. I guess it would be more insightful if it includes the complexity of each step (maybe as number of messages) as well as the number of steps (phases) needed for recovery mechanism.

We can only provide that information for OChimera. The recovery mechanisms of OChimera to perform a single view-change cost at-least two phases of communication (no global state recovery) and at-most four phases of communication (global state recovery).

For all other protocols, the cost of recovery depends on whichever consensus protocol and cross-shard exchange protocols are used internally to operate individual shards. Pbft-like view-change recovery mechanisms require at-least two phases of communication, this similar to the view-change of OChimera (without global state recovery).

Review: I felt the selected systems for comparison is a bit biased.

To address the concerns of this and other reviewers considering the comparison with other solutions, we have expanded our treatment of AHL and Chainspace.

Review: page 17: "We excluded CHAINSPACE, as it has a design very similar to RCHIMERA (but slightly more costly due to some overheads RCHIMERA can avoid)." I guess you should include CHAINSPACE in order to evaluate how much you gain by avoiding these overheads.

We have included Chainspace in the revision.

Review: The throughput experiment (Figure 9) compared against AHL while the latency experiment (Figure 10) compared against RINGBFT. The paper states: "RINGBFT is optimized to minimize communication costs and maximize throughput (by inducing very high latencies for multi-shard transactions)". So, I guess for fair comparison, both Figure 9 and Figure 10 must include AHL and RINGBFT.

We have included AHL in both experiments. In addition, we have also added Chainspace to both experiments. We have not included RingBFT in the throughput experiments, as it design is fundamentally different than the Chimera protocols, AHL, and Chainspace.

Review: Why does Figure 12 not include other systems?

We have only studied the other systems under optimal conditions (no malicious behavior).

Review: Nits Page 5, 11, 13, 15: "that content" → that contend page 6: "in a asynchronous" → an asynchronous Figure 8 has "GC" symbol but caption does not state what does it mean Figure 10 misses the X-axis title Figure 12 misses the Y-axis title

All fixed.

Review #3 (reviewer X8kA)

> **Review:** The paper spends a lot of time on its proposed solution (CChimera), beyond motivating the problem itself or explaining why their formalization is the right one.

We do not believe there is a*right* choice: many different approaches toward sharding exist, each with their own benefits and drawbacks. The focus of CHIMERA is to explorer high-throughput low-latency designs by restricting the types of workloads. To show that such a focus is waranted, we indeed provide and analyze a complete set of protocols build around the idea of restricting workloads to show the benefits of doing so with respect to latency and throughput.

> **Review:** It is not clear to me that the "sharded consensus-based system" formalization accurately reflects the designs used in sharded blockchains. For example, Polkadot and Eth2 both use a central "beacon" chain, which actually executes consensus, to checkpoint all the shards (which need no consensus of their own).

There are many different ways in witch blockchains are sharded in practice. We only explorer designs that aim to maximizing performance (in terms of transaction throughput) by introducing throughput scalability. Viewed via that lens, we see that sharded designs in which a *central* party decides the order of all transactions makes that central party the main bottleneck with regards to performance scalability (as the performance will naturally be bottlenecked by the speed by which the central party can do its job, see, e.g., the reference committee of AHL). Sharded designs such as used by the CHIMERA protocols, CHAINSPACE, and RINGBFT do *not have central parties*, as they are fully designed to enable the *concurrent* processing of transactions at distinct shards. These designs thereby introduce performance scalability (especially when not processing multi-shard transactions).

> **Review:** There are no proofs that the improvements over AHL represented by OChimera or RChimera are inherent to "limiting the types of workload supported." Indeed, it seems possible to implement an extremely general state machine on top of either, and achieve low cost of coordination. This calls the seemingly positive answer to the central question into doubt.
>
> ...
>
> In their response to previous reviewers, the authors promised to clarify "how to generalize OChimera and RChimera to general-purpose workloads." I'm not sure this is clear in the current paper. Moreover, I think it undercuts their central thesis. If it is possible "to generalize OChimera and RChimera to general-purpose workloads," then, given that OChimera and RChimera perform well, it is not clear whether one can "reduce the cost of coordination in the design of sharded consensus-based systems by limiting the types of workloads supported?" I thought the authors' intent was to "give a preliminary positive answer."
>
> ...
>
> Why does Chimera do any better than AHL?
>
> ...
>
> If OChimera and RChimera can address unlimited workloads, you need to explain exactly how to convert, say, AHL style transactions to RChimera transactions. Moreover, if you still want to argue that one can "reduce the cost of coordination in the design of sharded consensus-based systems by limiting the types of workloads supported," you have to emphasize performance improvements (either theoretical, practical, or both) of CChimera over RChimera or OChimera, and preferably show that these are fundamental, rather than stemming from design choices made. Comparing OChimera and RChimera to AHL etc. then serves mainly to prove that you're comparing to "better than state of the art," rather than a straw-man.

The CCHIMERA protocol takes most advantage of the workload limitations, as it uses those limitations to avoid anything akin coordination between shards while initially processing the transaction (coordination is only used in CCHIMERA to decide the outcome of a multi-shard transaction *after* individual shards have

chosen in which order they will process the transaction). Due to this, CChimera sees the highest throughputs and lowest latencies. Both OChimera and RChimera are more general protocols as they are more flexible (less limited) with regards to the assumptions on UTXO transactions originating from faulty replicas.

Of these protocols, RChimera is the easiest to further modify to general-purpose workloads: one can replace the UTXO-data model used by RChimera by a general-purpose data model with two-phase locking instead of object construction and destruction to guarantee isolated execution (depending on the type of transactions, this can involve locks at additional shards, however). By doing so, we will approach a similar design space to that of Chainspace or ByShard, however. We have added a clarification along these lines to Section 6.

Another direction of generalization is toward different types of limitations of workloads, e.g., toward CRDTs. We explorer this option in the conclusion.

The benefits of the Chimera protocols, when comparing with AHL, are overshadowed by the peculiarities of the design of AHL, however. As explained in Section 8.2, AHL uses a reference committee to coordinate multi-shard transaction processing for *all* multi-shard transactions. As such, this reference committee will quickly become a bottleneck with workloads heavy in multi-shard transactions. This previously-reported behavior of AHL can also be seen in our results, as we outlined at the end of Section 8.3,

The usage of a reference committee is contrasted by the design of the Chimera protocols: the Chimera protocols do *not* depend on an external coordinator to process multi-shard transactions. Instead, in the Chimera protocols, processing a multi-shard transaction τ is coordinated *only* by the shards affected by τ, due to which the Chimera protocol can concurrently process multi-shard transactions that affect distinct sets of shards without introducing bottlenecks.

We have reworded our discussion of AHL in both Section 8.2 and 8.3 to highlight these differences.

As Chainspace does not have the performance peculiarities of AHL, we have also added Chainspace in Section 8.3 and 8.4.

> **Review:** The limitations of CChimera seems so weak that they allow a considerably weaker system than "sharded consensus" to provide very similar guarantees, with less latency. See detailed comments.
>
> Detailed Comments Reviewer 3UEy and I both noticed that, for CChimera's limited types of workloads, it seems that a sharded consensus-based system may be unnecessary in the first place, when something simpler will suffice. I will try to explain what I mean. Consider a simpler (non-consensus) based system, which would provide liveness for "honest" owners:
>
> A node recognizes that a UTXO in OUTPUTS(T) has been created when it receives a message ACK(T) from every member of a quorum from each of SHARDS(T).
>
> A node N on shard S broadcasts ACK(T) when it receives a transaction T, and N recognizes that all resources in INPUTS(T) on shard S exist and have not yet been consumed, and T is valid. It then marks all resources in INPUTS(T) on shard S as consumed (in N's local storage).
>
> Safety: No 2 TXs can consume the same resource (it'd be marked as consumed by a quorum intersection in that resource's shard). No resource in OUTPUTS(T) can be recognized as created until all its inputs are consumed. Liveness: If only one TX is authorized to consume a given resource, then everything terminates in 1 broadcast. A TX which consumes as input a contested resource may stall indefinitely. A TX which consumes as input a not-yet-created resource may stall indefinitely. I note that transitive dependence on contested resources could result in stalling, which might be a weakness against CChimera. I think this violates R6. HOWEVER: you could require that clients themselves pass along a "proof this UTXO was created" to anyone else wanting to use the UTXO as input, and then you would get R6. Such a proof could consist of aggregated ACK messages from the transaction that created the UTXO. In fact, if you do this, nodes only have to store "consumed" sets. They don't even have to store "known to exist" sets. This simpler setup calls into question the usefulness of CChimera as a proof that limiting workload types can decrease cost of coordination in a sharded consensus-based system. It's not surprising that we can reduce coordination cost when we didn't really need a sharded consensus-based system in the first place.
>
> ...

Given that Reviewer 3UEy and I both immediately tried to think of a way to meet CChimera's use case without consensus, directly address either why we are mistaken (for which a more formal proof may be required than was in the rebuttal), or argue that your answer is only "preliminary," and CChimera proves that limited workloads can improve performance, but the search for usefully limited workloads that still require consensus-based sharded systems is ongoing.

We agree with the reviewer that one can further apply the idea of "reducing the cost of coordination" in (sharded) resilient systems significantly by weakening the guarantees provided. In our paper, we explorer one such approach: putting restrictions on the workload. As the reviewer points out, one can further weaken guarantees to further reduce the cost of maintaining the system: e.g., by utilizing a design based on small quorum systems (e.g., `https://doi.org/10.1109/DSN.2002.1028922` which come close to the approach outlined by the reviewer). The main difference between the description of the reviewer and our CHIMERA protocols is that our protocols provide stronger guarantees on the status of individual good replicas in all shards (due to the usage of consensus).

Review: I am not clear on the justifications for the difference between OChimera's consensus and PBFT. Why not choose a single PBFT leader from any leader of any shard, and then use standard PBFT? Since OChimera consensus fails if any leader chosen is corrupt, picking a single leader would be strictly more likely to succeed.

Using a singular PBFT across all affected replicas is easier said than done. Using a singular PBFT across all affected replicas is easier, but to make such a design high-performance we need that several instances of PBFT can process multi-shard transactions concurrently (namely those transactions that affect distinct shards). Hence, to maximize the ability of the system to concurrently process multi-shard transactions, we will need a PBFT instance for every possible subset of shards (each instance maintaining its own primary, operating in its own view, and coordinating its own multi-shard transactions). These independent PBFT instances still need some form of coordination among each other, however (as PBFT instances overlap in the replicas they control and different PBFT instances can decide on the same data). Hence, such an approach would likewise introduce new failure modes due to the interaction of different instances (with different primaries).

Review: The paper should probably note that replicas should be able to locally determine whether a transaction is "valid" or "well-formed" or obeys system invariants (e.g. no double-spends), and only then is it "applicable." Right now, it just says "The validity of transactions is a local requirement: whether a transaction τ is valid can be determined by checking whether all owners of inputs of τ support that transaction."

Review: I think this paper can be made appropriate for JSys with a little reframing.

This is a problem paper, so focus on the problem: what is are the "unlimited" types of workloads, and for what applications do you think "limited types of workloads" will be acceptable?

Spend more time explaining how your formalization of sharded consensus-based systems is the right one. What existing or proposed systems does it accurately describe?

Review: What precisely is the purpose of OChimera and RChimera in presenting this problem?

If OChimera and RChimera can only address limited workloads (but not as limited as CChimera), point that out, as they form additional "positive" answers to your question.

CCHIMERA is rather limited. Both OCHIMERA and RCHIMERA are attempts to reduce these limitations without directly falling back to a general-purpose two-phase commit model. Of these protocols, RCHIMERA is closest to a two-phase commit, except that RCHIMERA essentially only puts non-blocking locks on input objects and does not lock any of the outputs. As such, the isolation guarantees provided by RCHIMERA *if* RCHIMERA would operate on a different non-UTXO data model would be not as strong as general-purpose systems can achieve.

Review: Introduction, paragraph 3: I'm not sure it's technically true that permissions blockchains can have higher throughput or that they must be fully-replicated. Permissioned chains can use sharded storage, and non-permissioned proof-of-stake chains can use the same consensus mechanisms (e.g. PBFT) as permissions ones.

The general message of the third and fourth paragraph is as follows: when comparing blockchains, those operated with high-performance implementations of permissioned consensus protocols (e.g., PBFT) will outperform those that utilize permissionless consensus protocols; and those that utilize sharding have the *potential* to outperform non-sharded systems. We have slightly reworded the paragraphs to highlight our general message.

Review: "CChimera will operate perfectly" : you have to define "perfectly." In particular, you'll have to argue that one round of cross-shard broadcast is indeed "perfect."

Here, perfect simply refers to the fact that CCHIMERA is able to commit all transactions proposed and approved by well-behaved clients. At this point, the terminology "commit" has not yet been introduced, however. We have reworded the introduction to make this meaning clearer.

Review: Definition 3.1: define "decisions."

Definition 3.1 part 3: what does it mean to "force consensus." This is clearly getting at some kind of censorship resistance, but it's not spelled out, and it's not clear that known consensus protocols have this.

Section 4: "The primary P(S)" unstated assumption: this is the first time we've assumed we have a something called a primary.

Section 5: "View v of S" is used before it is defined (Is it defined?)

In this paper, we focus mainly on systems operated via permissioned consensus protocols. These protocols are often primary-backup protocols, e.g., protocols in the lineage of PBFT.

In practice, all primary-backup protocols can prove some form of censorship resistence: in a typical resilient system in which consensus decisions are based on external client requests, a client can force consensus decisions for its requests by simply requesting *all* non-faulty replicas to handle their client request. Next, non-faulty replicas suggest these client requests to the primary whom is then required to eventually propose these client requests. A failure to do so means a failure of the primary, leading to primary replacement.

We have modified Section 3 where consensus is defined to specify that we are assuming a primary-backup consensus such as PBFT. In addition, we have restructured Section 4 to no longer need a primary and Section 5 to properly introduce the terminology we use.

Review: Some of the terminology in section 3 is not uniform. For example, you declare that "Processing of T does not imply execution: the transaction could be invalid...," but then in the "high-level three-step approach toward processing..." From section 4, you "commit T if all affected shards...", which sounds like "execution" (by your earlier definition), not processing. Indeed, processing and execution are often conflated.

...

R4: Makes it sound like a system can commit a transaction without having processed it. This doesn't sound right.

R5: "Shard-applicable" is defined only with reference to a time. Here, do you mean eventually shard-applicable? Eventually always shard-applicable?

Section 4, "The good replicas in S will first determine whether T is valid and applicable": applicable at what time? We haven't processed it yet.

We have reworded the lead-in to the requirements and the Requirements R1, R4, and R5 to avoid any confusion. We have rephrased Section 4 to emphasize that shards running CCHIMERA only verify applicability partially during the first consensus round of CCHIMERA.

Review: I am not clear on the purpose of the "Internal propagation" and "concurrent resolution" techniques after Definition 4.1. Notably, they are not used in the proof of Theorem 4.3, so they seem unnecessary.

...

Section 4: "Hence, there is the danger of deadlocks if the other shards S' never perform their crossshard exchange steps." You never explain how deadlock would be avoided if the other shards S' never perform their cross shard exchange steps. In fact, I'm pretty sure it would deadlock if the other shards S' never perform their cross shard exchange steps. Internal propagation and Concurrent resolution seem unrelated to this point.

To avoid deadlock, CChimera only requires that the other shards eventually start processing τ. As shard $\mathcal{S}$ has already determined that τ is valid, other shards will not discard τ whenever they receive a request to process τ. *Internal propagation* ensures that all affected shards $\mathcal{S}'$ receive such a request to process τ. We have rephrased the wording in the paper to emphasize this usage of internal propagation.

Review: Section 4: ": CChimera effectively eliminates all coordination when deciding to process a multi-shard transaction" No it doesn't. It requires a phase of cross-shard broadcast, before which a transaction isn't really "committed."

From a per-shard viewpoint, the order in which replicas within a single shard process a multi-shard transaction will be decided *before* any coordination and, hence, independent of whichever order the other shards choose. Only the outcome of processing the transaction are influenced by the cross-shard exchange. We have clarified this viewpoint in the text.

Review: Section 5: The use of the term "Commit" within the processing (not execution) portion of the protocol is confusing. It is different from the commit in the execution part of the protocol.

We agree with the reviewer, this is an unfortunate collision of terminology as used in Pbft (the commit phase) and in multi-shard transaction processing (commiting a transaction). Where the Pbft version of commit is used, we have changed the wording to "finalize".

Review: Figure 5, "event $R \in S$ receives messages $\langle$ VCRecoveryRQ : v_i,ρ,$S_i\rangle$, $1 \leq i \leq f_S + 1$, from $f_S + 1$ distinct replicas in S do" is confusing. Do the subscripts really need to be $1..f_S + 1$? Why not just use _ for v_i and S_i, since we don't use those indices?

At Line 10, we look at $\mathbf{f}_{\mathcal{S}} + 1$ distinct messages. Each of these messages can have distinct values for the view number v_i and the set of certificates S_i. Hence, these subscripts are used to distinguish these values. Furthermore, at Line 11, we use the value v_i explicitly.

Review: Figure 6, line 15: is the comma meant to denote "AND"? How do I read this?

To clarify this, we have reformulated all long-conditions such as those in Figure 6.

Review: Proposition 5.3: "optimistic case" is not well defined.

We have moved the definition of "optimistic case" into the proposition (from right above the proposition).

Review: Section 5: "other multi-shard consensus protocols" such as? What exactly is a multi-shard consensus protocols? Citations?

We have clarified the term multi-shard consensus protocol above Example 5.1. We already mention there several other multi-shard consensus protocols (citations 3 and 4).

Review: It seems to me that RChimera represents a 2-Phase-Commit protocol, featuring 1 consensus phase in which all resources are locked, and then one in which the locks are released with either a commit or abort. Unless I am wrong, phrasing it this way (explicitly using 2-Phase-Commit terminology) would likely make your description shorter and easier for distributed systems researchers to quickly understand.

...

Section 6: suddenly it is possible to re-construct objects in D(S,T)? We've never reconstructed objects before. Couldn't we just mark them as "in use" or something?

Marking objects as in-use during the *local inputs step* and destructing (commit) or releasing (abort) them during the *decode outcome step* would indeed convey the same usage and semantics. We did not choose for such an approach, however, in an effort to keep RChimera as close-as-possible to CChimera (whom has the same local inputs step).

As outlined above, RChimera is indeed closely related to a 2-phase-commit protocol, except that the data model of RChimera allows RChimera to forgo write locks for objects it creates.

Review: Section 7: "It is this flexibility in dealing with aborted transactions that allows all variants of CHIMERA to operate with minimal and fully-decentralized coordination between shards" It is? It's not clear why this flexibility is necessary or sufficient.

We agree with the reviewer that the statement at the end of Section 7 is rather informal. For the working of CChimera, which minimizes any form of coordination, the stated flexibility is crucial to its working: the order in which individual shards process aborted transactions can be inconsistent with any serializable order among all shards. For both OChimera and RChimera, the statement is less absolute: both protocols can be further tweaked to provide a consistent ordering of all commit and abort decisions among all shards. We have reworded Section 7 to reflect this.

Review: Figure 8: the sequential column and the cross-shard communication column are a bit confusing. CChimera requires 1 consensus step + one cross-shard communication step, while OChimera's 1 consensus step includes the 3 cross-shard communication steps.

CChimera and RChimera both only have local consensus steps (normal consensus within a individual shards). Hence, these consensus steps do not require cross-shard communication and only the explicit corss-shard exchange steps are cross-shard communication steps for these protocols. OChimera is a multi-shard consensus step that performs consensus among all replicas of all shards affected by the processed multi-shard transaction. Hence, every step of the consensus protocol used involves cross-shard. We have included a clarification by including which protocols have local consensus and which protocols have multi-shard consensus.

Review: Figure 8: OChimera ends up looking very similar to SharPer. Did you need to introduce OChimera, or could you just have used SharPer?

On the surface, OChimera and SharPer are indeed very similar: they are both multi-shard PBFT variants. With all PBFT variants, the devil is in the detail, however.

With regards to a multi-shard PBFT variant, the main three details to look for are:

1. performance: is it possible to run multiple consensus decisions simultaneously at a single shard (which correlates to out-of-order processing in a normal PBFT implementation);
2. does the multi-shard setting introduce new attack vectors for faulty replicas that can affect normal replicas; and
3. the assumptions on communication.

In both these details, OChimera behaves different than SharPer.

First, performance. The main challenge to achieve high-performance PBFT consensus in a real world environment is message delay. On the one hand, a PBFT implementation that forces a sequential communication pattern (in which messages of round r+1 are always processed after round r is finished) will see its throughput severely limited (e.g., 10-20 consensus decisions per second with a low latency of only 20ms). On the other hand, a PBFT implementations with an out-of-order design that allows for the processing of multiple rounds at once—as was part of the original PBFT paper—can maximize the network resource usage at the primary (which is the main bottleneck in such designs) and achieve thousands of consensus decisions per second, even on moderately-fast networks and with many replicas. Due to the way in which SharPer syncrhonizes/orders multi-shard transactions, the protocol has no obvious way to support out-of-order processing, while OChimera supports out-of-order processing for UTXO worloads.

Second, attack vectors. Multi-shard multi-primary protocols such as OChimera and SharPer complicate the usual failure handling of PBFT: no longer can failure of consensus be attributed to a single primary (in case of reliable communication), but now a failure to achieve consensus can be due to any of the primaries failing. Due to this, one can easily establish situations in which faulty behavior by one primary cannot be distinguished from faulty behavior by other non-faulty primaries. Hence, to avoid that faulty primaries can trigger costly primary replacement steps affecting non-faulty replicas, one needs a robust failure detection and local/global view-change protocol in place. On the one hand, SharPer does provide a failure handling description in their paper (Section 4.1 of `https://doi.org/10.1145/3448016.3452807`), the description of SharPer is very high-level and mainly focusses on primaries that corrupt messages (which are straightforward to detect and deal with—the complexity of failure recovery are omitting primaries that refuse to send messages to some replicas, as that case is indistinguishable from other replicas and/or primaries misbehaving). As important cases are not fully detailed in the SharPer paper, it is near impossible to deterine how that protocol deals with all possible situations and to verify whether the protocol addresses all possible recovery scenarios effectively. On the other hand, our OChhimera design provides a complete description of the error handling part of the protocol. making it more suited for implementation.

Third, the assumptions on communication. An important part of the design of PBFT is its ability to resynchronize nodes whenever communication becomes reliable again. This often over-looked and misunderstood part of PBFT has two components: (1) a resyncrhonization mechanism to assure that replicas that attempt a view-change, start that process at roughly the same time (in PBFT, this is based on receiving a quorum of ViewChange messages); and (2) a mechamism to assure that individual replicas eventually making message delay assumptions that are in line with the real messages delays (in PBFT, this is typically achieved with exponential backup). Without these components, PBFT-like protocols are ineffective in asynchronous environments (as these protocols still rely on time-out assumptions to detect failure). SharPer does not detail how their resynchronization mechanisms works while, as any PBFT-like protocol, heavily depending on time-outs to detect any issues. Hence, without further details, SharPer can only truly operate in synchronous environments. OChimera, on the other hand, fully details all recovery mechanisms and will be suited for asynchronous environments.

We already included a highly-summarized version of the above in the original discussion of SharPer in Section 8.2. To further clarify the differences between SharPer and OChimera, we have slightly expanded this discussion in Section 8.2. (We do not believe that a full analysis of SharPer is suitable for the paper, hence, we only included a summary).

> **Review:** Figure 8: What does the word "Incentive" mean here?

As discussed in Section 8.1, objects affected by aborted transactions in CChimera remain pledged and cannot be reused. In practice, this loss of objects can provide an incentive for clients to not behave malicious. To aid the reader, we have added this explanation of the term incentive as a footnote to Figure 8.

> **Review:** Figure 9: With large numbers of objects, shouldn't most multi-shard transactions use all shards when there are only a few shards?
>
> ...
>
> Figure 9: Are you picking objects uniformly at random? It's not clear how many shards each transaction touches.

We agree with the reviewer that the description of the workload can be strongly improved, as it did not accurately and completely describe what was going on (especially in the case of a single shard). We have changed the wording as follows: "50% of the transactions are single-shard and the remaining transactions affect objects chosen fully-at-random. Hence, when multiple shards are present in the system, the remaining transactions are likely to be multi-shard transactions."

> **Review:** Figure 10: Why is AHL not in this figure?

For completeness, we have added both Chainspace and AHL. We also expanded the analysis of this figure in the experiment description.

> **Review:** Also, how does OChimera ever beat CChimera? It the underlying consensus even slower than OChimera's PBFT variant?

All protocols are bottlenecked by the communication at the primaries of shards (as these primaries broadcast proposal messages, which are much larger than any other messages).

In *general*, the multi-shard consensus of OChimera makes a trade-off in favor of latency at the expense of *higher communication costs* at the primaries of shards than the higher-latency combination of local consensus and cross-shard exchange used by CChimera. This is not always true, however. To further minimize latencies for CChimera and RChimera, we used a primary-initiated one-phase broadcast-based cross-shard exchange step that will succeed in a single communication phase whenever the primary in a sending shard is non-faulty and communication is reliable. Although this cross-shard exchange step has low latency and can deal with failures, it does impose a high cost at the primary in the sending cluster. Due to this, we see that when multi-shard transactions only affect a few shards (e.g., very few shards or very small transactions), then the overall communication costs of the multi-shard consensus performed by OChimera is slightly lower for each involved primary than the combined cost of a local consensus and cross-shard exchange step as performed by CChimera. Consequently, we see slightly higher throughput for OChimera than for CChimera in these settings.

We have added these clarifications to the paper by detailing the cross-shard exchange step used and by including the above analysis.

> **Review:** Figure 11: "Step" here is not defined until the bottom of the page, and even then it's defined implicitly in the parenthetical "(for CCHIMERA and OCHIMERA, this is equivalent to the number of per-shard consensus steps, for RCHIMERA this is half the number of per-shard consensus steps)."

We have added a clarification to the caption.

> **Review:** Section 8: "In environments in which inter-shard and intra-shard communication have similar (high) message delays, OCHIMERA will typically have lower latencies than CCHIMERA due to the large impact message delays have on the latency of consensus and cluster-sending steps, this even when CCHIMERA has higher throughput than OCHIMERA." I don't follow. How many message delays does each have? Are they different?

They are different. As summarized in Figure 8. On the one hand, OChimera only performs a PBFT-like multi-shard consensus, which takes *three* consecutive phases of communication. On the other hand, CChimera performs both a local consensus step and a cross-shard exchange step. Assuming the local consensus step is performed via PBFT, this results in *four* consecutive phases of communication (three for PBFT, one for the cross-shard exchange step). As this nuance is easy to miss and essential to understanding the performance differences of the Chimera protocols, we have included the above analysis also in text.

> **Review:** Figure 12: "Performance" did you mean "throughput"?

In both Figure 9 and 12, we used the plot title "performance" when discussing throughput. In both cases, the throughput is already mentioned on the y-axis. Hence, we have simplified the plot titles to no longer include "performance" and, in the case of Figure 9, only mention the size of transactions.

> **Review:** Figure 12: "malicious shards" did you mean "shards with a malicious participant"?

We have changed the wording in Figure 12 and the conclusion of Section 8.4 to clearly clarify what kind of malicious behavior is measured and what the effects of this behavior is.