Pilotfish: Distributed Execution for Scalable Blockchains

Quentin Kniep[1], Lefteris Kokoris-Kogias[2,3], Alberto Sonnino[3,4], Igor Zablotchi[3], and Nuda Zhang[5]

[1] ETH Zurich qkniep@ethz.ch
[2] IST Austria
[3] Mysten Labs {lefteris,alberto,igor}@mystenlabs.com
[4] University College London (UCL)
[5] University of Michigan nudzhang@umich.edu

Abstract. Scalability is a crucial requirement for modern large-scale systems, enabling elasticity and ensuring responsiveness under varying load. While cloud systems have achieved scalable architectures, blockchain systems remain constrained by the need to over-provision validator machines to handle peak load. This leads to resource inefficiency, poor cost scaling, and limits on performance. To address these challenges, we introduce Pilotfish, the first scale-out transaction execution engine for blockchains. Pilotfish enables validators to scale horizontally by distributing transaction execution across multiple worker machines, allowing elasticity without compromising consistency or determinism. It integrates seamlessly with the lazy blockchain architecture, completing the missing piece of execution elasticity. To achieve this, Pilotfish tackles several key challenges: ensuring scalable and strongly consistent distributed transactions, handling partial crash recovery with lightweight replication, and maintaining concurrency with a novel versioned-queue scheduling algorithm. Our evaluation shows that Pilotfish scales linearly up to at least eight workers per validator for compute-bound workloads, while maintaining low latency. By solving scalable execution, Pilotfish brings blockchains closer to achieving end-to-end elasticity, unlocking new possibilities for efficient and adaptable blockchain systems.

1 Introduction

A crucial property required by modern large-scale computing is *scalability* [14], which refers to a system's ability to dynamically adapt its performance as load changes, ensuring that the system remains responsive despite varying load. Scalability is fundamental because it is an essential requirement for elasticity [35], and thus in turn for a good user experience (e.g., responsiveness) at a sustainable cost. Without elasticity, systems either risk being overwhelmed during peak loads, leading to poor performance and user dissatisfaction, or they incur excessive costs during low-load periods by maintaining unnecessary resources.

Over the past decades, significant effort has been devoted to developing scalable software architectures for cloud-based systems [5]. However, the situation is starkly different for blockchain systems. Among core blockchain tasks, transaction execution is particularly challenging with respect to scalability. The current

dominant approach to transaction execution in blockchain involves ensuring that validator machines are sufficiently powerful to handle peak loads [8,53,58]. This approach is scalable up to a point, but has limitations: (1) it leads to resource inefficiency, as validators remain over-provisioned during low-load periods; (2) it has a resource ceiling, as even the most powerful single machine will eventually be insufficient if the load is high enough; (3) it has poor cost scaling, as high-end machines are expensive and limited to a few vendors.

In response to these challenges, we introduce Pilotfish, the first scale-out transaction execution engine for blockchain. The core idea of Pilotfish is to run each validator on multiple mutually trusting machines or workers, as opposed to running a single machine per validator. Each worker is only responsible for a subset of the validator's state, and only executes a subset of transactions. This approach opens the way toward elasticity, as it allows scaling each validator out and in as the load increases and decreases.

Pilotfish is designed to integrate seamlessly with the lazy blockchain architecture, which is increasingly used by modern blockchains [3,11,12,15,20,30,32, 44,45,56]: as of the time of writing, lazy blockchains account for over $20 billion in market capitalization [18].Lazy blockchains separate the problems of transaction dissemination, ordering, and execution. They provide a scalable solution to two of the three core blockchain tasks: dissemination (ensuring that client transactions are available at a quorum of validators) and ordering (establishing a reliable total order over transactions, also known as consensus). However, as mentioned above, state-of-the-art lazy blockchains do not solve the *execution scalability* problem: their execution is still designed to run on a single machine.

Pilotfish must address several challenges to achieve this. First (i), it must solve the distributed transaction problem, since the validator state is sharded across multiple worker machines, and transactions may span multiple shards. This is especially challenging since blockchains need to guarantee strong consistency (serializability) and determinism, without compromising on latency or throughput. Most existing approaches to distributed transactions cannot directly be applied to our setting: (1) the two-phase commit approach [41] guarantees strong consistency but is not scalable; (2) the relaxed consistency approach [19, 38] is scalable but sacrifices strong consistency, which is crucial for blockchain; (3) the restricted transaction approach [2, 22] is both scalable and strongly consistent, but sacrifices transaction generality. The most promising existing solution for our needs is that of deterministic databases [60], which balance scalability, consistency, and transaction generality. We borrow techniques from distributed databases and leverage the fact that in lazy blockchains, consensus precedes—and is decoupled from—execution, so by execution time, validators have agreed on a permanent ordering of transactions.

Secondly (ii), Pilotfish needs to tolerate workers crashing and recovering. To address this, Pilotfish maintains sufficient state among workers as *checkpoints* to allow recovering machines to catch up with the rest. A straightforward solution would be to resort to strong (and expensive) consensus-based replication techniques among workers internal to the validator [39, 46, 60]. However Pilotfish avoids such overhead by observing that consistency and availability of the commit sequence are already provided by the blockchain protocol. Thus, Pilot-

fish optimistically does lightweight, best-effort replication between workers, and relies on recovery from other validators only if optimistic replication fails.

Finally (iii), Pilotfish aims to support a simpler programming model where transactions may only partially specify their input read and write set (e.g., as required for Move [43]). This, however, creates an additional challenge for Pilotfish, as objects that might be accessed dynamically at execution time can be located in different workers. This means that objects cannot be overwritten until all previous transactions have finished, effectively reverting to sequential execution and enforcing write-after-write dependencies. This limitation would reduce the parallelizability of the workload. Pilotfish circumvents this issue by leveraging its enforced determinism, allowing in-memory execution to be lost and safely recovered in the event of crashes. Pilotfish relies on a novel versioned-queue scheduling algorithm that allows transactions with write-after-write conflicts to execute concurrently. We couple this with our crash recovery mechanism, which only persists consistent states. As a result, upon a crash, Pilotfish simply re-executes a few transactions, but thanks to the deterministic nature of the blockchain this does not pose any inconsistency risks.

We evaluate Pilotfish by studying its latency and throughput, while varying the number of workers per validator, the computational intensity, and the degree of contention of the workload. We find that Pilotfish scales linearly to at least 8 workers per validator when the workload is compute-bound, while keeping latency under 50 ms.

Discussion. While this work focuses on a scalable protocol for distributed blockchain transactions, achieving full elasticity poses additional challenges, particularly in dynamically scaling up and down workers and repartitioning objects. These aspects, while critical to practical implementations, are well-explored in existing literature on elastic systems: dynamic workload partitioning [23, 51], load-aware worker scaling [10, 51], and online object migration [19, 51].

2 System Model

Pilotfish implements a blockchain *validator*, composed internally of a black-box *Primary* machine, as well as a set of worker machines, simply called *workers*. The Primary is responsible for communicating with (the Primaries of) other validators in order to agree on an ordered sequence of transactions. The workers collectively execute the ordered sequence of transactions and update the validator's state accordingly.

Objects and Transactions. Pilotfish validators replicate the state of the blockchain represented as a set of *objects* [57]. Transactions can read and write (mutate, create, and delete) objects, and reference every object by its unique identifier *oid*. A transaction is an authenticated command that references a set of objects (by their unique identifier *oid*), and an entry function into a smart contract call identifying the execution code. The transaction divides the objects it references into two disjoint sets, (i) the read set $\mathcal{R}$ referencing input objects that the transaction may only read, and (ii) the write set $\mathcal{W}$ referencing objects that the transaction may mutate. In most cases, the identifier *oid* of each object of the read and write sets can be computed using only the information provided by the transaction, without the need to execute it or access any object's data.

In these cases, Pilotfish has complete knowledge of the read and write sets of the transaction. However, Pilotfish also supports dynamic accesses (Section 6) where the read and write set of a transaction is discovered only upon attempting to execute the transaction, adopting the execution model of Sui [13].

Network Model. We assume that the Primary and workers communicate by sending messages over the network through point-to-point connections. We assume that the network is fully connected and reliable: each message sent by a correct process (i.e., non-faulty machine) to a correct process is eventually delivered. Furthermore, we assume authenticated channels: the receiver of a message is aware of the sender's identity.

Synchrony Model. We consider the standard partially synchronous environment [25]. Specifically, there exists an unknown Global Stabilization Time (GST) and a positive known duration δ such that message delays are bounded by δ after GST: a message sent at time τ is received by time $\max(\tau, \text{GST}) + \delta$. It has been shown that in partial synchrony, crash failures can eventually be perfectly detected [1, 17], thus we assume an eventually perfect failure detector.

Threat Model. We assume that each validator is controlled by a single entity, or by a set of mutually trusting entities. This implies that the Primary and workers trust each other, and we therefore only consider crash failures for components internal to the validator (a validator as a whole may still exhibit Byzantine behavior in its interaction with other validators, but tolerating such failures is handled by the blockchain protocol, which is outside the scope of this work). For this reason, we do not require any cryptography assumptions, other than the authenticated channels.[6] For pedagogical reasons, for the first part of the paper we assume that workers cannot fail. Later in Section 5, we expand each logical worker to have a set of $n_e = 2f_e + 1$ replicas such that as long as for each worker there are $f_e + 1$ replicas available the system remains live and safe.[7] In case this threshold is breached the validator can still synchronize with the rest of the validators of the lazy blockchain through a standard recovery procedure [20] that is out of scope.

Core Properties. Appendix B proves that Pilotfish guarantees serializability, determinism, and liveness. Intuitively, serializability means that Pilotfish execution produces the same result as a sequential execution. Determinism means that every correct validator receiving the same sequence of transactions performs the same state transitions. Liveness means that all correct validators receiving a sequence of transactions eventually execute it.

Definition 1 (Pilotfish Serializability). *A correct validator executing the sequence of transactions* $[\mathsf{Tx}_1, \ldots, \mathsf{Tx}_n]$ *holds the same state as if the transactions were executed sequentially, in the given order.*

[6] Our network, synchrony, trust and cryptography assumptions only apply *internally* to the validator. By contrast, the outer blockchain protocol, which governs how validators interact with each other, may make entirely different assumptions on synchrony and types of failures and thus may require stronger cryptography primitives.

[7] Here, f_e refers to the number of replicas that may crash per logical worker, *internally to the validator*; in particular, f_e may be different from the number of validators in the blockchain that may be Byzantine (usually denoted by f).

Definition 2 (Pilotfish Determinism). *No two correct validators that executed the same sequence of transactions* $[Tx_1, \ldots, Tx_n]$ *have different states.*

Definition 3 (Pilotfish Liveness). *Correct validators receiving the sequence of transactions* $[Tx_1, \ldots, Tx_n]$ *eventually execute all transactions* $Tx_1, \ldots, Tx_n$.

3 Existing Designs & Pilotfish Overview

3.1 Previous Designs

Previous designs for scaling execution in lazy blockchains fall into two categories. The first is parallel execution [13, 31, 47], where each validator uses a high-end server to handle increased load. This approach lacks elasticity: the cost of running a powerful validator remains high regardless of actual load, leading to inefficiency during low usage and performance ceilings due to finite server resources.

The second category employs inter-validator sharding [4, 6, 9, 21, 33, 34, 37, 54, 56], in which the blockchain state is split into shards, with a subset of the validators handling each shard in parallel. However, inter-validator sharding has limitations related to security and performance. Firstly, sharding requires a sampling process from the full validator set to subsets of validators per shard, such that each shard has more than 2/3 honest members. These systems are thus less robust to adversarial attacks. For example Omniledger [37] assumes a 25% Byzantine adversary in order to provide sufficient 34% security in all the subsampled shards. In the same vein, the adversary's adaptivity should be limited to once an epoch, as otherwise the adversary could target all its power in a single shard and compromise it. Finally, sharding is also challenging from a performance perspective, as transactions that span multiple shards require expensive and slow Byzantine-resilient atomic commit protocols [54].

3.2 Intravalidator Sharding with Pilotfish

Through Pilotfish, we instead propose *intravalidator sharding*, as illustrated in Figure 1. Each validator consists of multiple *SequencingWorkers* that collect transaction data based on the commit sequence from the Primary, similar to transaction dissemination workers in lazy blockchains like Tusk [20], Bullshark [32], and Shoal [55]. Pilotfish innovates by distributing transaction execution on several *ExecutionWorkers*. Each ExecutionWorker stores a subset of the state, executes a subset of the transactions, and contributes its memory and storage to the system.

In Pilotfish, the Primary only manages metadata (agreement on a sequence of batch digests) allowing it to scale to large volumes of batches and transactions [20]. Actual batch storage is distributed among a potentially large number of SequencingWorkers. The key insight is that transaction execution is also distributed among numerous ExecutionWorkers, enabling horizontal scaling. As workers are added, the capacity to store state and process transactions increases.

Sharding strategy. Pilotfish uses its SequencingWorkers and its ExecutionWorkers to operate two levels of sharding. (i) Pilotfish shards transaction data among its SequencingWorkers. Transactions batches (and thus clients' transactions) are assigned to SequencingWorkers deterministically based on their digest.

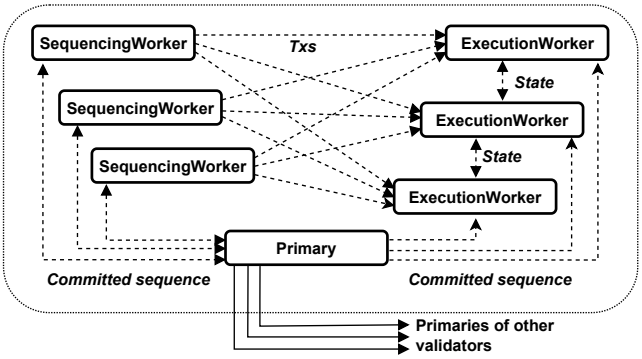


Fig. 1: Pilotfish validator's components. Each validator is composed of several SequencingWorkers to fetch and persist the client's transaction, one Primary to run Byzantine agreement on metadata, and several ExecutionWorkers to execute transactions. Each component may run on dedicated machines or be collocated with other components. Dotted arrows indicate internal messages exchanged between the components of the validator (localhost or LAN) and solid arrows indicate messages exchanged with the outside world (WAN).

SequencingWorker can be seen as architecturally equivalent to the worker machines used by lazy blockchains to decouple dissemination (performed by workers) from ordering (performed by the Primary). All transactions of a batch are persisted by the same SequencingWorker. Each SequencingWorker maintains a key-value store $\textsc{Batches}[\mathsf{BatchId}] \to \mathsf{Batch}$ mapping the batch digests $\mathsf{BatchId}$ to each batch handled by the SequencingWorker. (ii) Additionally, Pilotfish shards its state among its ExecutionWorkers. Each ExecutionWorker is responsible for a disjoint subset of the objects in the system (composing the state); objects are assigned to ExecutionWorkers based on their collision-resistant identifier *oid*. Every object in the system is handled by exactly one (logical) ExecutionWorker.

4 The Pilotfish System

Figure 2 shows the transaction life cycle in Pilotfish, from sequencing to execution. The Primary sends the committed sequence to all SequencingWorkers and ExecutionWorkers (❶). Below, we outline the core Pilotfish protocol at steps ❷, ❸, ❹, and ❺ of Figure 2. Appendix A presents the full algorithms.

ExecutionWorkers maintain the following key-value stores:

- $\textsc{Objects}[oid] \to o$ making all the objects handled by the ExecutionWorker accessible by their unique identifier.
- $\textsc{Pending}[oid] \to [(op, [\mathsf{Tx}])]$ mapping each object to a list of pending transactions $[\mathsf{Tx}]$ referencing *oid* in their read or write set and that are awaiting execution. The operation *op* indicates whether the transaction may only Read (R) the object or whether it may also write (W) it. This map is used as a 'locking' mechanism to track dependencies and determine which transactions can be executed in parallel. Entries relating to a transaction are removed from this map after its execution.
- $\textsc{Missing}[oid] \to [\mathsf{Tx}]$ mapping objects that are missing from $\textsc{Objects}$ to the transactions that reference them. It is used to track transactions that cannot

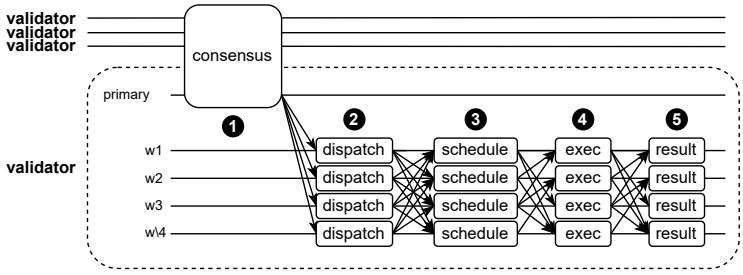


Fig. 2: Pilotfish overview. Every validator runs with 5 machines: one machine running the Primary and 4 machines running workers. Each worker machine collocates 1 SequencingWorker and 1 ExecutionWorker. The Primary runs a Byzantine agreement protocol to sequence batch digests (❶). SequencingWorkers receive the committed sequence and load the data of the corresponding transactions from their storage (❷). Each ExecutionWorkers receiving these transactions assigns a lock to each object referenced by the transaction to schedule their execution (❸). A deterministically-selected ExecutionWorker eventually receives the object's data referenced by the execution and executes it (❹). Finally, the ExecutionWorker signals all SequencingWorkers to update their state with the results of the transaction's execution (❺).

(yet) be executed because they reference objects that are not yet available. It is cleaned after execution.

Step ❷: Dispatch transactions. At a high level, each SequencingWorker i observes the commit sequence and loads from storage all the batches referenced by the committed sequence that they hold in their $\textsc{Batches}_i$ store (and ignores the others). The SequencingWorker then parses each transaction of the batch (in the order specified by the batch) to determine which objects it contains. At the end of this process, SequencingWorker i composes one ProposeMessage for each ExecutionWorker j of the validator: $\mathsf{ProposeMessage}_{i,j} \leftarrow (\mathsf{BatchId}, \mathsf{BatchIdx}, T)$. The message contains the batch digest BatchId, an index BatchIdx uniquely identifying the batch in the global committed sequence and a list of transactions T referencing at least one object handled by worker j. If no transactions affect worker j, the worker still receives an empty message so it can proceed.

Step ❸: Schedule execution. Each ExecutionWorker j awaits one ProposeMessage from each SequencingWorker. It then parses every transaction Tx included (in order) and extracts objects in Tx's read set $\mathcal{R}_j$ and write set $\mathcal{W}_j$ managed by ExecutionWorker j (and ignores the other objects that it does not handle).

Figure 3 illustrates an example snapshot of the $\textsc{Pending}_j$ store of a validator. ExecutionWorkers append every object of the write set $\mathcal{W}_j$ to their local $\textsc{Pending}_j$ indicating that Tx may mutate *oid*: $\textsc{Pending}_j[oid] \leftarrow \textsc{Pending}_j[oid] \cup (\mathsf{W}, \mathsf{Tx})$. The position of Tx in the $\textsc{Pending}_j$ indicates that Tx can only write *oid* after all transactions appended before in $\textsc{Pending}_j[oid]$ are executed, essentially indicating a write-after-write (or write-after-read) dependency.

ExecutionWorkers additionally register reads performed by Tx on an object id by looking at the latest entry in $\textsc{Pending}_j[oid]$. If the entry is a write then they append a new entry: $\textsc{Pending}_j[oid] \leftarrow \textsc{Pending}_j[oid] \cup (\mathsf{R}, \mathsf{Tx})$, indicating a read-after-write dependency. However, if the entry is a read then the transaction

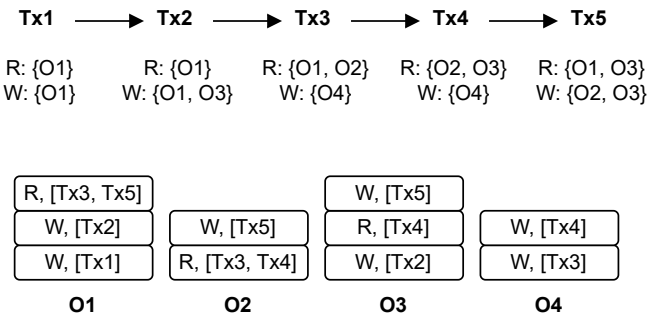


Fig. 3: Example snapshot of the PENDING queues of an ExecutionWorker. Pilotfish schedules the execution of the sequence $[\mathsf{Tx}_1, \mathsf{Tx}_2, \mathsf{Tx}_3, \mathsf{Tx}_4, \mathsf{Tx}_5]$. The ExecutionWorker stores Tx_1 as $(\mathsf{W}, [\mathsf{Tx}_1])$ in the queue of oid_1 as it only mutates oid_1. Tx_2 then mutates oid_1 and writes oid_3; it is thus store in the queue of oid_1 (implicitly taking Tx_1 as dependency) and oid_3. Tx_3 schedules a read for both oid_1 and oid_2 and a write for oid_4. Tx_4 reads oid_2 (it can thus read oid_2 in parallel with Tx_3, registering $(\mathsf{R}, [\mathsf{Tx}_3, \mathsf{Tx}_4])$ in the queue of oid_2) and oid_3, and writes oid_4. Finally Tx_5 reads oid_1 (it can thus read oid_1 in parallel with Tx_3), writes oid_2 and mutates oid_3.

Tx may be executed in parallel with any other transaction Tx' also reading oid. ExecutionWorkers thus modify the latest entry of the storage to reflect this possibility by setting Tx and Tx' at the same height in the PENDING_j store: $\text{PENDING}_j[oid][-1] \leftarrow (\mathsf{R}, [\mathsf{Tx}', \mathsf{Tx}])$.

A transaction Tx is ready to be executed when it reaches the head of the pending lists of all the objects it references. At this point, the ExecutionWorker loads from its OBJECTS_j store all the objects data it handles: $O_j \leftarrow \{\text{OBJECTS}[oid]$ s.t. $oid \in \text{HANDLEDOBJECTS}(\mathsf{Tx})\}$. It then composes a ReadyMessage for the dedicated ExecutionWorker that was selected to execute Tx: $\mathsf{ReadyMessage}_j \leftarrow (\mathsf{Tx}, O_j)$. The message contains the transaction Tx to execute, and a list of object data (O_j) referenced by the part of the read and write set of Tx handled by ExecutionWorker j.

If an object referenced by Tx is absent from the ExecutionWorker's local OBJECTS_j store, the ExecutionWorker waits until it all transactions sequenced before Tx are executed and then sends $\perp$ instead of the object's data. This signals that Tx is malformed and references non-existent objects or objects that should have been created but the origin transaction failed.

Step ❹: Execute transactions. Upon receiving a ReadyMessage message, an ExecutionWorker waits for one ReadyMessage from all other ExecutionWorkers handling at least one object referenced by Tx. At this point, the set of ReadyMessage provides the ExecutionWorker with the objects' data behind all objects referenced by Tx (or $\perp$ if missing). If all object data are available, Tx is executed; otherwise, it is aborted. Executing a transaction produces a set of objects to mutate or create O and a set of object ids to delete I: $(O, I) \leftarrow exec(\mathsf{Tx}, O')$. The ExecutionWorker then prepares a ResultMessage for all ExecutionWorkers. For ExecutionWorkers whose objects are not affected by Tx this serves as a heartbeat message whereas for those whose objects are mutated, created or deleted by

the transaction execution it informs them to update their object store Objects accordingly. If Tx aborts, the worker sends a ResultMessage with empty O, I.

Step ❺: Handle results. When an ExecutionWorker receives a ResultMessage, it: (i) persists locally the fact that the transaction has been executed by advancing a watermark keeping track of all executed transactions; (ii) updates each object into its local Objects store including deletions; and (iii) removes all occurrences of the transaction from its Pending store. It then tries to trigger the execution of the next transactions in the queues.

5 Crash Fault Tolerance

Section 4 presents the design of Pilotfish assuming all data structures are in-memory. However, critical validator components inevitably fail over time. To handle this, Pilotfish adopts a simple replication architecture, dedicating multiple machines to each ExecutionWorker. This internal replication allows the validator to continue operating despite crash faults. Pilotfish does not replicate the Primary, which handles only lightweight operations (and holds the signing key), nor does it replicate SequencingWorkers, which perform stateless work and can be rebooted from the latest persisted sequence number. We briefly detail our replication protocol here and defer more details to Appendices C and D.

5.1 Internal Replication

Figure 4 illustrates the replication strategy of Pilotfish. Each ExecutionWorker is replaced by $n_e = 2f_e + 1$ ExecutionWorkers. Pilotfish tolerates up to f_e simultaneous crash faults in a set of n_e replicated ExecutionWorkers. These replicas form a grid: each column represents replicas of a single shard; each row is a cluster containing exactly one replica from each shard. Within each cluster, workers exchange reads and maintain a consistent view of the object store.

The naïve way to achieve such reliability would be to run a black-box replication engine like Paxos [39] which is also the proposal of the state-of-the-art [60]. Pilotfish however greatly simplifies this process by leveraging (i) the Primary as a coordinator between the workers' replicas, (ii) external validators holding the blockchains state and the commit sequence, and (iii) the fact that execution is deterministic (given the commit sequence).

5.2 Normal Operation

Within each cluster, replicated ExecutionWorkers run the same core protocol as the unreplicated case. Inter-cluster communication is minimal, except for checkpoint updates. To enable recovery, each worker keeps: (1) a buffer of outgoing ReadyMessage instances (the reads it has served), which can be replayed if messages are lost, and (2) a set of checkpoints, each representing a consistent, on-disk snapshot of the local object store. Checkpoints are the only persistent state.

Garbage collection. Once a checkpoint is deemed stable—i.e., a quorum of $f_e + 1$ replicas in every shard confirm they have persisted a checkpoint after a certain transaction index—old checkpoints and buffered messages prior to that index are garbage-collected.

Bounding memory use. Even with garbage collection, differences in execution speeds of workers can prevent checkpoints from being safely garbage-collected,

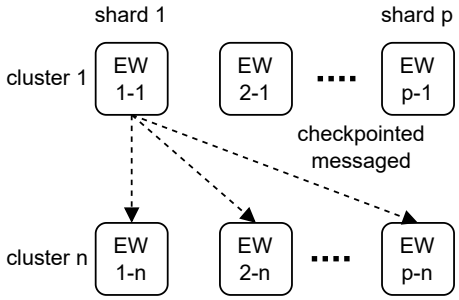


Fig. 4: Replication scheme for ExecutionWorkers. The object store is partitioned into shards, and each shard is replicated n_e-fold. Each row represents a cluster, and ExecutionWorkers within a cluster coordinate to process transactions. During normal operation, the only communication between clusters is the sending of checkpoints.

and thus lead to unbounded memory use. To prevent this, Pilotfish enforces a maximum of c checkpoints per worker. If a worker reaches this limit (e.g., $c = 2$), it must pause execution until it can safely discard an older checkpoint. This prevents unlimited checkpoint buildup and ensures that clusters can proceed without one shard outpacing the others indefinitely. Typically, $c = 2$ strikes a balance between performance and resource usage, letting faster clusters keep going as long as they are within one checkpoint boundary of slower ones.

5.3 Failure Recovery

Pilotfish uses two mechanisms to recover from failures: (1) *reconfiguration*, a rapid process that does not reduce the system's throughput, but requires roughly synchronized clusters; and (2) *checkpoint synchronization*, a slower procedure that coordinates multiple clusters if reconfiguration fails. If both approaches fail, the system can still recover from other blockchain validators.

Recovery through reconfiguration. When an ExecutionWorker crashes, workers which rely on it for reads may be unable to proceed. They detect the crash and establish a new connection with a replacement replica. Typically, this requires just two round trips: one to identify a new node that can provide reads, and another to complete the handshake. Other clusters keep executing, so throughput remains unaffected as long as the failure threshold f_e is not exceeded. Appendix D contains the full algorithm.

Recovery through checkpoint synchronization. If a worker is too far behind for reconfiguration alone, it triggers a synchronization process for itself and its peers, which fetch the latest checkpoint from an up-to-date replica. Once all peers reach the same state, the worker re-establishes missing members in its cluster (via reconfiguration). This cascades recovery across clusters that depend on the slow worker, ensuring no cluster loses liveness if another shard "fast-forwards" its state.

Disaster recovery. If an entire cluster is lost beyond the threat model of Section 2, the system can recover by booting a new cluster with the same peers set. This new cluster retrieves the system state from other validators, which

store stable checkpoints. Though it requires wide-area network communications and is slower, this worst-case path ensures Pilotfish remains operable even with minimal replication (e.g., $f_e = 1$).

6 Dynamic Reads and Writes

In most deterministic execution engines [27,29,47,60], transactions must specify the exact data they read and write. This constraint limits developers and encourages the over-prediction of read/write sets to ensure successful execution. In distributed execution, the problem is exacerbated by the need to transmit the data between ExecutionWorkers. This means that we might need to transmit large read/write sets between computers in order to access a single item (e.g., transfer a full array to dynamically access one cell).

Pilotfish supports dynamic reads/writes but confines them to parent-child object hierarchies. A child object is an object that is owned by another object, the parent. An example parent-child relationship is that between a dynamically allocated array and its individual cells.

In Pilotfish, a child object can only be accessed if the root object (the top-level object in a hierarchy of potentially numerous parents) is included in the transaction and the transaction has permission to access the root. This setup avoids overpredictions by allowing transactions to handle unexpected data accesses with minimal algorithmic changes.

One of the required modifications is to retain the reads in the queues until the transaction execution is completed. However, this leads to a loss of parallelism since we are unable to write a new version of an object until all transactions reading the previous version have finished. We resolve this false sharing situation without bloating memory usage in two ways. First, we treat every version of an object as a new object; this means that the queues in Figure 3 are per $(oid, \mathsf{Version})$ instead of per oid. Therefore, each queue consists of a single write as the initial transaction, followed by potentially several reads. This resolves the false sharing as future versions of an object initialize new queues and can proceed independently of whether the previous version is still locked because of a dynamic read operation. Unfortunately, this leads to objects potentially being written out of order, which could pollute our state and make consistent recovery from crashes impossible. For this reason, our second modification is buffering writes so that they are written to disk in order by leveraging the crash-recover algorithm in Section 5. Appendix E provides further details on how we handle child objects, complete algorithms, and formal proofs.

Algorithm modifications. Pilotfish handles the state of child objects like any other object: they are assigned to ExecutionWorkers that maintain their pending queues. The ExecutionWorkers schedule the execution of root objects as usual after processing a ProposeMessage (Algorithm 2) by updating the queues of all the objects that the transaction directly references. This means that they update the queues of (potentially) root objects as well as the queues of (potentially currently undefined) child objects. The security of this process is ensured by following the same procedure as for object creation. Hence, the ExecutionWorker will either create these objects or garbage-collect them. Finally, when the transaction is ready for execution, either a previous transaction would have

transferred ownership of child objects to the parent or the transaction would abort at execution.

On receiving a ReadyMessage, the ExecutionWorker starts execution. If it detects a new child object, it pauses and sends a UpdateProposeExec carrying an *augmented transaction* Tx+, which includes the child objects ID in its read/write sets. This UpdateProposeExec message is sent to shards handling one of the (newly discovered) child objects. This is safe because the parent is already locked, implicitly locking the child. If the transaction is not done, subsequent parent writes go to distinct queues, enabling on-demand multi-version concurrency.

Upon receiving UpdateProposeExec with Tx+ , the ExecutionWorker replaces Tx in its queues with Tx+ and adds Tx+ to the queues of any newly discovered child objects. When Tx+ reaches the front of every involved queue, it re-attempts execution. Eventually, the protocol identifies every child object that the transaction dynamically accesses, and Tx+ contains their explicit ids. At this point, the transaction execution can terminate successfully.

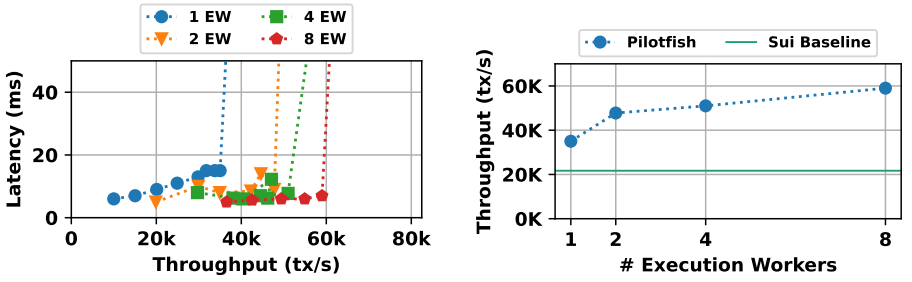
7 Implementation

We implement a networked multi-core Pilotfish execution engine in Rust on top of the Sui blockchain [42]. As a result, our implementation supports Sui-Move [43]. We made this choice because Sui-Move is a simple and expressive language that is easy to reason about, provides a well-documented transaction format explicitly exposing the input read and write set, and supports dynamic reads and writes. Our implementation uses `tokio` [62] for asynchronous networking across the Pilotfish workers, utilizing low-level TCP sockets for communication without relying on any RPC frameworks. While all network communications in our implementation are asynchronous, the core logic of the execution worker runs synchronously in a dedicated thread. This approach facilitates rigorous testing, mitigates race conditions, and allows for targeted profiling of this critical code path. In addition to regular unit tests, we created a command-line utility (called *orchestrator*) designed to deploy real-world clusters of Pilotfish with workers distributed across multiple machines. The orchestrator has been instrumental in pinpointing and addressing efficiency bottlenecks. We will open-source our Pilotfish implementation along with its orchestration utilities.[8].

8 Evaluation

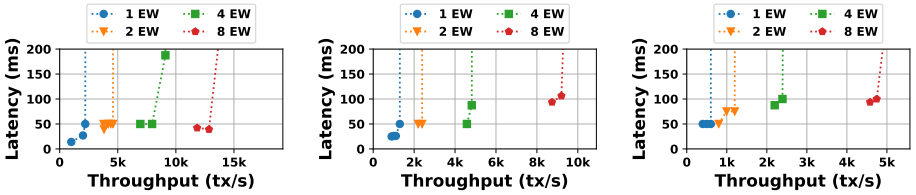
We evaluate the performance of Pilotfish through experiments on Amazon Web Services (AWS) to show that given a sufficiently parallelizable compute-bound load, the throughput of Pilotfish linearly increases with the number of ExecutionWorkers without visibly impacting latency. In order to investigate the spectrum of Pilotfish, we (a) run with transactions of increasing computational load and (b) create a contented workload that is not ideal for Pilotfish as it (i) increases the amount of communication among ExecutionWorkers and (ii) might increase the queuing delays in order to unblock later transactions. We show the performance improvements of Pilotfish over the baseline execution engine of Sui [42].

[8] `https://github.com/mystenlabs/sui/tree/sharded-execution`



(a) (b)

Fig. 5: Pilotfish latency vs throughput (a) and scalability (b) with simple transfers.



(a) `Fib-2500` (b) `Fib-5000` (c) `Fib-10000`

Fig. 6: Pilotfish latency vs. throughput for the heavy computation workloads.

8.1 Experimental Setup

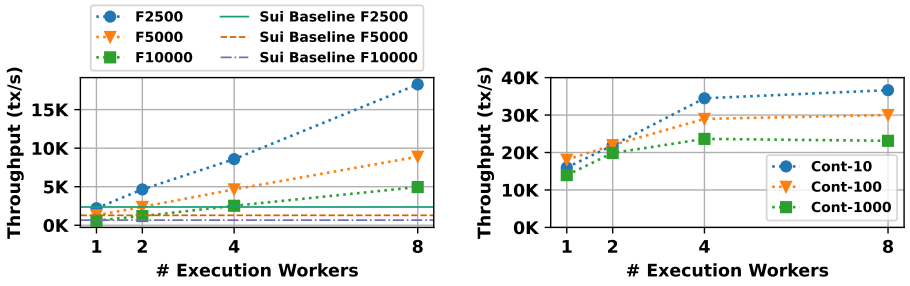
We deploy Pilotfish on AWS, using `m5d.8xlarge` within a single datacenter (us-west-1). Each machine provides 10 Gbps of bandwidth, 32 virtual CPUs (16 physical cores) on a 2.5GHz, Intel Xeon Platinum 8175, 128GB memory, and runs Linux Ubuntu server 22.04. We select these machines because they provide decent performance, and are in the price range of 'commodity servers'.

In all graphs, each data point represents median latency/throughput over a 5-minute run. We instantiate one benchmark client collocated with each SequencingWorker submitting transactions at a fixed rate for a duration of 5 minutes. We experimentally increase the load of transactions sent to the systems, and record the throughput and latency of executed transactions. As a result, all plots illustrate the 'steady state' latency of all systems under low load, as well as the maximal throughput they can serve, after which latency grows quickly. We vary the types of transactions throughout the benchmark to experiment with different contention patterns.

When referring to *latency*, we mean the time elapsed from when the client submits the transaction until the transaction is executed. By *throughput*, we mean the number of executed transactions over the entire duration of the run.

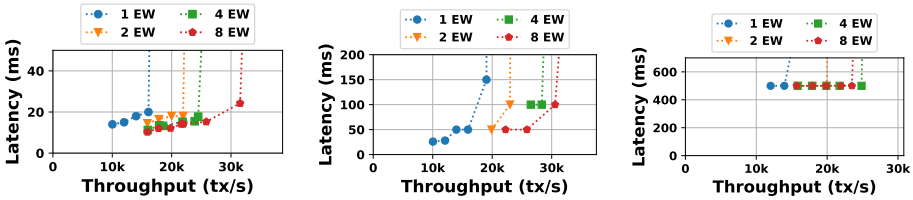
8.2 Simple Transfer Workload

In this workload, each transaction is a simple transfer of coins between objects. No two transactions conflict; each transaction operates on a different set of objects from the other transactions. Thus, this workload is completely parallelizable. Figure 5a shows latency vs throughput of Pilotfish on this workload with



(a) (b)

Fig. 7: (a) Pilotfish scalability with computationally heavy transactions. `F{X}` means that each transaction computes the X-th Fibonacci number. The horizontal lines show the single-machine throughput of the baseline on the same workloads. (b) Pilotfish scalability with condended transaction. Each transaction increments a counter. `Cont-X` means that for each counter we submit X increment transactions.



(a) 10 transactions/counter (b) 100 transactions/counter (c) 1000 transactions/counter

Fig. 8: Pilotfish latency vs throughput for the contended workloads. Please note the different y axis ranges between the three cases.

1, 2, 4 and 8 ExecutionWorkers, and Figure 5b shows how Pilotfish's maximum throughput scales when varying the number of ExecutionWorkers.

Figure 5b includes as baseline the throughput of the Sui execution engine.[9] Since the Sui transaction manager currently relies on stable storage, whereas Pilotfish is in-memory, this baseline is a lower bound on the expected performance of our system, when using a single ExecutionWorker.

We observe that in all cases, Pilotfish maintains a 20ms latency envelope for this workload. Note that latency exhibits a linear increase as the workload grows for a single ExecutionWorker, primarily because of transaction queuing. More specifically, we see that a single machine does not have enough cores to fully exploit the parallelism of the workload, so some transactions must wait to get scheduled. This effect no longer exists for higher numbers of ExecutionWorkers, showing that more hardware has a beneficial effect on service time.

Pilotfish scales up to around $60k$ tx/s. In contrast, the Sui baseline can only process around $20k$ tx/s as it cannot leverage the additional hardware. Pilotfish thus exhibits a $3\times$ throughput improvement over the baseline.

[9] We obtain the baseline by running Sui's single node benchmark with the `with-tx-manager` option.

Pilotfish's scalability is not perfectly linear in this workload; in particular, it becomes less steep after 2 ExecutionWorkers. This is because the simple transfers workload is computationally light, and so the system is not compute-bound. Thus adding more resources no longer improves performance proportionally. Section 8.3 illustrates the advantages of increasing the number of ExecutionWorkers further when the workload is compute-bound.

8.3 Computationally-Heavy Workload

We study the scenario when the workload remains compute-bound even at higher numbers of ExecutionWorkers. In this workload, transactions are computationally heavy. To achieve this, each transaction merges two coins and then iteratively computes the Xth Fibonacci number, where X is a configurable parameter. We study the behavior of Pilotfish for $X \in \{2500, 5000, 10000\}$. This workload is also perfectly parallel: transactions operate on disjoint sets of coins and thus do not conflict. Figure 6 and Figure 7a show the results: latency vs throughput and throughput scalability of Pilotfish, respectively. Figure 7a includes the behavior of Sui on the same workloads, as a baseline.

As expected the performance of Pilotfish is on par with the Sui baseline for all three computation intensities when running on a single ExecutionWorker. However, when computing resources are the bottleneck, Pilotfish scales linearly as more resources are added to the system. As a result, Pilotfish can process 20k, 10k, and 5k tx/s when setting $X = 2500$, $X = 5000$, and $X = 10000$, respectively, while maintaining the latency at around 50 ms. In contrast, the throughput of the baseline execution engine of Sui remains set to a maximum of 2,5k, 1k, and 500 tx/s (with respectively $X = 2500$, $X = 5000$, and $X = 10000$) as it is unable to take advantage of the additional hardware. As a result, Pilotfish can process about 10x more transactions than the Sui baseline.

8.4 Contended Workload

We study the behavior of Pilotfish when the workload is no longer perfectly parallelizable. To achieve this, we introduce contention by making transactions operate on non-disjoint sets of objects. More concretely, in this workload each transaction increments a counter; for each counter, we generate a configurable number Y of transactions that increment it. Thus, on average, each transaction needs to wait behind $Y/2$ other transactions in its counter's queue, before being able to execute. In our experiments, $Y \in \{10, 100, 1000\}$. The results are shown in Figure 8 and Figure 7b. Pilotfish reaches a throughput of 35k, 30k, and 22k tx/s for $Y = 10$, $Y = 100$, and $Y = 1000$ when operating with 4 ExecutionWorkers. For this workload, for technical reasons,[10] we could not include a Sui baseline.

As expected, we observe that as we increase contention, latency increases due to queueing (up to 500ms for $Y = 1000$) and throughput decreases. Nonetheless, Pilotfish is able to scale to 4 ExecutionWorkers. Similarly to the simple transfer workload (Section 8.2), this workload is not compute-bound, so adding compute beyond 4 ExecutionWorkers no longer improves performance proportionally.

[10] In Sui, each transaction expects object references for all input objects. Each object reference is computed based on the last transaction to modify the object. Therefore, it is difficult to pre-generate more than one valid transaction for the same object, before the experiment starts, because correct object references cannot be predicted.

Table 1: Comparison against existing deterministic approaches

	Distributed	**Crash Tolerance**	**Dynamic RW Set**	**No CC Aborts**
Bohm [27]	✗	✗	✗	✓
PWV [28]	✗	✓	✗	✓
QueCC [49]	✗	✗	✗	✓
SLOG [50]	✓	✓	✗	✗
Q-Store [48]	✓	✓	✗	✓
Calvin [61]	✓	✓	✗	✓
Aria [40]	✓	✓	✓	✗
Lotus [63]	✓	✓	✓	✗
Pilotfish (this work)	✓	✓	✓	✓

9 Related Work

Parallel blockchain executors. The main proposals in this area are those of Solana [52], Aptos [31], and Sui [13]. Solana [52] requires every transaction to fully specify its read and write sets, so it cannot support dynamic accesses in the same way as Pilotfish. Aptos uses the Block-STM [31] system for parallel transaction execution. Block-STM is designed with a focus on single-machine, multi-threaded performance, and it is unclear how or if its design can be extended to the scale-out, distributed deployment that Pilotfish targets. For instance, Block-STM executes transactions speculatively, and retries transactions which fail validation. This approach works well in a shared memory environment, where retries are relatively inexpensive, but it is not clear if it can be applied in a distributed environment, where retries are much more costly due to higher communication latency. Futhermore, BlockSTM focuses on a per-block execution model which requires large blocks to optimize throughput, at the expense of latency. By contrast, Pilotfish uses a streaming execution model that allows for low latency regardless of throughput. Finally, Sui [13] implicitly handles synchronization and scheduling through the tokio runtime [62]: a tokio task is spawned for each Sui transaction; this task waits for the transaction's dependencies to be satisfied (i.e. the required object versions to be available), and then executes the transaction in parallel with other tasks. It is unclear how to directly extend this approach to multiple machines, as required by Pilotfish.

Deterministic databases. Pilotfish is similar to deterministic database systems [59] that employ an order-then-execute approach. Table 1 summarizes the main differences between Pilotfish and existing deterministic approaches. As Table 1 shows, Pilotfish is the first distributed, crash fault tolerant deterministic execution engine that tolerates partially unspecified read/write sets and eliminates concurrency-control-related aborts. The closest works to Pilotfish are Calvin [60], Aria [40] and Lotus [63]. Calvin [61] proposes the use of consensus to address crashes, which in our setting is overkill since the blockchain already provides sufficient determinism to recover without strong coordination. Aria and Lotus differ from Pilotfish by not establishing a total order on transactions before execution, which can lead to some transactions aborting due to conflicts; such transactions have to be retried later, increasing latency.

References

1. Aguilera, M.K., Chen, W., Toueg, S.: Failure detection and consensus in the crash-recovery model. Distributed Comput. **13**(2), 99–125 (2000)
2. Aguilera, M.K., Merchant, A., Shah, M.A., Veitch, A.C., Karamanolis, C.T.: Sinfonia: A new paradigm for building scalable distributed systems. ACM Trans. Comput. Syst. **27**(3), 5:1–5:48 (2009). https://doi.org/10.1145/1629087.1629088, https://doi.org/10.1145/1629087.1629088
3. Al-Bassam, M.: Lazyledger: A distributed data availability ledger with client-side smart contracts. arXiv preprint arXiv:1905.09274 (2019)
4. Al-Bassam, M., Sonnino, A., Bano, S., Hrycyszyn, D., Danezis, G.: Chainspace: A sharded smart contracts platform. arXiv preprint arXiv:1708.03778 (2017)
5. Al-Dhuraibi, Y., Paraiso, F., Djarallah, N., Merle, P.: Elasticity in cloud computing: State of the art and research challenges. IEEE Trans. Serv. Comput. **11**(2), 430–447 (2018)
6. Amiri, M.J., Agrawal, D., Abbadi, A.E.: Sharper: Sharding permissioned blockchains over network clusters. In: SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021. pp. 76–88. ACM (2021), https://doi.org/10.1145/3448016.3452807
7. Aptos: Aptos. https://aptoslabs.com (2024)
8. Aptos Node Requirements. https://aptos.dev/en/network/nodes/validator-node/node-requirements, accessed: 2024-08-02
9. Avarikioti, Z., Desjardins, A., Kokoris-Kogias, L., Wattenhofer, R.: Divide & scale: Formalization and roadmap to robust sharding. In: International Colloquium on Structural Information and Communication Complexity. pp. 199–245. Springer (2023)
10. Automatically manage Amazon ECS capacity with cluster auto scaling. https://docs.aws.amazon.com/AmazonECS/latest/developerguide/cluster-auto-scaling.html, accessed: 2024-08-02
11. Babel, K., Chursin, A., Danezis, G., Kokoris-Kogias, L., Sonnino, A.: Mysticeti: Low-latency dag consensus with fast commit path. arXiv preprint arXiv:2310.14821 (2023)
12. Bagaria, V., Kannan, S., Tse, D., Fanti, G., Viswanath, P.: Prism: Deconstructing the blockchain to approach physical limits. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. pp. 585–602 (2019)
13. Blackshear, S., Chursin, A., Danezis, G., Kichidis, A., Kokoris-Kogias, L., Li, X., Logan, M., Menon, A., Nowacki, T., Sonnino, A., et al.: Sui lutris: A blockchain combining broadcast and consensus. arXiv preprint arXiv:2310.18042 (2023)
14. Bondi, A.B.: Characteristics of scalability and their impact on performance. In: Second International Workshop on Software and Performance, WOSP 2000, Ottawa, Canada, September 17-20, 2000. pp. 195–203. ACM (2000). https://doi.org/10.1145/350391.350432, https://doi.org/10.1145/350391.350432
15. Celestia: The first modular blockchain network. https://celestia.org (2022)
16. Chandra, T.D., Hadzilacos, V., Toueg, S.: The weakest failure detector for solving consensus. J. ACM **43**(4), 685–722 (jul 1996). https://doi.org/10.1145/234533.234549, https://doi.org/10.1145/234533.234549
17. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. J. ACM **43**(2), 225–267 (mar 1996). https://doi.org/10.1145/226643.226647, https://doi.org/10.1145/226643.226647
18. CoinMarketCap. http://www.coinmarketcap.com, accessed: 2024-08-02
19. Cooper, B.F., Ramakrishnan, R., Srivastava, U., Silberstein, A., Bohannon, P., Jacobsen, H., Puz, N., Weaver, D., Yerneni, R.: PNUTS: yahoo!'s hosted data

serving platform. Proc. VLDB Endow. **1**(2), 1277–1288 (2008). https://doi.org/10.14778/1454159.1454167, http://www.vldb.org/pvldb/vol1/1454167.pdf
20. Danezis, G., Kokoris-Kogias, L., Sonnino, A., Spiegelman, A.: Narwhal and tusk: a dag-based mempool and efficient bft consensus. In: Proceedings of the Seventeenth European Conference on Computer Systems. pp. 34–50 (2022)
21. Dang, H., Dinh, T.T.A., Loghin, D., Chang, E., Lin, Q., Ooi, B.C.: Towards scaling blockchain systems via sharding. In: Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019. pp. 123–140. ACM (2019), https://doi.org/10.1145/3299869.3319889
22. Das, S., Agrawal, D., Abbadi, A.E.: ElasTraS: An elastic, scalable, and self-managing transactional database for the cloud. ACM Trans. Database Syst. **38**(1), 5 (2013). https://doi.org/10.1145/2445583.2445588, https://doi.org/10.1145/2445583.2445588
23. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: Amazon's highly available key-value store. In: Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007. pp. 205–220. ACM (2007). https://doi.org/10.1145/1294261.1294281, https://doi.org/10.1145/1294261.1294281
24. Docs: Move VM. https://docs.dfinance.co/move_vm (2023)
25. Dwork, C., Lynch, N., Stockmeyer, L.: Consensus in the presence of partial synchrony. Journal of the ACM (JACM) **35**(2), 288–323 (1988)
26. Ethereum Foundation: Ethereum Virtual Machine (EVM). https://ethereum.org/en/developers/docs/evm/ (2023)
27. Faleiro, J.M., Abadi, D.J.: Rethinking serializable multiversion concurrency control. Proc. VLDB Endow. **8**(11), 1190–1201 (jul 2015). https://doi.org/10.14778/2809974.2809981, https://doi.org/10.14778/2809974.2809981
28. Faleiro, J.M., Abadi, D.J., Hellerstein, J.M.: High performance transactions via early write visibility. Proc. VLDB Endow. **10**(5), 613–624 (jan 2017). https://doi.org/10.14778/3055540.3055553, https://doi.org/10.14778/3055540.3055553
29. Fuel: The World's Fastest Modular Execution Layer. https://www.fuel.network (2024)
30. Gao, Y., Lu, Y., Lu, Z., Tang, Q., Xu, J., Zhang, Z.: Dumbo-ng: Fast asynchronous bft consensus with throughput-oblivious latency. In: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (2022)
31. Gelashvili, R., Spiegelman, A., Xiang, Z., Danezis, G., Li, Z., Malkhi, D., Xia, Y., Zhou, R.: Block-stm: Scaling blockchain execution by turning ordering curse to a performance blessing. In: PPoPP '23. p. 232–244. Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3572848.3577524, https://doi.org/10.1145/3572848.3577524
32. Giridharan, N., Kokoris-Kogias, L., Sonnino, A., Spiegelman, A.: Bullshark: Dag bft protocols made practical. arXiv preprint arXiv:2201.05677 (2022)
33. Hellings, J., Hughes, D.P., Primero, J., Sadoghi, M.: Cerberus: Minimalistic Multi-shard Byzantine-resilient Transaction Processing. J. Syst. Res. **3**(1) (2023), https://doi.org/10.5070/sr33161314
34. Hellings, J., Sadoghi, M.: ByShard: sharding in a Byzantine environment. VLDB J. **32**(6), 1343–1367 (2023), https://doi.org/10.1007/s00778-023-00794-0
35. Herbst, N.R., Kounev, S., Reussner, R.H.: Elasticity in cloud computing: What it is, and what it is not. In: Kephart, J.O., Pu, C., Zhu, X. (eds.) 10th International Conference on Autonomic Computing, ICAC'13, San Jose, CA, USA, June 26-28, 2013. pp. 23–27. USENIX Association (2013)

36. Howard, H., Malkhi, D., Spiegelman, A.: Flexible paxos: Quorum intersection revisited (2016)
37. Kokoris-Kogias, E., Jovanovic, P., Gasser, L., Gailly, N., Syta, E., Ford, B.: Omniledger: A secure, scale-out, decentralized ledger via sharding. In: 2018 IEEE symposium on security and privacy (SP). pp. 583–598. IEEE (2018)
38. Lakshman, A., Malik, P.: Cassandra: structured storage system on a P2P network. In: Tirthapura, S., Alvisi, L. (eds.) Proceedings of the 28th Annual ACM Symposium on Principles of Distributed Computing, PODC 2009, Calgary, Alberta, Canada, August 10-12, 2009. p. 5. ACM (2009). https://doi.org/10.1145/1582716.1582722, https://doi.org/10.1145/1582716.1582722
39. Lamport, L.: Paxos made simple. ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001) pp. 51–58 (2001)
40. Lu, Y., Yu, X., Cao, L., Madden, S.: Aria: A fast and practical deterministic oltp database. Proc. VLDB Endow. **13**(12), 2047–2060 (jul 2020). https://doi.org/10.14778/3407790.3407808, https://doi.org/10.14778/3407790.3407808
41. Mohan, C., Lindsay, B.G., Obermarck, R.: Transaction management in the r* distributed database management system. ACM Trans. Database Syst. **11**(4), 378–396 (1986). https://doi.org/10.1145/7239.7266, https://doi.org/10.1145/7239.7266
42. Mysten Labs: Build without boundaries. https://sui.io (2022)
43. Mysten Labs: Move Concepts. https://docs.sui.io/concepts/sui-move-concepts (2023)
44. Neu, J., Tas, E.N., Tse, D.: Ebb-and-flow protocols: A resolution of the availability-finality dilemma. In: 2021 IEEE Symposium on Security and Privacy (SP). pp. 446–465. IEEE (2021)
45. Neu, J., Tas, E.N., Tse, D.: The availability-accountability dilemma and its resolution via accountability gadgets. In: International Conference on Financial Cryptography and Data Security. pp. 541–559. Springer (2022)
46. Ongaro, D., Ousterhout, J.: The raft consensus algorithm. Lecture Notes CS **190**, 2022 (2015)
47. Protocol, S.: What Is SVM - The Solana Virtual Machine. https://squads.so/blog/solana-svm-sealevel-virtual-machine (2024)
48. Qadah, T., Gupta, S., Sadoghi, M.: Q-store: Distributed, multi-partition transactions via queue-oriented execution and communication. In: Proceedings of the 23rd International Conference on Extending Database Technology, EDBT 2020, Copenhagen, Denmark, March 30 - April 02, 2020. pp. 73–84. OpenProceedings.org (2020), https://doi.org/10.5441/002/edbt.2020.08
49. Qadah, T.M., Sadoghi, M.: QueCC: A queue-oriented, control-free concurrency architecture. In: Proceedings of the 19th International Middleware Conference, Middleware 2018, Rennes, France, December 10-14, 2018. pp. 13–25. ACM (2018), https://doi.org/10.1145/3274808.3274810
50. Ren, K., Li, D., Abadi, D.J.: SLOG: serializable, low-latency, geo-replicated transactions. Proc. VLDB Endow. **12**(11), 1747–1761 (2019). https://doi.org/10.14778/3342263.3342647, http://www.vldb.org/pvldb/vol12/p1747-ren.pdf
51. Serafini, M., Mansour, E., Aboulnaga, A., Salem, K., Rafiq, T., Minhas, U.F.: Accordion: Elastic scalability for database systems supporting distributed transactions. Proc. VLDB Endow. **7**(12), 1035–1046 (2014), http://www.vldb.org/pvldb/vol7/p1035-serafini.pdf
52. Solana Foundation: Sealevel—parallel processing thousands of smart contracts. https://solana.com/news/sealevel---parallel-processing-thousands-of-smart-contracts (2019)
53. Solana Validator Requirements. https://docs.solanalabs.com/operations/requirements, accessed: 2024-08-02

54. Sonnino, A., Bano, S., Al-Bassam, M., Danezis, G.: Replay attacks and defenses against cross-shard consensus in sharded distributed ledgers. In: 2020 IEEE European Symposium on Security and Privacy (EuroS&P). pp. 294–308. IEEE (2020)
55. Spiegelman, A., Aurn, B., Gelashvili, R., Li, Z.: Shoal: Improving dag-bft latency and robustness. arXiv preprint arXiv:2306.03058 (2023)
56. Stefo, C., Xiang, Z., Kokoris-Kogias, L.: Executing and proving over dirty ledgers. In: International Conference on Financial Cryptography and Data Security. pp. 3–20. Springer (2023)
57. Sui: Object Model. `https://docs.sui.io/concepts/object-model`, accessed: 2024-08-02
58. Sui Validator Node Configuration. `https://docs.sui.io/guides/operator/validator-config`, accessed: 2024-08-02
59. Thomson, A., Abadi, D.J.: The case for determinism in database systems. Proc. VLDB Endow. **3**(1–2), 70–80 (sep 2010). `https://doi.org/10.14778/1920841.1920855`, `https://doi.org/10.14778/1920841.1920855`
60. Thomson, A., Diamond, T., Weng, S.C., Ren, K., Shao, P., Abadi, D.J.: Calvin: Fast distributed transactions for partitioned database systems. In: Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data. p. 1–12. SIGMOD '12, Association for Computing Machinery, New York, NY, USA (2012). `https://doi.org/10.1145/2213836.2213838`, `https://doi.org/10.1145/2213836.2213838`
61. Thomson, A., Diamond, T., Weng, S.C., Ren, K., Shao, P., Abadi, D.J.: Fast distributed transactions and strongly consistent replication for oltp database systems. ACM Trans. Database Syst. **39**(2) (may 2014). `https://doi.org/10.1145/2556685`, `https://doi.org/10.1145/2556685`
62. Tokio is an asynchronous runtime for the Rust programming language. `https://tokio.rs/`, accessed: 2024-10-04
63. Zhou, X., Yu, X., Graefe, G., Stonebraker, M.: Lotus: Scalable multi-partition transactions on single-threaded partitioned databases. Proc. VLDB Endow. **15**(11), 2939–2952 (2022), `https://www.vldb.org/pvldb/vol15/p2939-zhou.pdf`

APPENDIX

A Algorithms

This section complements Section 4 by providing detailed algorithms for the core components of Pilotfish.

A.1 Detailed Algorithms

The function HANDLE(*oid*) of Algorithm 1 returns the ExecutionWorker that handles the specified object identifier *oid*. The function INDEX(Tx) in Algorithm 3 returns the index of the transaction Tx in the global committed sequence. The function ID(o) in Algorithm 5 returns the object id *oid* of the object o.

A.2 Running in Constant Memory

The algorithms described above leverage several temporary in-memory structures that need to be safely cleaned up to make the protocol memory-bound. The maps PENDING and MISSING are respectively cleaned up as part of normal protocol operations at Line 34 (empty queues are deleted) and Line 10 of Algorithm 5. All indices $i' < \mathbf{i}$ of the list $\mathbf{B}$ (Algorithm 2) can be cleaned after Line 9

Algorithm 1 Process committed sequence (step ❷ of Figure 2)

```
   // Called by SequencingWorkers upon receiving the committed sequence.
1: procedure ProcessSequencedBatch(BatchId, BatchIdx)
2:     // Ignore batches for other workers.
3:     if Handler(BatchId) ≠ Self then return
4:
5:     // Make one propose message for each ExecutionWorker.
6:     Batch ← Batches[BatchId]
7:     for w ∈ ExecutionWorkers do
8:         T ← [Tx ∈ Batch s.t. ∃oid ∈ Tx s.t. Handler(oid) = w]
9:         ProposeMessage ← (BatchIdx, BatchId, T)
10:        Send(w, ProposeMessage)
```

Algorithm 2 Process ProposeMessage (step ❸ of Figure 2)

```
1: i ← 0                                   ▷ All batch indices below this watermark are received
2: B ← [ ]                                 ▷ Received batch indices

   // Called by ExecutionWorkers upon receiving a ProposeMessage.
3: procedure ProcessPropose(ProposeMessage)
4:     // Ensure we received one message per SequencingWorker
5:     (BatchIdx, BatchId, T) ← ProposeMessage
6:     B[BatchIdx] ← B[BatchIdx] ∪ (BatchId, T)
7:     while len(B[i]) = |SequencingWorkers| do
8:         (_, T) ← B[i]
9:         i ← i + 1
10:
11:        // Add the objects to their pending queues
12:        for Tx ∈ T do
13:            for oid ∈ HandledObjects(Tx) do          ▷ Defined in Algorithm 3
14:                if oid ∈ W(Tx) then
15:                    Pending[oid] ← Pending[oid] ∪ (W, [Tx])
16:                else                                 ▷ oid ∈ R(Tx)
17:                    (op, T′) ← Pending[oid][−1]
18:                    if op = W then Pending[oid] ← Pending[oid] ∪ (R, [Tx])
19:                    else Pending[oid][−1] ← (R, T′ ∪ Tx)
20:
21:            // Try to execute the transaction
22:            TryTriggerExecution(Tx)                  ▷ Defined in Algorithm 3
```

of Algorithm 2 as they are no longer needed. Similarly, any transactions Tx with index Index(Tx) $<$ **j** can be removed from the set **E** (Algorithm 3) after Line 39 of Algorithm 3. Finally, any transaction Tx can be removed from the map **R** (Algorithm 4) after Line 5 of Algorithm 4.

B Security Proofs

We show that Pilotfish satisfies the properties of Section 2.

B.1 Serializability

We show that Pilotfish satisfies the serializability property (Definition 1 of Section 2). Intuitively, this property states that Pilotfish executes transactions in a way that is equivalent to the sequential execution of the transactions as it comes from consensus (Definition 4). The argument leverages the following arguments: (i) Pilotfish builds the pending queues Pending by respecting the transactions dependencies dictated by the consensus protocol (i.e., the sequential schedule), (ii) Pilotfish accesses objects in the same order as the sequential schedule, and (iii) Pilotfish executes transactions in the same order as the sequential schedule.

Definition 4 (Sequential Schedule). *A sequential schedule is a sequence of transactions* $[\mathsf{Tx}_1, \ldots, \mathsf{Tx}_n]$ *where each transaction* Tx_i *is executed after* Tx_{i-1}.

Algorithm 3 Core functions

```
1: j ← 0                                          ▷ All Tx indices below this watermark are executed
2: E ← ∅                                                          ▷ Executed transaction indices

3: function TryTriggerExecution(Tx)
4:     // Check if all dependencies are already executed
5:     if HasDependencies(Tx) then return
6:
7:     // Check if all objects are present
8:     M ← MissingObjects(Tx)
9:     if M ≠ ∅ then
10:        for oid ∈ M do Missing[oid] ← Missing[oid] ∪ Tx
11:        return
12:
13:    // Send object data to a deterministically-selected ExecutionWorker
14:    worker ← Handler(Tx)                          ▷ Worker handling the most objects of Tx
15:    O ← {Objects[oid] s.t. oid ∈ HandledObjects(Tx)}                          ▷ May contain ⊥
16:    ReadyMessage ← (Tx, O)
17:    Send(worker, ReadyMessage)
18:
19:    // Remove read-locks from the pending queues
20:    for oid ∈ R(Tx) do
21:        T′ ← AdvanceLock(Tx, oid)
22:        for Tx′ ∈ T′ do TryTriggerExecution(Tx′)

23: function HasDependencies(Tx)
24:    I ← HandledObjects(Tx)
25:    return ∃oid ∈ I s.t. Tx ∉ Pending[oid][0]

26: function MissingObjects(Tx)
27:    I ← HandledObjects(Tx)
28:    return {oid s.t. oid ∈ I and Objects[oid] =⊥ and j < Index(Tx) − 1}

29: function HandledObjects(Tx)
30:    return {oid s.t. oid ∈ Tx and Handler(oid) = Self}

31: function AdvanceLock(Tx, oid)
32:    // Cleanup the pending queue
33:    (op, T) ← Pending[oid][0]
34:    (op, T′) ← (op, l \ Tx)
35:    Pending[oid][0] ← (op, T′)
36:    return T′

37: function TryAdvanceExecWatermark(Tx)
38:    E ← E ∪ Index(Tx)
39:    while (j + 1) ∈ E do j ← j + 1
```

Definition 5 (Conflicting Transactions). *Two transactions Tx_i and Tx_j conflict on some object oid if both Tx_i and Tx_j reference oid in their read or write set and at least one of Tx_i or Tx_j references oid in its write set.*

Pending queues building. We start by arguing point (i), stating that Algorithm 2 builds the pending queues Pending by respecting the transaction dependencies dictated by the consensus protocol (i.e., the sequential schedule).

Lemma 1 (Sequential Batch Processing). *Pilotfish processes the batch with index Batch_j after processing the batch with index Batch_i if $j > i$.*

Proof. Let's assume by contradiction that Algorithm 2 processes the ProposeMessage referencing transactions of the batch with index Batch_j before processing the ProposeMessage referencing transactions of the batch with index Batch_i while

Algorithm 4 Process ReadyMessage (step ❹ of Figure 2)

```
1: R ← {}                                   ▷ Maps Tx to the object data it references (or ⊥ if unavailable)

   // Called by the ExecutionWorkers upon receiving a ReadyMessage.
2: procedure ProcessReady(ReadyMessage)
3:     (Tx, O) ← ReadyMessage
4:     R[Tx] ← R[Tx] ∪ O
5:     if len(R[Tx]) ≠ len(R(Tx)) + len(W(Tx)) then return
6:
7:     ResultMessage ← (Tx, ∅, ∅)
8:     if !AbortExec(Tx) then
9:         (O, I) ← exec(Tx, ReceivedObj[Tx])          ▷ O to mutate and I to delete
10:        for w ∈ ExecutionWorkers do
11:            O_w ← {o ∈ O s.t. Handler(o) = w}
12:            I_w ← {oid ∈ I s.t. Handler(oid) = w}
13:            ResultMessage ← (Tx, O_w, I_w)
14:    Send(w, ResultMessage)

   // Check whether the execution should proceed.
15: function AbortExec(Tx)
16:     return ∃o ∈ R[Tx] s.t. o =⊥
```

Algorithm 5 Process ResultMessage (step ❺ of Figure 2)

```
   // Called by the ExecutionWorkers upon receiving a ResultMessage.
1: procedure ProcessResult(ResultMessage)
2:     (Tx, O, I) ← ResultMessage
3:     TryAdvanceExecWatermark(Tx)                  ▷ Defined in Algorithm 3
4:     UpdateStores(Tx, O, I)
5:
6:     // Try execute transactions with missing objects
7:     for o ∈ O do
8:         oid ← Id(o)
9:         for Tx ← Missing[oid] do TryTriggerExecution(Tx)
10:        Delete Missing[oid]                      ▷ Prevent duplicate execution
11:
12:    // Try executing the next transaction in the queues
13:    for oid ∈ Tx do
14:        T' ← AdvanceLock(Tx, oid)
15:        for Tx' ∈ T' do TryTriggerExecution(Tx')

16: function UpdateStores(Tx, O, I)
17:     for o ∈ O do Objects[Id(o)] ← o
18:     for oid ∈ I do Delete Objects[oid]
```

$j > i$. This means that Algorithm 2 processes Batch_j at Line 12 before processing Batch_i at Line 12. However, the check of Algorithm 2 at Line 7 ensures that Batch_j can only be processed after all the batches with indices $k \in [0, \ldots, j[$. Since $j > i$, it follows that $i \in [0, \ldots, j[$, and thus Batch_j can only be processed after Batch_i. Hence a contradiction.

Lemma 2 (Transactions Order in Queues). *Let's assume two transactions $\mathsf{Tx}_j, \mathsf{Tx}_i$ such that $j > i$ conflict on the same oid; Tx_j is placed in the queue* Pending[*oid*] *after Tx_i.*

Proof. We first observe that if two transactions Tx_j and Tx_i are conflicting on object *oid* then they are placed in the same queue Pending[*oid*]. Indeed, both Tx_i and Tx_j are embedded in a ProposeMessage by Algorithm 1. They are then placed in the queue Pending[*oid*] by Algorithm 2 at Line 15 (if they reference *oid* in their write set) or Line 18 (if they reference *oid* in their read set).

We are thus left to prove that Tx_j is placed in PENDING[*oid*] after Tx_i. Since $j > i$ we distinguish two cases: (i) both Tx_j and Tx_i are part of the same batch with index BatchIdx and (ii) Tx_j and Tx_i are part of different batches with indices Batch_j and Batch_i respectively. In the first case (i), Tx_j and Tx_i are referenced in the same ProposeMessage by Algorithm 1 at Line 8 and Line 9 but respecting the order $j > i$. As a result, Tx_j is processed after Tx_i by the loop Line 12, and placed in the queue PENDING[*oid*] (at Line 15 or Line 18) after Tx_i. In the second case (ii), Tx_j and Tx_i are referenced in different ProposeMessage by Algorithm 1 at Line 9 but Lemma 1 ensures that the ProposeMessage referencing transactions of Batch_j is processed after the ProposeMessage referencing transactions of Batch_i. As a result, Tx_j is placed in the queue PENDING[*oid*] after Tx_i (at Line 15 or Line 18).

Sequential objects access. We now argue point (ii), namely that Algorithm 3 accesses objects in the same order as the sequential schedule.

Lemma 3 (Unlock after Access). *If a transaction T is placed in a queue* PENDING[*oid*]*, it can only be removed from that queue after accessing* OBJECTS[*oid*]*.*

Proof. We argue this lemma by construction of the algorithms of Pilotfish. Transaction T accesses OBJECTS[*oid*] only at Line 15 (Algorithm 3) and can only be removed from PENDING[*oid*] following a call to ADVANCELOCK(T, *oid*). This call can occur only in two places. It can first occur (i) at Line 21 of Algorithm 3 which happens after the access to OBJECTS[*oid*] (Line 15 of the same algorithm). It can then occur (ii) at Line 14 of Algorithm 5 which can only be triggered upon receiving a ResultMessage referencing T, which in turn can only be created after creating a ReadyMessage embedding T. However, creating the latter message only occurs at Line 17 of Algorithm 3, thus after accessing OBJECTS[*oid*] (Line 15 of that same algorithm).

Lemma 4 (Sequential Object Access). *If a transaction Tx_j is placed in a queue* PENDING[*oid*] *after a transaction Tx_i, then Tx_j accesses oid after Tx_i.*

Proof. Let's assume that Tx_j and Tx_i are respectively placed at positions j' and i' of the queue PENDING[*oid*], with $j' > i'$. Let's assume by contradiction that Tx_j accesses *oid* before Tx_i. Access to *oid* is only performed by Algorithm 3 at Line 15 after successfully passing the 'dependencies' check at Line 5. Lemma 3 thus ensures that Tx_i is still in PENDING[*oid*] when the call to HASHDEPENDECNIES(Tx_j) at Line 5 returns **False**. This is however a direct contradiction of the check at Line 25 which ensures that HASDEPENDECNIES(Tx_j) returns **False** only if Tx_j is in the PENDING[*oid*] at position $j' = 0$. However, since Tx_i is still in PENDING[*oid*], it follows that $0 \leq i' < j'$, thus a contradiction.

Sequential transaction execution. We finally argue point (iii), namely that Algorithm 4 executes transactions in the same order as the sequential schedule.

Lemma 5 (Execution after Object Access). *If a transaction T references oid in its read or write set, it can only be executed after accessing* OBJECTS[*oid*]*.*

Proof. We argue this lemma by construction of Algorithm 4. Transactions are executed only at Line 9 of Algorithm 4 and this algorithm is only triggered upon receiving a ReadyMessage. However, creating the latter message only occurs at Line 17 of Algorithm 3, thus after accessing OBJECTS[*oid*] (Line 15 of that same algorithm).

Theorem 1 (Serializability). *If a correct validator executes the sequence of transactions* $[\mathsf{Tx}_1, \ldots, \mathsf{Tx}_n]$*, it holds the same object state* S *as if the transactions were executed sequentially.*

Proof. Consider some execution E and let $G = (V, E)$ be E's conflict graph. Each transaction is a vertex in V, and there is a directed edge $\mathsf{Tx}_i \rightarrow \mathsf{Tx}_j$ if (1) Tx_i and Tx_j have a conflict on some object *oid* and (2) Tx_j accesses *oid* after Tx_i accesses *oid*. It is sufficient to show that there are no schedules where Tx_j is executed before Tx_i to prove serializability. Let's assume by contradiction that there is a schedule where Tx_j is executed before Tx_i, where $j > i$. Since Tx_j and Tx_i conflict on object *oid*, Lemma 2 ensures that Tx_j is placed in the queue PENDING[*oid*] after Tx_i. Lemma 4 then guarantees that Tx_j's access to *oid* occurs after Tx_i's access on *oid*. However Lemma 5 ensures that Tx_j's execution can only happen after accessing *oid*. It is then impossible to execute Tx_j before Tx_i, hence a contradiction. Since *oid* was chosen arbitrarily, the same reasoning applies to all objects on which Tx_i and Tx_j conflict.

B.2 Determinism

We show that Pilotfish satisfies the determinism property (Definition 2 of Section 2). Intuitively, this property ensures that all correct validators have the same object state after executing the same sequence of transactions. The proof follows from the following arguments: (i) all correct Pilotfish validators build the same dependency graph given the same input sequence of transaction, (ii) individual transaction execution is a deterministic process (Assumption 1), (iii) transactions explicitly reference their entire read and write set (Assumption 2), and (iv) all validators executing the same transactions obtain the same state.

Assumption 1 (Deterministic Individual Execution) *Given an input transaction* Tx *and objects* O*, all calls to* $\text{exec}(\mathsf{Tx}, O)$ *(Line 9 of Algorithm 4) return the same output.*

Assumption 2 (Explicit Read and Write Set) *Each transaction* Tx *explicitly references all the objects of its read and write set. That is, the complete read and write set of* Tx *can be determined by locally inspecting* Tx *without the need for external context.*

Assumption 1 is fulfilled by most blockchain execution environments such as the EVM [26], the SVM [47], and both the MoveVM [24] (used by the Aptos blockchain [7]) and the Sui MoveVM [43] (used by the Sui blockchain [13]). All execution engines except the Sui MoveVM also fulfill Assumption 2 (Section 6 and Appendix E remove this assumption to make Pilotfish compatible with Sui).

We rely on the following lemmas to prove determinism in Theorem 2.

Lemma 6. *Given the same sequence of transactions* $[\mathsf{Tx}_1, \ldots, \mathsf{Tx}_n]$*, all correct validators build the same execution schedule (that is, they build same queues* PENDING*).*

Proof. We argue this property by construction of Algorithm 1 and Algorithm 2. Since all validators receive the same input sequence $[\mathsf{Tx}_1, \ldots, \mathsf{Tx}_n]$ and Algorithm 1 respects the order of transactions (Line 8), all correct validators create the same sequence of ProposeMessage. Lemma 1 then ensures that Algorithm 2 processes each ProposeMessage respecting the transaction order. Finally, Lemma 2 ensures that all correct validators place the transactions in the same order in the queues. Since this process is deterministic, all correct validators build the same queues PENDING.

Lemma 7. *No two correct validators creating the same ResultMessage (Line 13 of Algorithm 4) obtain a different object state* OBJECTS*.*

Proof. We argue this property by construction of Algorithm 5 and by assuming that the communication channel between all ExecutionWorkers of each validator preserves the order of messages.[11] Once a validator creates a ResultMessage (Line 13 of Algorithm 4), it is processed by Algorithm 5. This algorithm first calls TRYADVANCEEXECWATERMARK$(\cdot)$ (Line 3) which does not alter the object state OBJECTS nor the data carried by the ResultMessage, and then deterministically updates the object state OBJECTS (Line 4) based exclusively on the content of the ResultMessage. As a result, all correct validators obtain the same object state

Theorem 2 (Pilotfish Determinism). *No two correct validators that executed the same sequence of transactions* $[\mathsf{Tx}_1, \ldots, \mathsf{Tx}_n]$ *have different stores* OBJECTS*.*

Proof. We argue this property by induction. Assuming the sequence of conflicting transactions $[\mathsf{Tx}_1, \ldots, \mathsf{Tx}_{n-1}]$ for which this property holds, we consider transaction Tx_n. Lemma 6 ensures that all correct validators build the same execution schedule and thus all correct validators execute conflicting transactions in the same order. After scheduling (Algorithm 2), all correct validators create a ReadyMessage referencing Tx_n and the set of objects O (Line 17 of Algorithm 3). Since all validators have the same conflict schedule and the application of the inductive argument ensures that all settled dependencies of Tx_n led to the same state OBJECTS across validators, all correct validators load the same set of objects O and thus create the same ReadyMessage. As a result, all correct validators run Algorithm 4 with the same input and thus execute the same sequence of transactions. By construction of Algorithm 4 and Assumption 2, they all call calls to $exec(\mathsf{Tx}, O)$ (Line 9 of Algorithm 4) with the same inputs Tx and O. Given that Assumption 1 ensures that all calls to $exec(\mathsf{Tx}, O)$ are deterministic, all correct validators thus create the same ResultMessage (Line 13). Finally, Lemma 7 ensures that all validators creating the same ResultMessage obtain the

[11] Our implementation (Section 7) satisfies this assumption by implementing all communication through TCP.

same object state OBJECTS. The inductive base is argued by construction: all correct validators start with the same object state OBJECTS, and thus create the same ReadyMessage (Line 17 of Algorithm 3) and ResultMessage (Line 13 of Algorithm 4) upon executing the first transaction Tx_1, which leads to the same state update across correct validators.

B.3 Liveness

We show that Pilotfish satisfies the liveness property (Definition 3 of Section 2). Intuitively, this property guarantees that valid transactions (Definition 6) are eventually executed. The proof argues that (i) all transactions are eventually processed (Definition 7), and (ii) among those transactions, valid ones are not aborted.

Definition 6 (Valid Transaction). *A transaction T with index $idx = \text{INDEX}(T)$ is valid if all objects referenced by its read and write set are created by a transaction T' with index $idx' < idx$.*

Definition 7 (Processed Transaction). *A transaction T is said* processed *when it is either executed or aborted and the object state* OBJECTS *is updated accordingly.*

Eventual transaction processing. We start by arguing point (i), that is all transactions are eventually processed. This argument relies on several preliminary lemmas leading Lemma 12.

Lemma 8. *The Pilotfish scheduling process is deadlock-free (no circular dependencies).*

Proof. Consider some execution E and let $G = (V, E)$ be E's conflict graph. Each transaction is a vertex in V, and there is a directed edge $\mathsf{Tx}_i \to \mathsf{Tx}_j$ if (1) Tx_i and Tx_j have a conflict on some object *oid* and (2) Tx_j accesses *oid* after Tx_i accesses *oid*. It is sufficient to show that G contains no cycles to prove liveness. That is, it is sufficient to show that G contains no edges $\mathsf{Tx}_j \to \mathsf{Tx}_i$, where $j > i$. Let's assume by contradiction that G has an edge $\mathsf{Tx}_j \to \mathsf{Tx}_i$, where $j > i$. Then, by rule (1) of the construction of G, Tx_j and Tx_i must conflict on some object *oid*. Lemma 2 ensures that Tx_j is placed in the queue in PENDING[*oid*] after Tx_i . Lemma 4 then guarantees that Tx_j's access to *oid* occurs after Ti's access on *oid*. It is then impossible for G to contain an edge $\mathsf{Tx}_j \to \mathsf{Tx}_i$ as this violates rule (2) of the construction of G, hence a contradiction. Since *oid* was chosen arbitrarily, the same reasoning applies to all objects on which Tx_i and Tx_j conflict.

Lemma 9. *If a transaction T is processed (Definition 7), it is eventually removed from all queues* PENDING[*oid*] *where oid is referenced by the read or write set of T.*

Proof. We argue this lemma by construction of Algorithm 3 and Algorithm 5. Transaction T is removed from all queues PENDING[*oid*] upon calling ADVANCELOCK(T, *oid*).

This call can occur in two places. (i) The first call occurs at Line 21 of Algorithm 3 to effectively release read locks. This call occurs right after creating a ReadyMessage referencing T (Line 17), a necessary step to trigger Algorithm 4 and thus process T. (ii) The second call occurs at Line 14 of Algorithm 5 to effectively release write locks. This call occurs right after updating Objects[*oid*] and thus terminating the processing of T.

Lemma 10. *If the sequence of transactions* $[\mathsf{Tx}_1, \ldots, \mathsf{Tx}_n]$ *is processed (Definition 7), the watermark* $\mathbf{j}$ *(Line 1 of Algorithm 3) is advanced to* n*.*

Proof. The processing of T involves updating the object state Objects (Line 4 of Algorithm 5). However, the watermark $\mathbf{j}$ is only updated upon calling

$$\textsc{TryAdvanceExecWatermark}(\mathsf{Tx}_i)$$

at Line 3 of this same algorithm. Thus, by construction, the buffer $\mathbf{E}$ contains every processed transaction Tx_i (Line 38 of Algorithm 3), and once the sequence $[\mathsf{Tx}_1, \ldots, \mathsf{Tx}_n]$ is processed, the watermark $\mathbf{j}$ is advanced to

$$\mathbf{j} = \max\{\textsc{Index}(\mathsf{Tx}_1), \ldots, \textsc{Index}(\mathsf{Tx}_n)\} = n$$

(Line 39 of Algorithm 3).

Lemma 11. *A transaction* T *is eventually processed (Definition 7) if it has neither missing dependencies nor missing objects that could be created by earlier transactions. That is,* T *is eventually processed if Algorithm 3 creates a* ReadyMessage *referencing* T*.*

Proof. We argue this lemma by construction of Algorithm 4 and Algorithm 5. Algorithm 4 receives a ReadyMessage from all ExecutionWorkers and the check Line 5 of Algorithm 4 passes. Transaction T is then either executed (if the check Line 16 passes) or aborted (if the check Line 16 fails), and Algorithm 4 creates a ResultMessage referencing T (Line 13). Algorithm 5 then receives this ResultMessage and accordingly updates its object Objects (after an infallible call to $\textsc{TryAdvanceExecWatermark}(T)$ Line 39).

Lemma 12 (Eventual Transaction Processing). *All correct validators receiving the sequence of transactions* $[\mathsf{Tx}_1, \ldots, \mathsf{Tx}_n]$ *eventually process (Definition 7) all transactions* $\mathsf{Tx}_1, \ldots, \mathsf{Tx}_n$*.*

Proof. Lemma 8 ensures that the transaction scheduling process is deadlock-free (no circular dependencies) and Pilotfish thus triggers their execution (Line 3 of Algorithm 3). We are then left to prove that these scheduled transactions are processed (Definition 7). Since Theorem 1 ensures the Pilotfish schedule is equivalent to a sequential schedule, we prove liveness of the sequential schedule. We argue the lemma's statement by induction. Assuming the sequence of transactions $[\mathsf{Tx}_1, \ldots, \mathsf{Tx}_{n-1}]$ for which this statement holds, we consider transaction Tx_n. Assuming Tx_{n-1} is processed and a sequential schedule, all transactions Tx_i with $i < n-1$ are also processed. Lemma 9 thus ensures these transactions are removed from all queues Pending[$\cdot$]. As a result, when triggering the execution

of Tx_n (Line 3 of Algorithm 3), the check HasDependencies(Tx_n) (Line 5 of Algorithm 3) returns **False** (since $\forall oid \in \mathcal{R}(\mathsf{Tx}_n) \cup \mathcal{W}(\mathsf{Tx}_n)$: Pending[oid][0] = Tx_n). Furthermore, since all transactions Tx_i with $i \leq n-1$ are already processed, Lemma 10 ensures that the watermark $j = n-1$ (Line 1 of Algorithm 3) and thus MissingObjects(Tx_n) returns $\emptyset$. Finally, Algorithm 3 creates a ReadyMessage referring Tx_n and thus Lemma 11 ensures Tx_n is eventually processed. We argue the inductive base by observing that the first transaction Tx_1 has no dependency (by definition); thus both checks HasDependencies(Tx_1) and MissingObjects(Tx_1) pass (respectively at Line 5 and Line 8 of Algorithm 3); and Lemma 11 then ensures Tx_1 is processed.

Valid transaction execution. We now argue point (ii), that valid transactions are not aborted. This argument relies on several preliminary lemmas leading Lemma 15.

Lemma 13. *If a transaction T references an object oid in its write set, it is only removed from the queue* Pending[oid] *after it is processed (Definition 7).*

Proof. We argue this lemma by construction: Transaction T is removed from Pending[oid] only upon a call to AdvanceLock(T, oid). However, since oid is referenced by the write set of T (rather than its read set), this function is called over oid only at Line 14 of Algorithm 5. This call is thus after Algorithm 5 updates Objects[oid] (at Line 4) and thus after the transaction is processed.

Lemma 14. *If a transaction T is valid, the call to* Objects[oid] *(Line 15 of Line 39) never returns $\perp$, for any oid referenced by the read or write set of T.*

Proof. Let's assume by contradiction that there exists a oid referenced by the read or write set of a valid transaction T where the call to Objects[oid] (Line 15 of Line 39) returns $\perp$. Since T is valid, it means that the object oid is created by a conflicting transaction T' with index $idx' <$ Index(T) that has not yet been processed (Definition 7). In which case, Lemma 4 states that both T and T' are placed in the same queue Pending[oid] and Lemma 13 states that T' is still present in the queue Pending[oid]. This is however a direct contradiction of check HasDependencies(T) ensuring that T does not access Objects[oid] until it is at the head of the queue Pending[oid] (Line 15 of Algorithm 3).

Lemma 15. *A valid transaction T it is never aborted; that is the call* AbortExec(T) *(Line 8 of Algorithm 4) returns* ***False****.*

Proof. Let's assume by contradiction that AbortExec(T) returns **True** while T is valid. This means that the check at Line 16 of Algorithm 4 found at least one missing object ($\perp$) referenced in the read or write set of T, and thus that Algorithm 4 received at least one ReadyMessage message referring $\perp$ instead of an object data. However, Lemma 14 ensures that the call to Objects[oid] (Line 15 of Algorithm 3) never returns $\perp$ if oid is referenced by the read or write set of a valid transaction, hence a contradiction.

Liveness proof. We finally combine Lemma 12 and Lemma 15 to prove liveness.

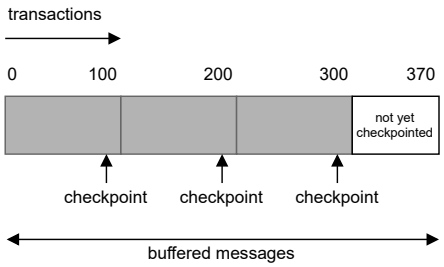


Fig. 9: Example of a snapshot of the local state at an ExecutionWorker. It checkpoints its OBJECTS store after every 100 transactions and keeps a buffer of outgoing ReadyMessage.

Theorem 3 (Liveness). *All correct validators receiving the sequence of transactions* $[\mathsf{Tx}_1, \ldots, \mathsf{Tx}_n]$ *eventually execute all the valid transactions of the sequence.*

Proof. Lemma 12 ensures that all correct validators eventually process (Definition 7) all transactions $\mathsf{Tx}_1, \ldots, \mathsf{Tx}_n$. Lemma 15 ensures that valid transactions are never aborted and thus executed.

C Crash Fault Tolerance

C.1 Internal Replication

For the replication protocol, ExecutionWorkers maintains the following network connections: (i) a constant set of ***peers***, containing the identifier for every worker in its cluster. Workers in each cluster have the same **peers** set; (ii) a dynamic set of **read-to**, containing the additional identifiers with whom the worker is temporarily serving reads to; and (iii) a dynamic set of **read-from**, containing the additional identifiers with whom the worker is receiving reads from. The protocol maintains that **read-from** and **read-to** relations are symmetric — Worker a is in worker b's **read-from** set if and only if worker b is in a's **read-to** set. Finally, we assume the use of an eventually strong failure detector [16].

C.2 Normal Operation

Dealing with finite memory. The number of checkpoints and buffered messages held by an ExecutionWorker cannot grow indefinitely. Hence, we introduce a garbage collection mechanism that deprecates old checkpoints. When an ExecutionWorker completes a checkpoint, it broadcasts a message

$$\mathsf{CheckpointedMessage} \leftarrow (\mathit{shard}, \mathsf{TxIdx})$$

to every other ExecutionWorker in *every* shard, indicating that an ExecutionWorker of shard *shard* successfully persisted a checkpoint immediately after executing TxIdx.

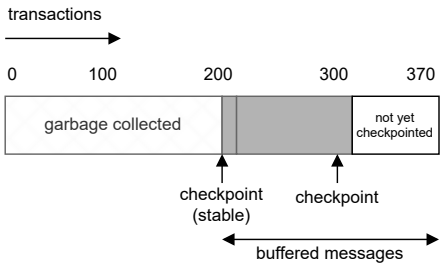


Fig. 10: Example of a snapshot of the local state at an ExecutionWorker. It has received a quorum of CheckpointedMessage messages from each shard for the transaction at checkpoint 1. Hence, checkpoint 1 is stable, and the worker safely deletes checkpoints and buffers before it.

An ExecutionWorker deems a checkpoint at TxIdx as *stable* after receiving a quorum of $f_e + 1$ CheckpointedMessage[12] from each shard. When a worker learns that a checkpoint is stable, it deletes all checkpoints and buffered messages prior to that checkpoint. This illustrated in Figure 10.

Bounding strategy. To avoid exhausting resources, each ExecutionWorker also holds a bounded number c of checkpoints at any time. This number dictates how far ExecutionWorkers are allowed to diverge in terms of their rate of execution; the fastest cluster can be ahead of the slowest cluster in its quorum by up to $c - 1$ checkpoints.[13]

A worker pauses processing when creating a new checkpoint, which will exceed c. This may be a symptom of failures in the system, e.g., many slow or failed workers or network issues. Hence, pausing provides backpressure to fast replicas in order for stragglers to catch up. Figure 11 illustrates this mechanism in a system with three clusters and $c = 2$. Each worker of each cluster holds a stable checkpoint at boundary 1. Clusters 1 and 2 are slow and have yet to reach checkpoint boundary 2. Cluster 3 is fast and hence may execute beyond checkpoint boundary 2, while maintaining a second (non-stable) checkpoint at boundary 2. However, because workers are limited to storing two checkpoints, workers in cluster 3 are blocked from executing past boundary 3 before (i) their checkpoint at boundary 2 is established as stable and (ii) their checkpoint 1 is garbage-collected.

By default, we set $c = 2$ as a good trade-off between performance and storage/memory costs. With a limit of two checkpoints, a fast cluster can execute past a second checkpoint without waiting for a quorum. As such, different clusters are allowed to progress at different speeds without blocking, as long as they stay within one checkpoint. The system then progresses, most of the time, at the speed of the fastest cluster.

[12] Quorum sizes can be varied to optimize between normal case disk-usage and recoverability during failure, similar to Flexible Paxos [36].

[13] Note that ExecutionWorkers within a cluster are always tightly coupled due to the quorum definition, and can never be apart by 1 or more checkpoints

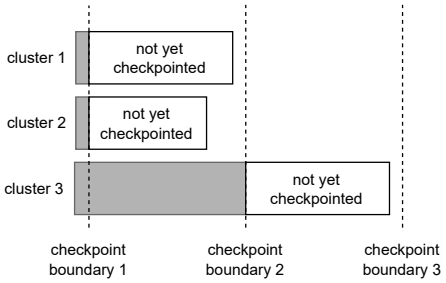


Fig. 11: Consider a system with three clusters, and with each ExecutionWorker allowed to hold up to $c = 2$ checkpoints. This snapshot of the clusters' progress shows that ExecutionWorkers in each cluster have a stable checkpoint at boundary 1. Cluster 3 is fast and hence may execute beyond checkpoint boundary 2 while maintaining a second (non-stable) checkpoint at boundary 2. However, it is not permitted to create a checkpoint at boundary 3 or execute past it before it learns that boundary 2 is stable.

C.3 Failure Recovery

Recovery through reconfiguration. When an ExecutionWorker fails, other workers in its cluster or **read-to** set may not be able to execute transactions, as they may no longer be served the necessary reads. To restore transaction processing, these workers trigger the *reconfiguration* illustrated by Figure 12. In order to detect these crashes, we assume the existence of a failure detector [16] with strong completeness. Ideally, we would use an eventually perfect failure detector, but an eventually strong one suffices for liveness (but might cause worse load-balancing on some ExecutionWorkers).

The crux of recovery is as follows: When a worker detects a failure, it tries to find another worker to get the reads previously managed by the failed worker. In the normal case, this takes two round trips: one trip to find an appropriate **read-from** member and another to establish the relationship with that new member. Meanwhile, all other clusters except the one with the failed worker operate as normal. Hence, there is no loss of throughput when failures are within the tolerated threshold. We defer the details of the recovery algorithm to Appendix D.

Recovery through checkpoints synchronization. The recovery through reconfiguration may fail when the recovering worker finds itself slow after the first round trip. It then needs to perform a *checkpoint synchronization* procedure before retrying recovery. This synchronization is necessary as there may no longer be clusters with sufficiently old *buffers* for the recovering worker to continue execution through the normal recovery procedure.

The gist of the synchronization procedure is as follows:. (i) The worker instructs every peer to perform *synchronization*; (ii) any node in the slow worker's **read-to** set is instructed to itself perform recovery; (iii) the worker then downloads the latest checkpoints from another replica and waits for every peer to sync to the same state; and (iv) the worker finds replacements for any missing members in its cluster by running the *reconfiguration* again. We defer the details of the synchronization algorithm to Appendix D.

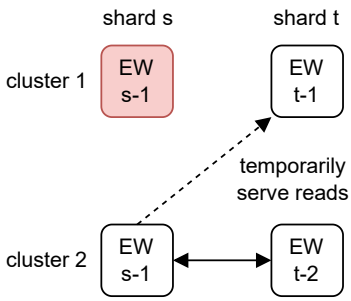


Fig. 12: Failure recovery. Suppose $EW_{s,1}$ crashes. As a result, $EW_{t,1}$ cannot receive reads of shard s from $EW_{s,1}$. After $EW_{t,1}$ performs recovery, it establishes a new read relationship with another replica of shard s, in this case $EW_{s,2}$, as illustrated by the dashed arrow. $EW_{t,1}$ is now in $EW_{s,2}$'s **read-to** set, and correspondingly, $EW_{s,2}$ is in $EW_{t,1}$'s **read-from** set. Otherwise, cluster 2 operates as usual, as represented by the solid arrow.

The synchronization protocol's pivotal aspect lies in its ability to activate synchronization and recovery across all clusters that transitively rely on a slow worker for reads. This occurs because the clusters, which were dependent on the slow worker for their reads, may exhibit lag as well. If the slow worker were to unilaterally fast-forward its state, these clusters could potentially lose liveness.

Disaster recovery. In case of a disaster that affects all ExecutionWorkers of a cluster and the threat model of Section 2 doesn't hold, the cluster can be recovered by booting a new cluster with the same **peers** set. The new cluster will then be able to recover the state of the cluster from the other validators of the blockchain. This is possible because the system state is replicated across multiple validators, of which at least half are honest. The new cluster can then download the latest stable checkpoint from the other validators and use it to perform a recovery through checkpoints. This recovery is slow as it requires WAN communications, but it is only used in extreme circumstances, and its existence allows Pilotfish to be reasonably configured with low replication factors (e.g., $f_e = 1$).

D Detailed Recovery Protocol

This section completes Appendix C by providing the algorithms allowing ExecutionWorkers to recover from crash-faults and proving the security of Pilotfish in this setting.

D.1 Recovery Algorithms

Recovery Protocol. Suppose ExecutionWorker x crashes. Any non-faulty worker e detecting this failure deletes x from its **read-from** and **read-to** sets. If x is e's peer, or a member of its **read-from** set, e may no longer be served reads from x's shard. In this case, e calls RECOVER(x), listed in Algorithm 7.

In the algorithm, first the execution worker, denoted as e, initiates the process by establishing a view on the current execution status of workers in shard x.

Algorithm 6 Process CheckpointedMessage

```
1: C ← {}                                   ▷ Maps TxIdx to checkpoint blob
2: S ← {}                                   ▷ Maps TxIdx to set of sent ReadyMessage
3: R ← 0                                    ▷ Counts the received CheckpointedMessage

4: procedure ProcessCheckpointed(CheckpointedMessage)
5:     (shard, TxIdx) ← CheckpointedMessage
6:     R[(shard, TxIdx)] ← R[(shard, TxIdx)] + 1
7:     if R[(shard, TxIdx)] ≥ f_e + 1 then
8:         for i ← 0; i < TxIdx; i ← i + 1 do
9:             Delete C[i]
10:            Delete S[i]
11:            Delete R[(∗, TxIdx)]
```

This is achieved through a call to $(w, \mathsf{TxIdx}^*) \leftarrow \textsc{GetStatus}(x.\text{shard})$, where w represents the most up-to-date worker in a quorum of workers within x.shard, having executed up to at least TxIdx^*.

Subsequently, based on the obtained result, the execution worker e takes one of two distinct actions. If e's current execution state is *after* TxIdx^*, it initiates the $\textsc{NewReader}(w, \mathsf{TxIdx}^*)$ operation, requesting w to serve its reads that were previously handled by x. Otherwise, if e's current state is *before* TxIdx^*, it engages in the synchronization procedure (Algorithm 9) before attempting recovery. This synchronization step becomes crucial, as there might no longer be clusters with sufficiently old buffers for the recovering worker e to proceed through the standard recovery process. In such instances, e employs the synchronization procedure to load the checkpointed state of another cluster, ensuring a seamless recovery process in the distributed system.

Synchronization Protocol. Synchronize (Algorithm 9) is called by source e and brings e and its peers up to date through loading the checkpointed state of another set of workers. This process on a high level works as follows.

Initially, e communicates with its **read-from** and **read-to** nodes, issuing NotifySync messages to prompt the removal of e from their respective communication sets. Additionally, e notifies its **peers** to cease normal operations and engage in synchronization through the Synchronize procedure.

Afterwards, e clears its own **read-from** and **read-to** sets and requests the current status of the worker w in its shard by attempting to download w's checkpoint. A synchronization message is sent to w, which responds based on whether the checkpoint at TxIdx^* has been deleted or not. If deleted, e retries the synchronization protocol; otherwise, w sends its state snapshot, and e loads it using the $\textsc{LoadCheckpoint}(w)$ procedure.

Finally, e brodcasts a synchronization completion messages to all peers and awaits their responses. If an incoming SyncComplete has a TxIdx greater than TxIdx^*, then e retries the synchronization protocol to ensure uniformity across the entire cluster. The procedure concludes when e receives SyncComplete containing TxIdx from all non-suspected peers, allowing it to initiate recovery for any remaining suspected peers.

D.2 Proofs Modifications

We show that Pilotfish satisfies the security properties defined in Section 4 despite the crash-failure of f_e out of $2f_e + 1$ ExecutionWorkers in all shards.

Algorithm 7 Recovery procedure

```
    // Global states
 1: read-to, read-from, peers
 2: suspected                          ▷ set of suspected workers, updated by failure detector
 3: curr-txidx                         ▷ highest txn that is locally executed and persisted
 4: my-shard                           ▷ identifier for the local shard

 5: procedure RECOVER(x)               ▷ x ∈ peers ∪ read-from
 6:     s ← x.shard
 7:     (w, TxIdx*) ← GETSTATUS(s)
 8:     if TxIdx* ≤ curr-txidx then
 9:         success ← NEWREADER(w, TxIdx*)
10:         if success then
11:             read-from ← read-from ∪ {w}
12:             return True           ▷ recovery is successful
13:         else
14:             return False          ▷ recovery failed, caller should retry
15:     else
16:         for p ∈ peers do
17:             SEND(p, NotifySync)
18:         SYNCHRONIZE()
19:         return true               ▷ May run recovery for each crashed peer

20: procedure GETSTATUS(s)             ▷ s is a shard identifier
21:     for w ∈ shard s do
22:         SEND(w, Recover)
23:     r ← receive RecoverOk
24:     replies ← {r}
25:     r_h ← r                       ▷ reply with highest txid
26:     while |replies| < f + 1 do
27:         r ← receive RecoverOk
28:         r_h ← r if r.txid > r_h.txid else r_h
29:         replies ← replies ∪ {r}
30:     return (r_h.src, r_h.txid)

31: procedure NEWREADER(w, TxIdx*)
32:     NewReader ← (TxIdx*)
33:     SEND(w, NewReader)
34:     reply ← receive reply from w
35:     if reply = NewReaderOk then
36:         return true               ▷ reconfiguration success
37:     else
38:         return false
```

Assumptions. The security of the recovery protocol relies on the following assumptions.

Assumption 3 (Correct Majority) *At least $f_e + 1$ out of $2f_e + 1$ ExecutionWorkers of every shard are correct at all times.*

Assumption 4 (Eventual Synchrony) *The network between ExecutionWorkers is eventually synchronous [25].*

It has been shown that in eventual synchrony, crash failures can eventually be perfectly detected [17]. Thus, Assumption 4 implies the following:

Assumption 5 (Eventually Strong Failure Detector) *There exists an eventually perfect failure detector $\Diamond\mathcal{S}$ with the following properties: (i)* Strong completeness: *Every faulty process is eventually permanently suspected by every nonfaulty process. (ii)* Eventual strong accuracy: *Eventually, correct processes are not suspected by any correct process.*

Algorithm 8 Recovery procedure message handlers

```
   // Global states
1: read-to, read-from, peers
2: stable-txid                                  ▷ txid of locally-stored stable checkpoint
3: buffer                                       ▷ local store of sent ReadyMessage
4: checkpoints                                  ▷ snapshots of local state

5: procedure ProcessRecover(Recover)
6:     RecoverOk ← (stable-txid)
7:     Send(Recover.src, RecoverOk)

8: procedure ProcessNewReader(NewReader)
9:     src ← NewReader.src
10:    TxIdx* ← NewReader.txid
11:    if TxIdx* < stable-txid then
12:        Send(src, Abort)
13:    else
14:        read-to ← read-to ∪ {src}
15:        Send(src, NewReaderOk)
16:        for r ∈ buffer do
17:            if r.dst = src ∧ r.txid ≥ stable-txid then
18:                Send(src, r)              ▷ forward all buffered messages since stable-txid

19: procedure ProcessNotifySync(NotifySync)
20:    src ←NotifySync.src
21:    if src ∈peers then
22:        perform Syncrhonize
23:    if src ∈read-to then
24:        read-to ← read-to \ {src}
25:    if src ∈read-from then
26:        read-from ← read-from \ {src}

27: procedure ProcessSync(Sync)
28:    src ← Sync.src
29:    TxIdx* ← Sync.txid
30:    if TxIdx* < stable-txid then
31:        Send(src, Abort)
32:    else
33:        Send(src, checkpoints[TxIdx*])
```

Serializability and determinsm proof. We argue both the serializability and determinism of the protocol by showing that ExecutionWorkers process the same input transactions regardless of crash-faults. That is, no ExecutionWorker skips the processing of an input transaction or processes a transaction twice. Both serializability and determinsm then follow from the proofs of Appendix B.

Lemma 16. *No ExecutionWorker skips the processing of an input transaction.*

Proof. Let's assume by contradiction that a worker w with state Objects skips the processing of the input transaction Tx_j. This means that (i) w included in its **read-from** set a worker w' with state Objects$'$, where Objects$'$ is the state Objects after the processing of the list of transactions $\mathsf{Tx}_i, \ldots, \mathsf{Tx}_j$; and (ii) that w processes a ResultMessage from w' referencing transaction Tx_j. This is however a direct contradiction of check Line 8 of Algorithm 7 ensuring that w only includes w' in its **read-from** set after its latest processed transaction **curr-txidx** is at least Tx_j (that is, $\mathsf{Tx}_j \leq$ **curr-txidx**), hence a contradiction.

Lemma 17. *No ExecutionWorker processes the same input transaction twice.*

Proof. Let's assume by contradiction that a worker w with state Objects processes the same input transactions Tx_j twice. This means that (i) w included

Algorithm 9 Synchronization procedure

```
1: procedure SYNCHRONIZE
2:     for w ∈ read-to ∪ read-from do
3:         SEND(w, NotifySync)
4:     read-from ← ∅
5:     read-to ← ∅
6:     while true do
7:         (w, TxIdx*) ← GETSTATUS(my-shard)
8:         Sync ← (TxIdx*)
9:         SEND(w, Sync)
10:        reply ← receive reply from w
11:        if reply ≠ Abort then
12:            LOADCHECKPOINT(w)                          ▷ Load checkpoint from w
13:            for p ∈ peers do
14:                SyncComplete ← TxIdx*
15:                SEND(p, SyncComplete)
16:            replies ← {}                               ▷ wait for SyncComplete responses
17:            while ∄r.(r ∈ replies ∧ r.txid > TxIdx*) do
18:                r ← receive SyncComplete
19:                if ∀p ∈ peers \ suspected. ∃r.(r ∈ replies ∧ r.src = p ∧ r.txid = TxIdx*) then
20:                    return true
```

in its **read-from** set a worker w' with state OBJECTS$'$, where OBJECTS$'$ is the state OBJECTS prior to the processing of the list of transactions $\mathsf{Tx}_i, \ldots, \mathsf{Tx}_j$; and (ii) that w processes a ResultMessage from w' referencing transaction Tx_j. This is however impossible as w could only include w' in its **read-from** set after calling Line 9 (Algorithm 7), and thus after w' updates its state to OBJECTS by processing $\mathsf{Tx}_i, \ldots, \mathsf{Tx}_j$ at Line 18 (Algorithm 8). As a result, w could not have processed a ResultMessage from w' referencing Tx_j while its state is different from OBJECTS.

Lemma 18. *ExecutionWorkers process the same set of input transactions regardless of crash-faults.*

Proof. This lemma follows from the observation that, despite crash-faults, no ExecutionWorker skips any transaction (Lemma 16) nor processes any transaction twice (Lemma 17). As a result, ExecutionWorkers process the same set of input transactions regardless of crash-faults.

Liveness proof. Suppose a worker x crashes. By the strong completeness property of the failure detector, a correct worker e eventually detects this failure, and performs the recovery procedure (Line 5 of Algorithm 7) to find another correct ExecutionWorker of shard s to replace x.We thus argue the liveness property in Lemma 22 by showing that the recovery procedure presented at Line 5 of Algorithm 7 eventually succeeds; that is, it eventually exits at either Line 12 or Line 19. The protocol then resumes normal operation, and the liveness of the system follows from the liveness of the normal operation protocol (Appendix B). As intermediary steps, we show that the procedures GETSTATUS(·) (Line 20 of Algorithm 7), NEWREADER(·) (Line 31 of Algorithm 7), and SYNCHRONIZE (Algorithm 9) eventually terminate.

Lemma 19. *Any call by a correct worker to* GETSTATUS(·) *(Line 20 of Algorithm 7) eventually terminates.*

Proof. We argue this lemma by construction. Let's assume an ExecutionWorker w calls GetStatus(s) on a shard s. It this sends a Recover message to all workers of shards s (Line 22 of Algorithm 7). By Assumption 4, each of these workers eventually receive the messages, and correct ones reply with a RecoverOk message (Line 5 of Algorithm 8). By Assumption 3, there are at least f_e+1 correct workers in shard s. Worker w thus eventually receives at least f_e+1 RecoverOk responses (Assumption 4). Check Line 26 of Algorithm 7 then succeeds and ensures that GetStatus(s) returns.

Lemma 20. *A call* NewReader($w, \cdot$) *(Line 31 of Algorithm 7) to a correct worker w eventually successfully terminates; that is, it returns* ***True****.*

Proof. Suppose a correct worker calls NewReader(w, TxIdx*) for a correct worker w. By construction, the values (w, TxIdx*) are the result of the prior call to GetStatus($\cdot$) (Line 20 of Algorithm 7). Then, given a period of synchrony where messages are delivered much quicker than checkpoint intervals (Assumption 4), TxIdx* is a valid checkpoint at w. As such w responds to the caller's request with NewReaderOk, and the caller successfully terminates.

Lemma 21. *Any call to* Synchronize *(Algorithm 9) eventually terminates.*

Proof. We argue this lemma by construction of Algorithm 9. Let w be a correct worker calling Synchronize. By Lemma 19, the call to GetStatus(**my-shard**) (Line 7 of Algorithm 9) eventually returns. Then, w eventually receives a non-Abort reply (*reply* $\neq$ Abort) at Line 10 given sufficiently many executions of the loop (Line 6) and a period of network synchrony where messages are delivered much quicker than checkpoint intervals on other clusters (Assumption 4). Worker w then loads a remote snapshot, and waits for a set of SyncComplete messages from **peers** $\setminus$ *suspected*. If w receives a message with a larger TxIdx, w retries the synchronization loop at line 6. By the strong completeness property of the failure detector (Assumption 5), w eventually suspect all failed peers, and hence receive all responses from the **peers** $\setminus$ *suspected* set. Moreover, once messages are delivered quicker than the checkpoint intervals within clusters (Assumption 4), all peers undergoing Synchronize will synchronize to the same TxIdx* after sufficient retries.

Lemma 22. *A call to* Recover($\cdot$) *(Line 5 of Line 5) eventually successfully terminates. That is, it eventually exists at either Line 12 or Line 19*

Proof. Consider an ExecutionWorker executing the procedure Recover(x) (Line 5 of Algorithm 7) with $x \in$ **peers** $\cup$ **read-from**. The ExecutionWorker first calls GetStatus($x.shard$) (Line 20) which is guaranteed to terminate (Lemma 19). We then have two cases: (i) the call enters the if-branch (Algorithm 7 Line 8), and (ii) the call enters the else-branch (Algorithm 7 Line 15). We prove that the recovery procedure eventually successfully terminates in both cases. In the first case (i), the ExecutionWorker calls NewReader(w, TxIdx*) (Line 31) which is guaranteed to eventually successfully terminate by Lemma 20. The ExecutionWorker then adds w to its **read-from** set and successfully terminates. In the second case (ii), Line 17 of Algorithm 7 ensures that every correct peer eventually performs the synchronization procedure. Lemma 21 then guarantees that the call to Synchronize (Line 18) eventually terminates. The ExecutionWorker then successfully terminates.

Algorithm 10 Core functions (dynamic objects)

```
1: function TryTriggerExecution(Tx)
2:     // Check if all dependencies are already executed
3:     if HasDependencies(Tx) then return
4:
5:     // Check if all objects are present
6:     M ← MissingObjects(Tx)
7:     if M ≠ ∅ then
8:         for oid ∈ M do Missing[oid] ← Missing[oid] ∪ Tx
9:         return
10:
11:    // Send object data to a deterministically-selected ExecutionWorker
12:    worker ← Handler(Tx)                          ▷ Worker handling the most objects of Tx
13:    O ← {Objects[oid] s.t. oid ∈ HandledObjects(Tx)}                    ▷ May contain ⊥
14:    ReadyMessage ← (Tx, O)
15:    Send(worker, ReadyMessage)
```

E Detailed Dynamic Objects Protocol

This section completes Section 6 by providing the modifications to the algorithms of Appendix A and proving the security of Pilotfish while supporting dynamic reads and writes.

E.1 Algorithms Modifications

We specify the modifications to the algorithms of Appendix A to support dynamic read and writes.

The main difference between Algorithm 10 and Algorithm 3 of Appendix A is the removal of Line 22. Instead of immediately clearing the read locks after accessing the read set's objects, Algorithm 5 removes all read and write locks later.

The main change between Algorithm 11 and Algorithm 4 of Appendix A is the rescheduling of Tx upon discovering a dynamic object. The algorithm first calls UppdateRWSet(Txoid') Line 11 to update the read or write set of Tx with the newly discovered object oid' and then calls RescheduleTx(Tx, oid') at Line 12 to notify all concerned workers that the transaction needs to be rescheduled for execution.

Finally, Algorithm 12 updates the queue Pending[oid'] to trigger re-execution of Tx once oid' is available.

E.2 Proofs Modifications

We specify the modifications to the proofs of Appendix B to prove the serializability, determinism, and liveness (Section 4) of the dynamic reads and writes algorithm. The main modifications arise from the fact that Assumption 2 (Appendix B) is not guaranteed in the dynamic reads and writes algorithm. We instead rely on Assumption 6 below.

Assumption 6 (Transaction References Root) *If transaction Tx dynamically accesses an object oid', it explicitly references its root object oid.*

The Sui MoveVM [43] (used in our implementation) satisfies this assumption. As a result, this part of our design is specific to the Sui MoveVM and cannot directly generalize to other deterministic execution engines unless they implement it as well.

Algorithm 11 Process ReadyMessage (dynamic objects)

```
1: R ← {}                                   ▷ Maps Tx to the object data it refefrernces (or ⊥ if unavailable)

   // Called by the ExecutionWorkers upon receiving a ReadyMessage.
2: procedure ProcessReady(ReadyMessage)
3:     (Tx, O) ← ReadyMessage
4:     R[Tx] ← R[Tx] ∪ O
5:     if len(R[Tx]) ≠ len(R(Tx)) + len(W(Tx)) then return
6:
7:     ResultMessage ← (Tx, ∅, ∅)
8:     if !AbortExec(Tx) then
9:         r ← exec(Tx, ReceivedObj[Tx])
10:        if r = (⊥, oid′) then
11:            UpdateRWSet(Tx, oid′)              ▷ Update R(Tx) or W(Tx) with oid′
12:            RescheduleTx(Tx, oid′)             ▷ Reschedule Tx with discovered oid′
13:            return
14:        (O, I) ← r                             ▷ O to mutate and I to delete
15:        for w ∈ n_e do
16:            O_w ← {o ∈ O s.t. Handler(o) = w}
17:            I_w ← {oid ∈ I s.t. Handler(oid) = w}
18:            ResultMessage ← (Tx, O_w, I_w)
19:    Send(w, ResultMessage)

   // Reschedule execution with discovered object.
20: function RescheduleTx(Tx, oid′)
21:    AugTx ← (Tx, oid′)
22:    for w ∈ n_e do
23:        if ∃oid ∈ Tx s.t. Handler(oid) = w then
24:            Send(w, AugTx)
```

Algorithm 12 Process AugTx (dynamic objects)

```
1: procedure ProcessAugmentedTx(AugTx)
2:     (Tx, oid′) ← augmentedtx
3:     if oid′ ∈ W(Tx) then
4:         Pending[oid′] ← Pending[oid′] ∪ (W, [Tx])
5:     else                                       ▷ oid′ ∈ R(Tx)
6:         (op, T′) ← Pending[oid′][−1]
7:         if op = W then Pending[oid′] ← Pending[oid] ∪ (R, [Tx])
8:         else Pending[oid′][−1] ← (R, T′ ∪ Tx)
9:
10:    // Try to execute the transaction
11:    TryTriggerExecution(Tx)                    ▷ Defined in Algorithm 3
```

Serializability. We replace Lemma 3 (Appendix B) with Lemma 23 below. The rest of the proof remains unchanged. Intuitively, Pilotfish prevents the processing of conflicting transactions until all dynamic objects are discovered. This limits concurrency further than the base algorithms presented in Appendix A but Appendix E.3 shows how to alleviate this issue by indexing the queues Pending[·] with versioned objects, that is, tuples of $(oid, version)$, rather than only object ids.

Lemma 23 (Unlock after Processing). *If a transaction Tx is removed from the pending queue of an object oid then Tx has already been processed (Definition 7 of Appendix B).*

Proof. We argue this lemma by construction of Algorithm 5. By definition (Definition 7), the processing of Tx terminates at Algorithm 5 Line 4. However, the only way Tx can be removed from Pending[oid] is by a call to AdvanceLock(Tx, oid). This call only occurs at one place, at Line 14 of that same algorithm, thus after finishing the processing of tx.

The following corollary is a direct consequence of Lemma 23 and facilitates the proofs presented in the rest of the section.

Corollary 1 (Simulateous Removal). *A transaction Tx is removed from all queues at Line 14 of Algorithm 5.*

Proof. We observe that the proof of Lemma 23 states that the only way to remove Tx from a queue is by calling AdvanceLock(T, oid) and that this call occurs only at one place, at Line 14 of Algorithm 5.

Determinism. The call to $exec(\mathsf{Tx}, O)$ at Line 9 of Algorithm 4 only completes when all objects dynamically accessed by Tx are provided by the set O or are specified as $\perp$. Since objects are uniquely identified by id, we need to show that all honest validators discover the same set of dynamically accessed objects.

Lemma 24 (Consistent Dynamic Execution). *If a correct validator successfully calls $exec(\mathsf{Tx}, O)$ with an dynamically accessed object $o' \in O$ s.t. $o' \neq \perp$ then no correct validators calls $exec(\mathsf{Tx}, O)$ with $o' = \perp$.*

Proof. Let's assume by contradiction that a correct validator A calls $exec(\mathsf{Tx}_j, O)$ (Line 9 of Algorithm 11) with $o' \in O$ s.t. $o' \neq \perp$ while another correct validator B calls $exec(\mathsf{Tx}_j, O)$ with $o' = \perp$. This means that validator A called $exec(\mathsf{Tx}_j, O)$ after processing a previous transaction Tx_i that created o', and that validator B called $exec(\mathsf{Tx}_j, O)$ before processing Tx_i. By Assumption 6 both transactions Tx and Tx' conflict on the root of o', named o, and Lemma 2 (Appendix B) ensures that they are both placed in the same queue Pending[oid] (with $oid = \text{Id}(o)$). Lemma 6 (Appendix B) ensures that both validator hold Tx_j and Tx_i in the same order in Pending[oid], and since validator A processed Tx_i before Tx_j, it means that both validators placed Tx_i in the queue Pending[oid] before Tx_j. However Lemma 23 ensures that Tx_i is not removed from Pending[oid] until processed and thus that validator B executed Tx_j despite Tx_i is still in Pending[oid]. However check HasDependencies(Tx_j) (Line 3 of Algorithm 10) prevents Tx_j from accessing object o (since it is not at the head of the queue Pending[oid]). This is a contradiction of Lemma 5 (Appendix B) stating that Tx_j cannot be executed before accessing o. Since o' was chosen arbitrarily, the same reasoning applies to all objects dynamically accessed by Tx_j.

Lemma 24 replaces the reliance on Assumption 2 in the proof of Theorem 2 (Appendix B).

Liveness. Lemma 12 of Appendix B assumes that all calls to $exec(\mathsf{Tx}, \cdot)$ are infallible. However, supporting dynamic objects requires us to modify Algorithm 4 as indicated in Algorithm 11 and make $exec(\mathsf{Tx}, O)$ fallible. The final Lemma 27 in this paragraph proves that this change does not compromise liveness, since all dynamically accessed objects are eventually discovered, and thus all calls to $exec(\mathsf{Tx}, \cdot)$ eventually succeed.

Lemma 25 (Mirrored Dynamic Object Schedule). *If a transaction Tx_j is placed in a queue Pending[oid'] of a dynamically accessed object oid' after a transaction Tx_i, then Tx_j is also placed in the queue Pending[oid] of the root object oid after Tx_i.*

Proof. Let's assume by contradiction that Tx_j is placed in the queue PENDING[oid'] of a dynamically accessed object oid' after a transaction Tx_i but before Tx_i is placed in the queue PENDING[oid] of the root object oid. By construction, Tx_i can only discover oid' upon execution (Line 9 of Algorithm 11). However, Lemma 5 ensures that Tx_i cannot be executed before accessing oid. This means Tx_i access oid despite Tx_j is already in the queue PENDING[oid']. This is however a contradiction of check HASDEPENDENCIES(Tx_i) (Line 3 of Algorithm 10) ensuring that Tx_i is at head of PENDING[oid] and thus placed in that queue before Tx_j.

Lemma 26 (Dynamic Access at Head of Queue). *When discovering a dynamically accessed object oid' by executing transaction Tx_j and adding Tx_j to queue of* PENDING[oid']*, Tx_j is at the head of the queue* PENDING[oid'].

Proof. Let's assume by contradiction that there exists a transaction Tx_i is at the head of PENDING[oid'] while adding Tx_j to the queue PENDING[oid']. By Assumption 6, both transactions Tx_i and Tx_j conflict on the root of oid', named oid, and Lemma 2 (Appendix B) ensures that they are both placed in the same queue PENDING[oid]. Given that Tx_i is at the head of PENDING[oid'] and that Corollary 1 states that transactions are removed from all queues at the same call, Tx_i is also present in the queue PENDING[oid]. Furthermore, since Tx_i is placed in PENDING[oid'] before Tx_j, Lemma 25 ensures that Tx_i is also placed in the queue PENDING[oid] before Tx_j. Since the discovery of the dynamic object oid' can only occur upon executing a transaction accessing it (at Line 9 of Algorithm 11) and Tx_i is placed in PENDING[oid] before Tx_j, it means that Pilotfish executed Tx_j while Tx_i is still in the queue PENDING[oid]. This is a direct contradiction of check HASDEPENDENCIES() (Line 3 of Algorithm 10).

Lemma 27 (Unlock after Processing). *All objects dynamically accessed by Tx are eventually discovered. That is, eventually $exec(\mathsf{Tx}, \cdot) \neq (\bot, \cdot)$.*

Proof. Lemma 23 ensures that when $exec(\mathsf{Tx}, \cdot)$ (Line 9 of Algorithm 11) returns $(\bot, oid')$, Tx remains at the head of the queue PENDING[oid] (for any $oid \in \mathsf{Tx}$). By construction, this only happens when Tx discovers a dynamic access to object oid'; Tx is then added to the queue PENDING[oid'] (Algorithm 12). Lemma 26 ensures that Tx is at the head of the queue PENDING[oid'] and thus ready for execution by referencing the newly discovered object oid'. As a result, all dynamically accessed objects are eventually discovered, and thus $exec(\mathsf{Tx}, \cdot) \neq (\bot, \cdot)$.

E.3 Versioned Queues Scheduling

This section shows the necessary changes to the algorithms of Appendix E.1 and data structures of Appendix A to move from per-object queues to per-object-version queues. A prerequisite for this is versioned storage of the object data itself, that is, OBJECTS should be a map OBJECTS[$oid, \mathsf{Version}$] $\rightarrow o$ instead of OBJECTS[oid] $\rightarrow o$, which keeps old object versions for as long as they are referenced. Given this, a transaction only writing (not reading) an object does not have to wait on any transaction reading the previous version. An example of this new queueing system can be seen in Figure 13. Also, the resulting dependencies

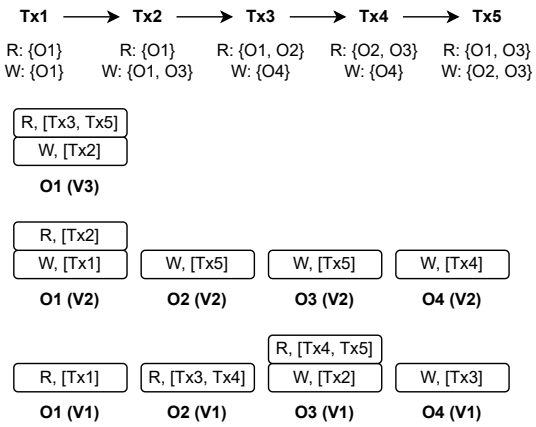


Fig. 13: Example of per-object-version queues, the same transactions as in Figure 3.

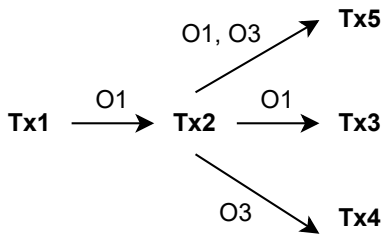


Fig. 14: Example of the happens-before/waiting-on relationship resulting from the per-object-version queues of Figure 13. Edge labels indicate which object is responsible for the dependency.

between transactions can be seen in Figure 14. Without per-version queues all five transactions would have to be executed sequentially instead.

One observation with these per-version queues is that each queue now only contains a single writing transaction (at the very beginning of the queue) and possibly many reading transactions following it. This makes it straightforward to keep track of dependencies between transactions directly, without explicitly creating the queues. We use two maps for this: (a) CurrentWriter$[oid] \rightarrow$ TxIdx keeps track of which transaction is writing the most recent version of any object, and (b) WaitingOn[TxIdx] $\rightarrow$ [TxIdx] keeps for each transaction a list of transactions currently writing object versions it depends on. Additionally, a reverse mapping WaitedOnBy can be used to enable fast deletion from WaitingOn. Once WaitingOn is empty for a transaction, it is ready for execution. When enqueueing a transaction, we can check CurrentWriter for all objects it reads to see which other transactions it needs to wait on. This process is shown in detail in Algorithm 13 and Algorithm 14.

Algorithm 13 Process ProposeMessage (Split-Queues)

```
1: i ← 0                                          ▷ All batch indices below this watermark are received
2: B ← [ ]                                        ▷ Received batch indices

   // Called by ExecutionWorkers upon receiving a ProposeMessage.
3: procedure ProcessPropose(ProposeMessage)
4:     // Ensure we received one message per SequencingWorker
5:     (BatchIdx, BatchId, T) ← ProposeMessage
6:     B[BatchIdx] ← B[BatchIdx] ∪ (BatchId, T)
7:     while len(B[i]) = n_s do
8:         (-, T) ← B[i]
9:         i ← i + 1
10:
11:        // Add the objects to their pending queues
12:        for Tx ∈ T do
13:            for oid ∈ HandledObjects(Tx) do                ▷ Defined in Algorithm 3
14:                Tx' ← CurrentWriter[oid]
15:                if oid ∈ W(Tx) then
16:                    CurrentWriter[oid] ← Tx
17:                if oid ∈ R(Tx) then
18:                    WaitingOn[Tx'] ← WaitingOn[Tx'] ∪ {Tx}
19:                    WaitedOnBy[Tx] ← WaitedOnBy[Tx] ∪ {Tx'}
20:
21:                // Try to execute the transaction
22:                TryTriggerExecution(Tx)                    ▷ Defined in Algorithm 3
```

Algorithm 14 Core functions (Split-Queues, only modified shown)

```
1: j ← 0                                          ▷ All Tx indices below this watermark are executed
2: E ← ∅                                          ▷ Executed transaction indices

3: function HasDependencies(Tx)
4:     return WaitingOn[Tx] ≠ ∅

5: function AdvanceLock(Tx, oid)
6:     // Cleanup the pending queue
7:     T ← ∅
8:     for oid ∈ W(Tx) do
9:         if CurrentWriter[oid] = Tx then                ▷ Tx is still the most recent write
10:            CurrentWriter[oid] ← ⊥
11:    for Tx' ∈ WaitedOnBy[Tx] do
12:        WaitingOn[Tx'] ← WaitingOn[Tx'] \ {Tx}
13:        WaitedOnBy[Tx] ← WaitedOnBy[Tx] \ {Tx'}
14:        if WaitingOn[Tx'] = ∅ then
15:            T ← T ∪ {Tx'}
16:    return T
```