Pilotfish: Distributed Transaction Execution for Lazy Blockchains

Quentin Kniep
qkniep@ethz.ch
ETH Zurich

Lefteris Kokoris-Kogias
lefteris@mystenlabs.com
IST Austria and Mysten Labs

Alberto Sonnino
alberto@mystenlabs.com
Mysten Labs and University College London (UCL)

Igor Zablotchi
igor.zablotchi@gmail.com
Mysten Labs

Nuda Zhang
nudzhang@umich.edu
University of Michigan

ABSTRACT

Pilotfish is the first scale-out blockchain execution engine able to harness any degree of parallelizability existing in its workload. Pilotfish allows each validator to employ multiple machines, named ExecutionWorkers, under its control to scale its execution layer. Given a sufficiently parallelizable and compute-intensive load, the number of transactions that the validator can execute increases linearly with the number of ExecutionWorkers at its disposal.

In addition, Pilotfish maintains the consistency of the state, even when many validators experience simultaneous machine failures. This is possible due to the meticulous co-design of our crash-recovery protocol which leverages the existing fault tolerance in the blockchain's consensus mechanism.

Finally, Pilotfish can also be seen as the first distributed deterministic execution engine that provides support for dynamic reads, as transactions are not required to provide a fully accurate read and write set. This loosening of requirements would normally reduce the parallelizability available by blocking write-after-write conflicts, but our novel *versioned-queues* scheduling algorithm circumvents this by exploiting the lazy recovery property of Pilotfish, which only persists consistent state and re-executes any optimistic steps taken before the crash.

In order to prove our claims we implemented the common path of Pilotfish with support for the MoveVM and evaluated it against the parallel execution MoveVM of Sui. Our results show that our simpler scheduling algorithms outperforms Sui even with a single execution worker, but more importantly provides linear scalability up to 4 ExecutionWorkers even for simple asset-transfers and to any number of ExecutionWorkers for more computationally heavy workloads.

KEYWORDS

Distributed Execution, Deterministic Execution, Blockchains

ACM Reference Format:

Quentin Kniep, Lefteris Kokoris-Kogias, Alberto Sonnino, Igor Zablotchi, and Nuda Zhang. 2024. Pilotfish: Distributed Transaction Execution for Lazy Blockchains . In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 23 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Lazy blockchains [1, 4, 5, 10, 14, 21, 23, 30, 31, 38] separate the problems of data dissemination, consensus, and execution. A potentially distributed external [1, 10, 30] or dedicated [5, 14, 31] subsystem acts as data-dissemination layer to receive clients transactions, persist them as *batches* of transactions, and disseminate them to other validators. Each validator contains a single *Primary* component running Byzantine agreement to sequence the digests of these batches. As a result, transaction data may be scattered across multiple machines (that we call *SequencingWorkers*) and consensus is only achieved on metadata. This architecture allows lazy blockchains to sequence transactions extremely efficiently [14] but considers the retrieval of transactions from the SequencingWorkers and their execution as open problems [14, 23].

To the best of our knowledge, there exists no execution engine tackling these problems without crippling the system's performance and scalability. Existing blockchains' execution engines are designed to run on a single machine that is expected to hold all the transaction data required for execution. Thus, when operating on top of a lazy blockchain, they need to fetch all the data from the remote SequencingWorkers to collocate them on the executor machine. This is an extremely expensive operation that defeats one of the major selling points of lazy blockchains: the separation between data dissemination and consensus. Furthermore, this execution machine needs to operate and persist the entire state of the system, introducing (a) an I/O bottleneck at execution since the state does not fit in the main memory and (b) a storage bottleneck since most machines have a couple of drives mounted. As a result, existing systems fail to handle high throughput in real workloads. They are essentially monolithic wrappers loading all transactions data and executing them by calling a virtual machine, such as the Ethereum virtual machine (EVM) [20] or the Move VM [15]. The EVM is reported to peak at only 20k transactions per second when executing simple transactions (without JIT) and neglecting the retrieval of transaction data [9, 35], which also matches our benchmarks of the MoveVM. These numbers are significantly lower than the throughput of the underlying blockchain, reaching over 100k even in large geo-distributed deployments [4, 14, 21, 23].

Pilotfish is the first blockchain execution engine tackling both the problems of remote data retrieval and distributed execution. It is specifically designed for lazy blockchains and allows each validator to harness multiple machines under its control to scale its

execution layer. These machines are potentially different from the SequencingWorkers used by lazy blockchains to store and disseminate transaction data and are named *ExecutionWorkers*. Specifically, given a sufficiently parallelizable load, the number of transactions that the validator can execute with Pilotfish increases linearly with the number of ExecutionWorkers. Pilotfish achieves this through state sharding [2, 25, 36]. Each ExecutionWorker handles a subset of the state, executes only a subset of the transactions, and contributes its compute and memory to the overall execution engine. As a result, for a compute-heavy workload, Pilotfish can scale execution from 2.5k to 20k transactions per second while maintaining the latency to 50ms by scaling the number of ExecutionWorkers from 1 to 8.

This approach, however, introduces multiple engineering and design challenges. First (i), Pilotfish requires a mechanism to efficiently fetch transactions from the SequencingWorkers and dispatch them only to the appropriate ExecutionWorkers. Existing execution engines do not tackle this problem and simply load all transactions on a single executor machine, thus preventing scalability. Furthermore, since Pilotfish distributes the execution state across multiple machines, it requires a mechanism to dynamically fetch the data required for execution. Pilotfish solves these challenges through a custom protocol that leverages the highly-available commit sequence as a ground truth for coordination. Pilotfish is designed to be used in conjunction with a lazy blockchain, providing it with the sequence of commits authenticating and dictating a total order on transaction data.

Secondly (ii), Pilotfish requires an efficient memory-storage interface that tolerates some SequencingWorkers and ExecutionWorkers crashing and recovering. Existing systems leverage various caching and MVCC techniques [7] that do not naturally generalize to the setting in which Pilotfish operates. Pilotfish needs to handle partial crash-recovery, where each SequencingWorker or ExecutionWorker may crash and recover at a different pace than the others. Pilotfish thus needs to maintain enough state among ExecutionWorker as *checkpoints* to allow recovering machines to catch up with the rest. An simple solution would be to resort to expensive internal replication techniques [28, 32], but Pilotfish goes without it by relying on the globally available commit sequence that is generated by the lazy blockchain.

Finally (iii), Pilotfish aims to support a simpler programming model where the transactions partially specify the input read and write set (e.g., as required for Move [27]). This, however, creates an additional challenge for Pilotfish, as objects that might be accessed dynamically can be located in different ExecutionWorkers, meaning that we cannot overwrite them until all previous transactions have finished, effectively reverting back to sequential execution. Thus, in essence, enforcing write-after-write dependencies to be respected. Fortunately, we can circumvent this issue by leveraging the fact that in-memory execution can be lost in the event of crashes. Hence, we design a novel versioned-queue scheduling algorithm that allows for transactions with write-after-write conflicts to execute concurrently. We couple this with our crash recovery mechanism, which enforces only consistent states to persist. As a result, in the case of a crash, Pilotfish simply needs to redo some computation, 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 ExecutionWorkers, the computational intensity, and the degree of contention of the workload. We find that Pilotfish scales linearly to at least 8 ExecutionWorkers when the workload is compute-bound, and to 4 ExecutionWorkers when the workload is not compute-bound. We compare Pilotfish against a non-scale-out Sui [8] baseline and find that Pilotfish matches or exceeds the performance of the baseline even when running on a single machine, and then continues to outperform the baseline by up to 8× when running on 8 ExecutionWorkers.

Contributions. We make the following contributions:

- We present Pilotfish, the first blockchain execution engine allowing a validator to harness multiple machines under its control to horizontally scale execution.
- We extend Pilotfish to tolerate ExecutionWorker faults recover efficiently.
- We show how Pilotfish supports dynamic reads and writes, thus supporting programming models where the input read and write set is only partially specified.
- We formally prove the serializability, linearizability, and liveness of Pilotfish, even in the presence of faults.
- We provide a full implementation of Pilotfish and empirically demonstrate its scalability through a rigorous evaluation under varying system loads.

2 BACKGROUND AND DEFINITIONS

Figure 1 illustrates the main components of the Pilotfish execution engine. Each component may run on a dedicated machine (e.g., rack servers in a data center) or may be collocated on one or a handful of machines. Each validator is composed of several *SequencingWorkers* that collect (and potentially persist) the transaction data, given the commit sequence from the Primary. In a worker-based lazy blockchain such as Tusk [14], Bullshark [23], Shoal [37] or any Narwhal-based system [14], these would be the transaction dissemination workers. 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.

Pilotfish is specifically designed for lazy blockchains. The Primary only manages metadata (agreement on a sequence of batch digests) allowing it to scale to large volumes of batches and transactions [14]. Actual batch storage is distributed amongst a potentially large number of SequencingWorkers. The key insight regarding the scaling properties of Pilotfish is that the execution of transactions can be distributed amongst a large number of ExecutionWorkers. Pilotfish thus naturally integrates with existing lazy blockchains to scale horizontally. As the number of workers increases, so does the capacity to store state and process transactions.

2.1 System Model

Objects and transactions. Pilotfish validators replicate the state of the system represented as a set of *objects*. Transactions can read and write (mutate, create, and delete) objects, and reference every object by its unique identifier *oid*. A transaction is an authenticated

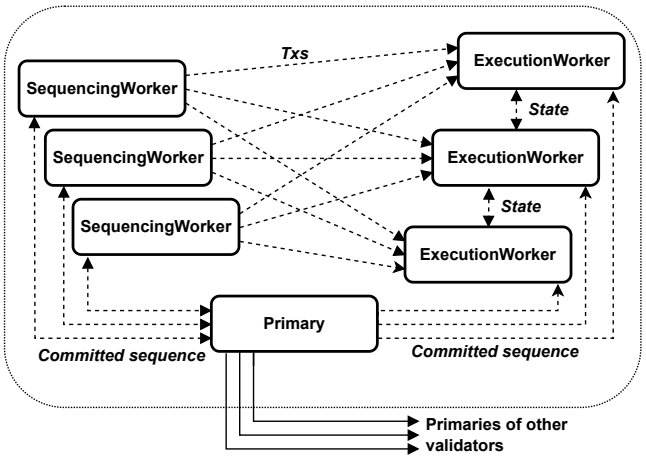


Figure 1: Pilotfish validator's components. Each validator is composed of several SequencingWorkers to fetch and persist the client's transaction, one Primary to run byzantine agreement on metadata, and several ExecutionWorkers to execute transactions. Each component may run on dedicated machines or 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).

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 reads and writes (Section 5) where the read and write set of the transaction is discovered only upon attempting to execute the transaction, adopting the execution model of Sui [8].

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

$$\textsc{Batches}[\mathsf{BatchId}] \rightarrow \mathsf{Batch}$$

mapping the batch digests BatchId to each batch handled by the SequencingWorker. (ii) Additionally, Pilotfish shards its state amongst 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.

Data dissemination and consensus. Most lazy blockchains [5, 10, 14, 23] separate the problem of data dissemination from consensus. This technique allows the Primary to run Byzantine agreement on very small messages (hashes of transaction batches instead of the transactions themselves) while dedicated workers constantly create and share batches in the background. As a result, the data-dissemination layer scales by adding more workers; Narwhal [14] discusses that there would need to be around 12,000 workers before the Primary becomes the bottleneck.

Pilotfish is designed to be integrated with existing lazy blockchains [1, 5, 10, 14, 19, 31]. Specifically, lazy blockchains accept incoming transactions from clients, assemble them into *batches*, and disseminate them to other validators using a variety of machines. Each validator contains a Primary machine running a BFT consensus protocol (treated as a black box) with the Primaries of the other validators. After consensus, the primary of each correct validator obtains the same sequence of batch digests along with the locations storing each batch [5, 14, 30]. Furthermore, there exists a *proof of availability* [13] for each sequenced batch ensuring that a quorum of validators stores it and can be fetched by the SequencingWorkers for execution. This proof of availability may rely on an external system [3, 5] or can be handled by a dedicated subsystem of the validator [14, 23]. Pilotfish builds on top of this architecture by providing a scalable execution layer.

2.2 Threat Model

We assume a message-passing system with a set of validators running a consensus protocol to totally order clients' transactions (solid arrows on Figure 1). Each validator is internally composed of several machines (connected by dotted arrows in Figure 1) and each machine is responsible for a shard of the state.

For the first part of the paper we assume that machines cannot crash-fail. Later in Section 4, we expand each logical shard to have a set of $n_e = 2f_e + 1$ replicas such that as long as for each shard there are f_e+1 replicas available the system remains live and safe[1]. In case this threshold is breached the validator can still synchronize the stalled shards with the rest of the validators of the lazy blockchain through a standard recovery procedure [14] that is out of scope.

2.3 Core Properties

Pilotfish guarantees basic serializability, linearizability, and liveness properties. Intuitively, serializability means that Pilotfish execution produces the same result as a sequential execution. Linearizability 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 2.1 (Pilotfish Seriazability). A correct validator executing the sequence of transactions $[\mathsf{Tx}_1, \ldots, \mathsf{Tx}_n]$ holds the same store Objects as if the transactions were executed sequentially, in the given order.

[1]Note that f_e here refers to number of validator-internal per shard replicas that may crash, and may be different from the number of validators in the blockchain that may be Byzantine (usually denoted by f).

Definition 2.2 (Pilotfish Linearizability). No two correct validators that executed the same sequence of transactions $[\mathsf{Tx}_1, \ldots, \mathsf{Tx}_n]$ have different stores Objects.

Definition 2.3 (Pilotfish Liveness). All correct validators receiving the sequence of transactions $[\mathsf{Tx}_1, \ldots, \mathsf{Tx}_n]$ eventually execute all the transactions $\mathsf{Tx}_1, \ldots, \mathsf{Tx}_n$.

Section 6 proves that Pilotfish satisfies these properties. Finally, Pilotfish is horizontally scalable, that is, the number of transactions that a validator can execute increases with the number of ExecutionWorkers it controls (given a sufficiently parallelizable workload). We demonstrate this property empirically in Section 8.

3 THE PILOTFISH SYSTEM

We introduce Pilotfish, the first scale-out distributed execution engine designed for lazy blockchains. Pilotfish is coupled with a consensus protocol to execute the committed sequence of transactions. Figure 2 illustrates the full transaction life cycle in Pilotfish, from the moment it is sequenced to when it is executed. The part of the validator that runs consensus (called the *Primary*) sends the committed sequence to all its SequencingWorkers and ExecutionWorkers (❶). This section presents the Pilotfish protocol through the operations performed by the SequencingWorkers and ExecutionWorkers at steps ❷, ❸, ❹, and ❺ of Figure 2. Appendix A presents the full algorithms.

ExecutionWorkers handle their objects by maintaining the following key-value stores:

- $\textsc{Objects}[oid] \rightarrow o$ making all the objects handled by the ExecutionWorker accessible by their unique identifier.
- $\textsc{Pending}[oid] \rightarrow [(op, [\mathsf{Tx}])]$ mapping each object to a list of pending transactions [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.
- $\textsc{Missing}[oid] \rightarrow [\mathsf{Tx}]$ mapping objects that are missing from Objects to the transactions that reference them. It is used to track transactions that cannot (yet) be executed because they reference objects that are not yet available. It is cleaned after execution.

Step ❷ of Figure 2: 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 the batch does not contain any transactions with objects handled by ExecutionWorker j, the list is empty, but the message must be sent anyway to inform all workers that they can proceed to the next transaction and provide liveness in the face of malformed transactions created by malicious clients. The full algorithm describing how workers handle the committed sequence is in Algorithm 2 in Appendix A.

Step ❸ of Figure 2: Schedule execution. Each ExecutionWorker j waits for a ProposeMessage from each SequencingWorker i. They then parse every transaction Tx included in the message (in the order specified by the message) and extract all the objects of its read set $\mathcal{R}_j$ and write set $\mathcal{W}_j$ handled by ExecutionWorker j (and ignore 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* (Line 15 of Algorithm 3, see Appendix A):

$$\textsc{Pending}_j[oid] \leftarrow \textsc{Pending}_j[oid] \cup (\mathsf{W}, \mathsf{Tx})$$

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. The full algorithm describing how ExecutionWorkers handle a ProposeMessage from the SequencingWorkers and schedule transactions execution can be found in Algorithm 3 and Algorithm 4 in Appendix A.

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 (Line 18 of Algorithm 3):

$$\textsc{Pending}_j[oid] \leftarrow \textsc{Pending}_j[oid] \cup (\mathsf{R}, \mathsf{Tx})$$

indicating a read-after-write dependency. However, if the entry is a read then the transaction 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 $\textsc{Pending}_j$ store (Line 19 of Algorithm 3):

$$\textsc{Pending}_j[oid][-1] \leftarrow (\mathsf{R}, [\mathsf{Tx}', \mathsf{Tx}])$$

A transaction Tx is ready to be executed when it reaches the head of the pending lists of all the objects it references (Line 5 of Algorithm 4). At this point, the ExecutionWorker loads from its $\textsc{Objects}_j$ store all the objects data it handles (Line 15 of Algorithm 4):

$$O_j \leftarrow \{\textsc{Objects}[oid] \text{ s.t. } oid \in \textsc{HandledObjects}(\mathsf{Tx})\}$$

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 $\textsc{Objects}_j$ store, the ExecutionWorker waits until it all transactions sequenced before Tx are executed (Line 28 of Algorithm 4, see Appendix A) and then sends $\perp$ instead of the objects data. This

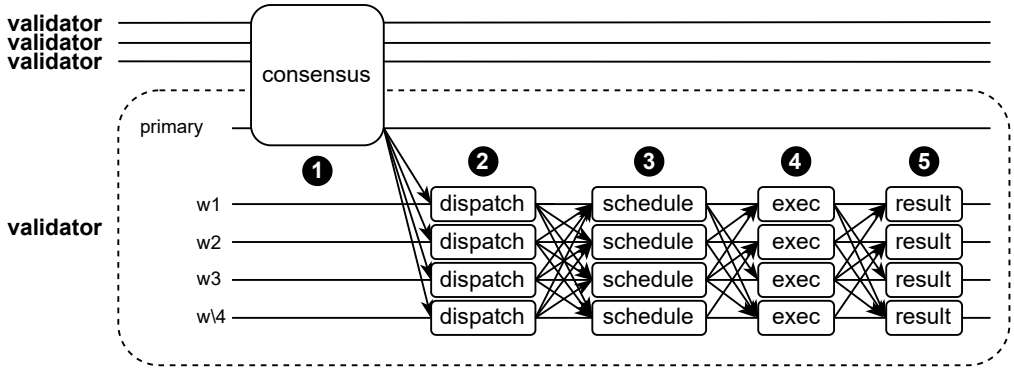


Figure 2: Pilotfish overview. Every validator runs with 5 machines: one machine running the Primary and 4 machines running workers. Each worker machine collocates 1 SequencingWorker and 1 ExecutionWorker. The Primary runs a byzantine agreement protocol to sequence batch digests (❶). SequencingWorkers 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-elected 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).

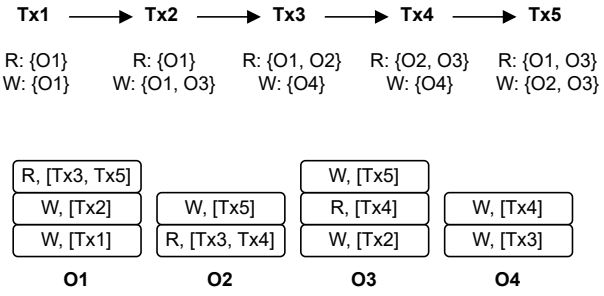


Figure 3: Example snapshot of the PENDING queues of an ExecutionWorker. Pilotfish schedules the execution of the sequence $[\mathsf{Tx}_1, \mathsf{Tx}_2, \mathsf{Tx}_3, \mathsf{Tx}_4, \mathsf{Tx}_5]$. The ExecutionWorker stores Tx_1 as $(\mathbf{W}, [\mathsf{Tx}_1])$ in the queue of oid_1 as it only 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 $(\mathbf{R}, [\mathsf{Tx}_3, \mathsf{Tx}_4])$ 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.

signals that Tx is malformed and references non-existent objects or objects that should have been created but the origin transaction failed.

Step ❹ of Figure 2: Execute transactions. The ExecutionWorker receives the first ReadyMessage message, it waits to receive one ReadyMessage from all other ExecutionWorkers handling at least one object referenced by Tx (Line 5 of Algorithm 5). At this point, the set of ReadyMessage provides the ExecutionWorker with the objects' data behind all objects referenced by Tx, or $\perp$ if an object is (deterministically) considered unavailable. If all object data are available (Line 8 of Algorithm 5), the ExecutionWorker simply executes the transaction in its next available CPU core; otherwise it aborts the execution of Tx. Executing a transaction produces a set of objects to mutate or create O and a set of object ids to delete I (Line 9 of Algorithm 5):

$$(O, I) \leftarrow exec(\mathsf{Tx}, O')$$

The ExecutionWorker then prepares a ResultMessage for all ExecutionWorkers. For the 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 the ExecutionWorker aborts the execution of Tx, it simply sends a ResultMessage to with the transaction Tx and empty sets O and I to all ExecutionWorkers (Line 7 of Algorithm 5). Algorithm 5 in Appendix A describes how ExecutionWorkers handles a ReadyMessage and execute scheduled transactions.

Step ❺ of Figure 2: Handle results. Finally, when an ExecutionWorker receives a ResultMessage, it performs the following operations. If the message includes some mutated objects it (i) persists locally the fact that the transaction has been executed by advancing a watermark of keeping track of all executed transactions (Line 3 of Algorithm 6, see Appendix A); (ii) updates each object into its local OBJECTS store including deletions (Line 4 of Algorithm 6); and (iii) removes all occurrences of the transaction from its PENDING store (Line 14 of Algorithm 6). It then tries to trigger the execution of the next transactions in the queues (Line 15 of Algorithm 6). Algorithm 6 of Appendix A describes how ExecutionWorkers handle a ResultMessage from the ExecutionWorkers and modify their store to reflect the effects of the transaction's execution.

4 CRASH FAULT TOLERANCE

Section 3 presented the design of Pilotfish assuming all data structures are held in memory. However, after long running times, critical components of the validator inevitably wear out and fail. We thus adapt Pilotfish to follow a simple replication architecture, allowing the validator to dedicate multiple machines to each ExecutionWorker. This replication is internal to each validator and allows it to continue its operation despite crash faults.

Pilotfish does not replicate the Primary as it only performs lightweight operations (it does not heavily utilize its CPU, network, or storage) and holds the critical validator's signing key used for consensus. Similarly, it does not replicate SequencingWorkers as

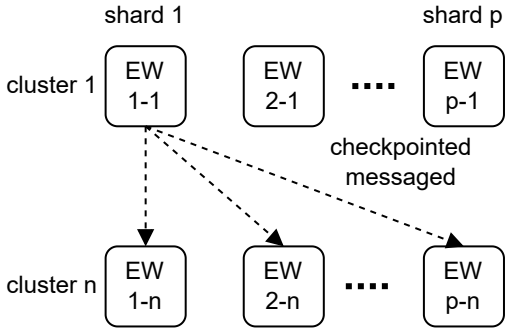


Figure 4: Replication scheme for ExecutionWorkers. The object store is partitioned into shards, and each shard is replicated n_e-folds. Each row represents a cluster, and ExecutionWorkers within a cluster coordinate to process transactions. During normal operation, the only communication between clusters is the sending of CheckpointedMessage, as shown by the dashed arrows.

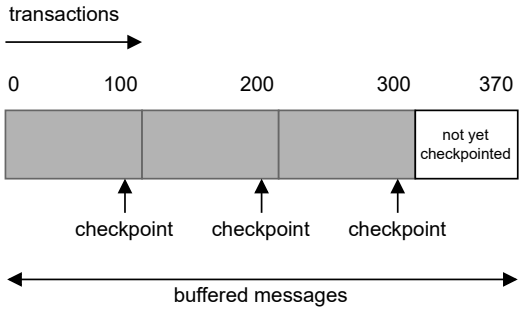


Figure 5: Example of a snapshot of the local state at an ExecutionWorker. It checkpoints its Objects store after every 100 transactions and keeps a buffer of outgoing ReadyMessage.

their work is stateless (assuming a distributed data storage) and idempotent; if one crashes, we can simply boot a new machine that takes over from the latest persisted sequence number.

This section describes the general protocol and Appendix C provides detailed algorithms and security proofs.

4.1 Internal Replication

Figure 4 illustrates the replication strategy of Pilotfish. Each ExecutionWorker is replaced by n_e ExecutionWorkers replicating its operations. Pilotfish operates despite the crash faults of f_e out of $n_e = 2f_e + 1$ ExecutionWorkers. We logically arrange ExecutionWorker replicas in a grid, where each column contains replicas of a shard. Each row in the grid, which contains exactly one other worker from each shard, forms a *cluster*. Under normal operation, a worker serves reads to and receives reads from its cluster peers to process transactions, and each cluster holds a consistent view of the object store.

The naive way to achieve such reliability would be to run a black-box replication engine like Paxos [28] which is also the proposal of the state-of-the-art [42]. 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).

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 [11].

4.2 Normal Operation

During normal operation, ExecutionWorkers within each cluster run the same protocol as in the un-replicated case. They do not communicate with workers in other clusters besides transmitting some additional checkpoint information (detailed below).

ExecutionWorkers maintain the following additional state, allowing two internal recovery flows (Section 4.3). First, they maintain a *buffer* of outgoing ReadyMessage messages, which is a set of ReadyMessage messages that the worker has served to its peers. This tracks the reads served by this worker and allows a prompt recovery in the case of lost messages or transient failures. Second, they maintain a set of *checkpoints*, which are consistent snapshots of the object store that the worker has persisted to disk. The worker maintains a copy-on-write version of the object store, and checkpoints are the only persistent state that the worker maintains.

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

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

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

An ExecutionWorker deems a checkpoint at TxIdx as *stable* after receiving a quorum of $f_e + 1$ CheckpointedMessage[2] from each shard. When a worker learns that a checkpoint is stable, it deletes all checkpoints and buffered messages prior to that checkpoint. This is detailed in Algorithm 1, and illustrated by Figure 6.

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

[2]Quorum sizes can be varied to optimize between normal case disk-usage and recoverability during failure, similar to Flexible Paxos [24].

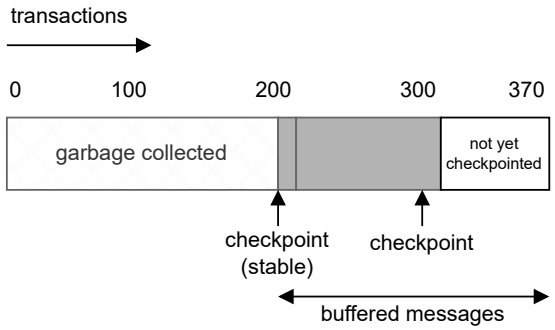


Figure 6: Example of a snapshot of the local state at an ExecutionWorker. It has received a quorum of CheckpointedMessage messages from each shard for the transaction at checkpoint 1. Hence, checkpoint 1 is stable, and the worker safely deletes checkpoints and buffers before it.

Algorithm 1 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)]
```

can be ahead of the slowest cluster in its quorum by up to $c - 1$ checkpoints[3].

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 7 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.

[3]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

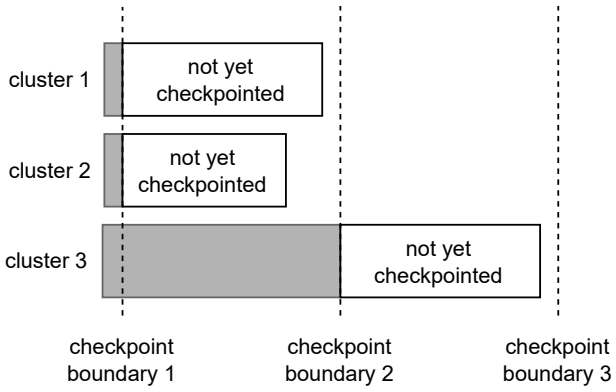


Figure 7: Consider a system with three clusters, and with each ExecutionWorker allowed to hold up to $c = 2$ checkpoints. This snapshot of the clusters' progress shows that ExecutionWorkers in each cluster have a stable checkpoint at boundary 1. Cluster 3 is fast and hence may execute beyond checkpoint 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.

4.3 Failure Recovery

Pilotfish provides two recovery mechanisms: (i) *reconfiguration*, which is a fast recovery mechanism that does not impact the system's throughput but only works when clusters are roughly synchronized, and (ii) *checkpoint synchronization*, which is a slower recovery mechanism that requires a synchronization protocol between clusters. If both recoveries fail, then the system can still recover from other validators.

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 8. In order to detect these crashes, we assume the existence of a failure detector [11] 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 C.

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.

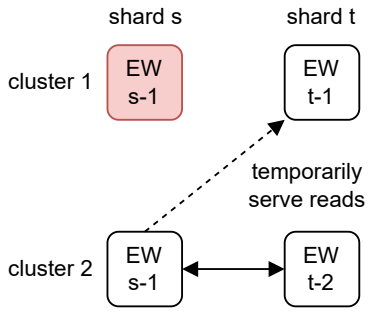


Figure 8: Failure recovery. Suppose $EW_{s,1}$ crashes. As a result, $EW_{t,1}$ cannot receive reads of shard s from $EW_{s,1}$. After $EW_{t,1}$ performs recovery, it establishes a new read relationship with another replica of shard s, in this case $EW_{s,2}$, as illustrated by the dashed arrow. $EW_{t,1}$ 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 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 C.

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 2.2 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., $f_e = 1$).

5 DYNAMIC READS AND WRITES

Within the majority of deterministic execution engines [17, 34, 40, 42], transactions explicitly define the complete set of data they read from and write to. However, this constraint poses limitations on developers and encourages the over-prediction of sets to ensure successful execution. In distributed execution, the problem is exacerbated as we need to transmit the data between ExecutionWorkers. This means that we might need to transmit large read-sets between computers in order to access a single item (e.g., transfer a full array to dynamically access one cell).

Within Pilotfish, we support dynamic reading and writing operations, but we limit them to the hierarchical relationship between parent and child objects. A child object is an object that is owned by another object, known as the parent object. In that case, the child object may only be used if the root object (the initial one in a hierarchy of potentially numerous parents) is included in the transaction and the transaction is granted permission to access the parent. A classic parent-child relationship is that of the parent being the array object and the children being individual cells, dynamically allocated as the array grows. Consequently, transactions in Pilotfish are not prone to overpredictions since they can effectively manage mispredictions by implementing a few fundamental algorithmic modifications.

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*, 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 4. Appendix D 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. That is, child objects are assigned to ExecutionWorkers that maintain a pending queue to schedule their transactions' execution.

The ExecutionWorkers schedule the execution of root objects as usual after processing a ProposeMessage (Algorithm 3) 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.

Upon processing a ReadyMessage (Algorithm 5), the ExecutionWorker attempts to execute the transaction. Upon discovering the need to access a child object in order to proceed with execution, it halts. Subsequently, it generates an UpdateProposeExec message containing an *augmented transaction* Tx+. An augmented transaction includes the typical information found in a traditional transaction, but it also includes the specific identification of each child object that the transaction is known to access, which is added to its read and write sets. This UpdateProposeExec message is sent to

shards handling one of the (newly discovered) child objects. This is safe, as to access the child, the parent should already be locked by the transaction. Therefore, there is an implicit lock on the child, which is just unknown to the shard handling the child. Furthermore, in the event that the transaction has not yet been completed, any subsequent write on the parent will result in both versions being stored in distinct queues, effectively implementing an on-demand multi-versioned concurrency control system.

When the ExecutionWorker receives a UpdateProposeExec message with Tx+, it substitutes any instance of Tx in its queues with Tx+ (typically, there is only one instance of Tx+ in the queues). Additionally, it appends Tx+ to the queue of any child object referenced by the transaction.

Once Tx+ reaches the front of the queues for all objects in its write set, regardless of whether they are child objects or not, the transaction is ready for another execution attempt. Eventually, the protocol identifies every child object that the transaction dynamically accesses, and Tx+ contains their explicit ids. At this point, the transaction can finally be executed.

6 SECURITY ARUGMENTS

We provide an intuition that Pilotfish satisfies the properties of Section 2.3. Appendices B, C.2 and D.2 provide formal proofs.

6.1 Security of the Base Pilotfish Protocol

We informally argue the security of the base Pilotfish protocol presented in Section 3. The full proof can be found in Appendix B.

Serializability. 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 B.1 of Appendix B). The proof leverages the following arguments: (i) Pilotfish builds the pending queues PENDING by satisfying the transaction 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.

Linearizability. We argue that Pilotfish satisfies the linearizability property (Definition 2.2 of Section 2.3). 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 transactions; (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.

Liveness. We argue that Pilotfish satisfies the liveness property (Definition 2.3 of Section 2.1). Intuitively, this property guarantees that valid transactions (Definition B.12) are eventually executed. The proof argues that (i) all transactions are eventually processed (Definition B.13, see Appendix B), and (ii) among those transactions, valid ones are not aborted.

6.2 Modifications for Crash Recovery

We modify the above arguments to show that Pilotfish satisfies the security properties defined in Section 3 despite the crash-failure of f_e out of $2f_e + 1$ ExecutionWorkers in all shards.

Serializability and Linearizability. Appendix C.2 argues both the serializability and linearizability 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. The proof follows from the following observations: (i) ExecutionWorkers persist watermarks of the last transaction they successfully executed (as specified by the serial schedule). These watermarks, together with the knowledge of the commit sequence, prevent them from re-executing old transactions upon recovery. (ii) Appendix C.2 shows that there always exists at least one correct ExecutionWorker holding enough information to allow crash-recovering ExecutionWorkers to synchronize. This information either takes the form of checkpoints or buffered messages (see Section 4).

Livness. The liveness argument relies on the completeness of an eventually strong failure detector [12]. Intuitively, a correct ExecutionWorker w eventually detects the failure of a peer x, and performs the recovery procedure (Line 5 of Algorithm 7, see Appendix C.1) to find another correct ExecutionWorker of shard s to replace x.Appendix C.2 thus argues the liveness property by showing that the recovery procedure presented at Line 5 of Algorithm 7 eventually succeeds. The protocol then resumes normal operation, and the liveness of the system follows from the liveness of the normal operation protocol (Section 6.1).

6.3 Modifications to Support Dynamic Objects

We finally show that the algorithm modifications described in Section 5 and appendix D.1 do not break the serializability, linearizability, and liveness properties defined in Section 2.3.

Serializability and linearizability. The main proof modifications arise from the fact that Assumption 2 (Appendix B) is not guaranteed in the modified dynamic reads and writes algorithm. That is, transactions do not always entirely specify their read and write sets. 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 D.3 shows how to alleviate this issue by indexing the queues PENDING$[\cdot]$ with versioned objects, that is, tuples of $(oid, version)$, rather than only object ids.

Liveness. Lemma B.18 of Appendix B assumes that all calls to $exec(\mathsf{Tx}, \cdot)$ are infallible. However, supporting dynamic objects requires us to modify Algorithm 5 as indicated in Algorithm 11 and make the call $exec(\mathsf{Tx}, O)$ fallible. Appendix D.2 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.

7 IMPLEMENTATION

We implement a networked multi-core Pilotfish execution engine in Rust on top of the Sui blockchain [26]. As a result, our implementation supports Sui-Move [27]. 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`[4] 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 are open-sourcing our Pilotfish implementation along with its orchestration utilities[5].

8 EVALUATION

We evaluate the throughput and latency 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 then show the performance improvements of Pilotfish over the baseline execution engine of Sui [26].

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.

[4]https://tokio.rs

[5] https://github.com/mystenlabs/sui/tree/sharded-execution (commit `cad6b94`)

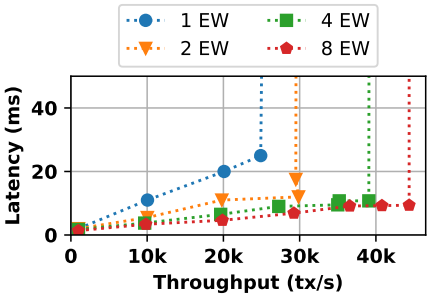


Figure 9: Pilotfish latency vs. throughput with simple transfers.

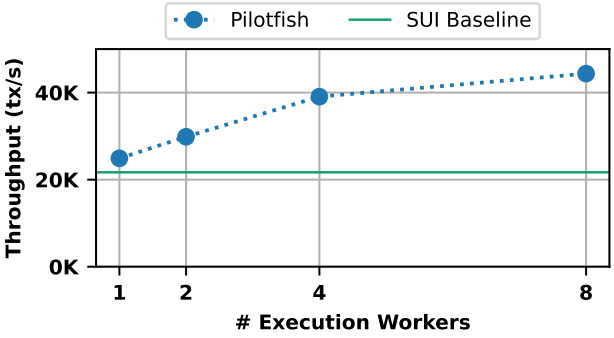


Figure 10: Pilotfish scalability with simple transfers.

When referring to *latency*, we mean the time elapsed from when the client submits the transaction to when the transaction is executed. When referring to *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 from one object to another. We generate the transactions such that no two transactions conflict; each transaction operates on a different set of objects from the other transactions. Thus, this workload is completely parallelizable. Figure 9 shows latency vs throughput of Pilotfish on this workload with 1, 2, 4 and 8 ExecutionWorkers, and Figure 10 shows how Pilotfish's maximum throughput scales when varying the number of ExecutionWorkers. Figure 10 includes as baseline the throughput of the Sui execution engine[6].

We observe that in all but one case, Pilotfish maintains a 20ms latency envelope for this workload. Note that latency exhibits a linear increase as the workload grows for a single execution worker, primarily because of the effects 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 have to wait to get scheduled. This effect no longer exists for higher numbers of ExecutionWorkers, illustrating that adding more hardware has a beneficial effect on service time.

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

Pilotfish scales up to around $45k$ transactions per second. In contrast, the Sui baseline can only process around $20k$ tx/s as it cannot take advantage of the additional hardware. Pilotfish thus exhibits a 2× throughput improvement over the baseline.

Pilotfish's scalability is linear until 4 ExecutionWorkers, but this trend is not maintained for 8 ExecutionWorkers: this indicates that at 4 ExecutionWorkers, the system is no longer compute-bound, and 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 11 and Figure 12 show the results: latency vs throughput and throughput scalability of Pilotfish, respectively. Figure 12 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

In this 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 13 and Figure 14. 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, we do not include the Sui baseline, because Sui's single node benchmark does not allow generating multiple transactions that operate on the same object.

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

9 RELATED WORK

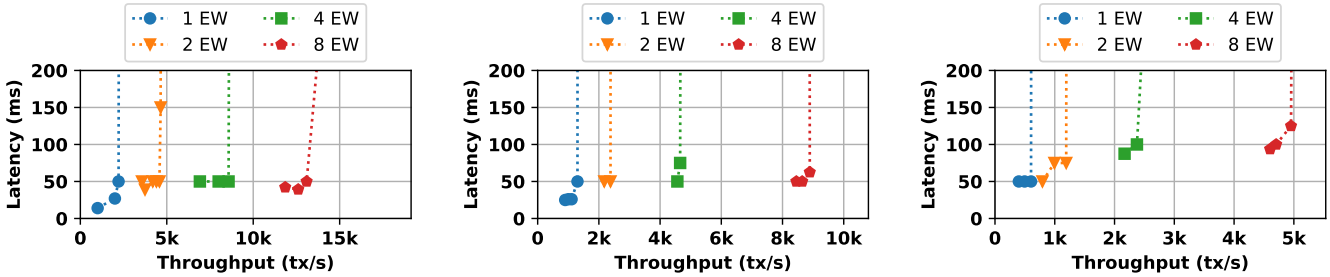
Blockchains executors. Parallel execution in blockchains is a relatively new research area. The main proposal is that of Block-STM [22], however, it is designed with shared memory focus, which makes it hard to adjust for deployment in more than one machine and hence unable to scale out. Nevertheless, one of the core benefits of Block-STM is that it does not need to know the read and write sets. Pilotfish only partially supports it through dynamically accessed objects (Section 5). It remains an open question whether a distributed deterministic execution engine without the need for any hints is practical.

Deterministic databases. Pilotfish is similar to deterministic database systems [41] that employ an order-then-execute approach. This means that they have a sequencing layer that determines a total order for incoming transactions and a scheduling layer that ensures replicas or threads execute transactions in serial equivalence to the total order. Most of these systems are designed for a single-machine setting [17, 18] which means that they cannot scale out. Additionally, all of them require a perfect prediction of write sets to function safely. A small subset of literature has proposed the use of multiple replicas [29, 43] but they only focus on the main system and do not address the question of what happens when some of the servers, especially as the system scales, inevitably crash. Calvin [43] proposes the use of consensus to address crashes, which, as we show, is overkill since the sequencing layer already provides sufficient determinism to recover without strong coordination.

zkVMs and layers 2. Recently, layer-2 solutions [33] and especially the use of zkVMs [6] have been suggested as a way to speed up the execution of blockchains. ZkVMs reduce the problem of execution to that of one prover replica and many verifier replicas that do not re-execute but simply check the proofs. These solutions are not comparable to Pilotfish as they are actually different ways to do deterministic execution and, as a result, compatible with Pilotfish, which could in principle deploy a zkVM instead of the MoveVM. We opted to use these execution engines in our design because zkVMs do not support parallelism due to the way in which proofs are generated. If, in the future, parallel zkVMs become possible, then Pilotfish could be adapted to help them scale out.

10 CONCLUSION

Pilotfish is the first blockchain execution engine allowing a blockchain validator to harness multiple machines under its control to horizontally scale execution. Pilotfish supports dynamic reads and writes, thus supporting programming models where the input read and write set is only partially specified by the transaction. Pilotfish also tolerates crash-faults internal to the validator and is provably serializability, linearizability, and live of Pilotfish. Our implementation of Pilotfish empirically demonstrates its scalability under varying



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

Figure 11: Pilotfish latency vs. throughput for the heavy computation workloads.

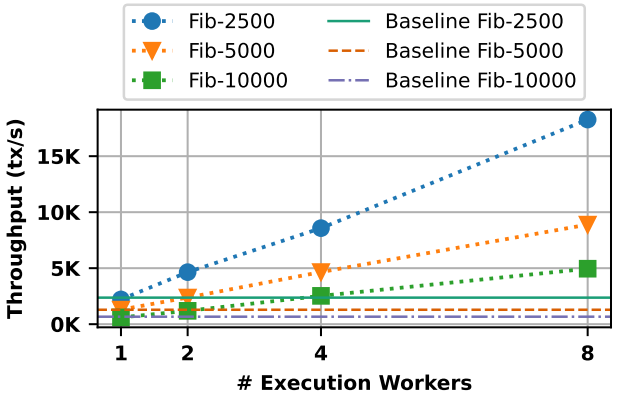


Figure 12: Pilotfish scalability with computationally heavy transactions. `Fib-X` means that each transaction computes the X-th Fibonacci number. The horizontal lines show the single-machine throughput of the baseline on the same workloads.

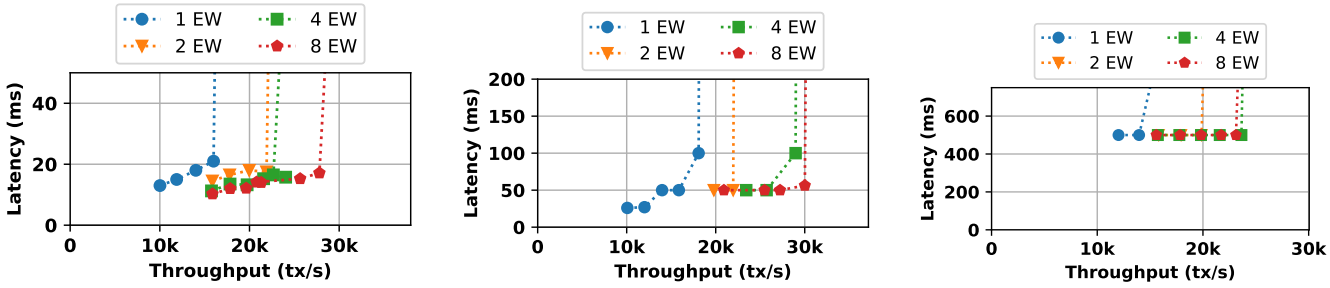
system loads, showing it outperforms the baseline Sui execution engine by up to 10x under heavy CPU loads.

ACKNOWLEDGMENTS

This work is funded by MystenLabs. We thank George Danezis and Dahlia Malkhi for their feedback on the early manuscript. We also extend our thanks Mark Logan and Xun Li who provided insights on the bottlenecks of the Sui blockchain execution engine. Special thanks to Marios Kogias for suggesting the idea of a queue-based locking mechanism to parallelize execution within a machine.

REFERENCES

[1] Mustafa Al-Bassam. 2019. Lazyledger: A distributed data availability ledger with client-side smart contracts. *arXiv preprint arXiv:1905.09274* (2019).
[2] Mustafa Al-Bassam, Alberto Sonnino, Shehar Bano, Dave Hrycyszyn, and George Danezis. 2017. Chainspace: A sharded smart contracts platform. *arXiv preprint arXiv:1708.03778* (2017).
[3] Mustafa Al-Bassam, Alberto Sonnino, and Vitalik Buterin. 2018. Fraud proofs: Maximising light client security and scaling blockchains with dishonest majorities. *arXiv preprint arXiv:1809.09044* 160 (2018).
[4] Kushal Babel, Andrey Chursin, George Danezis, Lefteris Kokoris-Kogias, and Alberto Sonnino. 2023. Mysticeti: Low-Latency DAG Consensus with Fast Commit Path. *arXiv preprint arXiv:2310.14821* (2023).
[5] Vivek Bagaria, Sreeram Kannan, David Tse, Giulia Fanti, and Pramod Viswanath. 2019. Prism: Deconstructing the blockchain to approach physical limits. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 585–602.
[6] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. 2018. Scalable, transparent, and post-quantum secure computational integrity. *Cryptology ePrint Archive* (2018).
[7] Philip A Bernstein and Nathan Goodman. 1983. Multiversion concurrency control—theory and algorithms. *ACM Transactions on Database Systems (TODS)* 8, 4 (1983), 465–483.
[8] Same Blackshear, Andrey Chursin, George Danezis, Anastasios Kichidis, Lefteris Kokoris-Kogias, Xun Li, Mark Logan, Ashok Menon, Todd Nowacki, Alberto Sonnino, et al. 2023. Sui lutris: A blockchain combining broadcast and consensus. *arXiv preprint arXiv:2310.18042* (2023).
[9] Ethereum Foundation Blog. 2023. C++ DEV Update - July edition. https://blog.ethereum.org/2016/07/08/c-dev-update-summer-edition.
[10] Celestia. 2022. The first modular blockchain network. https://celestia.org.
[11] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. 1996. The Weakest Failure Detector for Solving Consensus. *J. ACM* 43, 4 (jul 1996), 685–722. https://doi.org/10.1145/234533.234549
[12] Tushar Deepak Chandra and Sam Toueg. 1996. Unreliable Failure Detectors for Reliable Distributed Systems. *J. ACM* 43, 2 (mar 1996), 225–267. https://doi.org/10.1145/226643.226647
[13] Shir Cohen, Guy Goren, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. 2022. Proof of availability & retrieval in a modular blockchain architecture. *Cryptology ePrint Archive* (2022).
[14] George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. 2022. Narwhal and Tusk: a DAG-based mempool and efficient BFT consensus. In *Proceedings of the Seventeenth European Conference on Computer Systems*. 34–50.
[15] Docs. 2023. Move VM. https://docs.dfinance.co/move_vm.
[16] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. 1988. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)* 35, 2 (1988), 288–323.
[17] Jose M. Faleiro and Daniel J. Abadi. 2015. Rethinking Serializable Multiversion Concurrency Control. *Proc. VLDB Endow.* 8, 11 (jul 2015), 1190–1201. https://doi.org/10.14778/2809974.2809981
[18] Jose M. Faleiro, Daniel J. Abadi, and Joseph M. Hellerstein. 2017. High Performance Transactions via Early Write Visibility. *Proc. VLDB Endow.* 10, 5 (jan 2017), 613–624. https://doi.org/10.14778/3055540.3055553
[19] Matthias Fitzi, Peter Ga, Aggelos Kiayias, and Alexander Russell. 2018. Parallel chains: Improving throughput and latency of blockchain protocols via parallel composition. *Cryptology ePrint Archive* (2018).
[20] Ethereum Fundation. 2023. Ethereum Virtual Machine (EVM). https://ethereum.org/en/developers/docs/evm/.
[21] Yingzi Gao, Yuan Lu, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. 2022. Dumbo-ng: Fast asynchronous bft consensus with throughput-oblivious latency. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*.
[22] Rati Gelashvili, Alexander Spiegelman, Zhuolun Xiang, George Danezis, Zekun Li, Dahlia Malkhi, Yu Xia, and Runtian Zhou. 2023. Block-STM: Scaling Blockchain Execution by Turning Ordering Curse to a Performance Blessing *(PPoPP '23)*. Association for Computing Machinery, New York, NY, USA, 232–244. https://doi.org/10.1145/3572848.3577524
[23] Neil Giridharan, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. 2022. Bullshark: Dag bft protocols made practical. *arXiv preprint arXiv:2201.05677* (2022).
[24] Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman. 2016. Flexible Paxos: Quorum intersection revisited. arXiv:1608.06696 [cs.DC]
[25] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. 2018. Omniledger: A secure, scale-out, decentralized ledger



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

Figure 13: Pilotfish latency vs throughput for the contended workloads. Please note the different y axis ranges between the three cases.

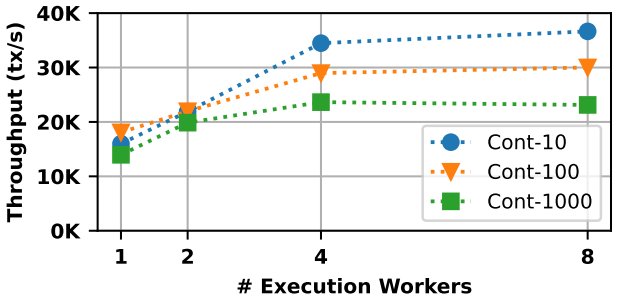


Figure 14: Pilotfish scalability with condended transaction. Each transaction increments a counter. `Cont-X` means that for each counter we submit X increment transactions.

Algorithm 2 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)
```

via sharding. In *2018 IEEE symposium on security and privacy (SP)*. IEEE, 583–598.
[26] Mysten Labs. 2022. Build without boundaries. https://sui.io.
[27] Mysten Labs. 2023. Move Concepts. https://docs.sui.io/concepts/sui-move-concepts.
[28] Leslie Lamport. 2001. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)* (2001), 51–58.
[29] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. 2020. Aria: A Fast and Practical Deterministic OLTP Database. *Proc. VLDB Endow.* 13, 12 (jul 2020), 2047–2060. https://doi.org/10.14778/3407790.3407808
[30] Joachim Neu, Ertem Nusret Tas, and David Tse. 2021. Ebb-and-flow protocols: A resolution of the availability-finality dilemma. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 446–465.
[31] Joachim Neu, Ertem Nusret Tas, and David Tse. 2022. The availability-accountability dilemma and its resolution via accountability gadgets. In *International Conference on Financial Cryptography and Data Security*. Springer, 541–559.
[32] Diego Ongaro and John Ousterhout. 2015. The raft consensus algorithm. *Lecture Notes CS* 190 (2015), 2022.
[33] Joseph Poon and Thaddeus Dryja. 2016. The bitcoin lightning network: Scalable off-chain instant payments. (2016).
[34] Squads Protocol. 2024. What Is SVM - The Solana Virtual Machine. https://squads.so/blog/solana-svm-sealevel-virtual-machine.
[35] Ethereum Research. 2023. EVM performance. https://ethresear.ch/t/evm-performance/2791.
[36] Alberto Sonnino, Shehar Bano, Mustafa Al-Bassam, and George Danezis. 2020. Replay attacks and defenses against cross-shard consensus in sharded distributed ledgers. In *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 294–308.
[37] Alexander Spiegelman, Balaji Aurn, Rati Gelashvili, and Zekun Li. 2023. Shoal: Improving DAG-BFT Latency And Robustness. *arXiv preprint arXiv:2306.03058* (2023).
[38] Christos Stefo, Zhuolun Xiang, and Lefteris Kokoris-Kogias. [n. d.]. Executing and Proving over Dirty Ledgers. ([n. d.]).
[39] Aptos Team. 2024. Aptos. https://aptoslabs.com.
[40] Fuel Team. 2024. The World's Fastest Modular Execution Layer. https://www.fuel.network.
[41] Alexander Thomson and Daniel J. Abadi. 2010. The Case for Determinism in Database Systems. *Proc. VLDB Endow.* 3, 1–2 (sep 2010), 70–80. https://doi.org/10.14778/1920841.1920855
[42] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. 2012. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (Scottsdale, Arizona, USA) *(SIGMOD '12)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/2213836.2213838
[43] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. 2014. Fast Distributed Transactions and Strongly Consistent Replication for OLTP Database Systems. *ACM Trans. Database Syst.* 39, 2, Article 11 (may 2014), 39 pages. https://doi.org/10.1145/2556685

A ALGORITHMS

This section complements Section 3 by providing detailed algorithms for the core components of Pilotfish.

A.1 Detailed Algorithms

The function Handle(oid) of Algorithm 2 returns the ExecutionWorker that handles the specified object identifier oid. The function Index(Tx) in Algorithm 4 returns the index of the transaction Tx in the global committed sequence. The function ID(o) in Algorithm 6 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 6. All indices $i' < \mathbf{i}$ of the list **B** (Algorithm 3) can be cleaned after Line 9 of Algorithm 3 as they are no longer needed. Similarly, any

Algorithm 3 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 4
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 4
```

transactions Tx with index Index(Tx) < **j** can be removed from the set **E** (Algorithm 4) after Line 39 of Algorithm 4. Finally, any transaction Tx can be removed from the map **R** (Algorithm 5) after Line 5 of Algorithm 5.

B SECURITY PROOFS

We show that Pilotfish satisfies the properties of Section 2.3.

B.1 Serializability

We show that Pilotfish satisfies the serializability property (Definition 2.1 of Section 2.3). 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 B.1). 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 B.1 (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}.

Definition B.2 (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 3 builds the pending queues Pending by respecting the transaction dependencies dictated by the consensus protocol (i.e., the sequential schedule).

Lemma B.3 (Sequential Batch Processing). *Pilotfish processes the batch with index $Batch_j$ after processing the batch with index $Batch_i$ if $j > i$.*

Algorithm 4 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-elected 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
```

Algorithm 5 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 =⊥
```

Proof. Let's assume by contradiction that Algorithm 3 processes the ProposeMessage referencing transactions of the batch with

Algorithm 6 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 4
 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]
```

index Batch_j before processing the ProposeMessage referencing transactions of the batch with index Batch_i while $j > i$. This means that Algorithm 3 processes Batch_j at Line 12 before processing Batch_i at Line 12. However, the check of Algorithm 3 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 B.4 (Transactions Order in Queues). *Let's assume two transactions Tx_j, 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 2. They are then placed in the queue Pending[*oid*] by Algorithm 3 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 2 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 2 at Line 9 but Lemma B.3 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 4 accesses objects in the same order as the sequential schedule.

Lemma B.5 (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 4) 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 4 which happens after the access to Objects[*oid*] (Line 15 of the same algorithm). It can then occur (ii) at Line 14 of Algorithm 6 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 4, thus after accessing Objects[*oid*] (Line 15 of that same algorithm). □

Lemma B.6 (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 4 at Line 15 after successfully passing the 'dependencies' check at Line 5. Lemma B.5 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 5 executes transactions in the same order as the sequential schedule.

Lemma B.7 (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 5. Transactions are executed only at Line 9 of Algorithm 5 and this algorithm is only triggered upon receiving a ReadyMessage. However, creating the latter message only occurs at Line 17 of Algorithm 4, thus after accessing Objects[*oid*] (Line 15 of that same algorithm). □

Theorem B.8 (Serializability). *If a correct validator executes the sequence of transactions $[Tx_1, \ldots, Tx_n]$, 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 $\text{Tx}_i \rightarrow \text{Tx}_j$ 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 B.4 ensures that Tx_j is placed in the queue Pending[*oid*] after Tx_i. Lemma B.6 then guarantees that Tx_j's access to *oid* occurs after Tx_i's access on *oid*. However Lemma B.7 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 Linearizability

We show that Pilotfish satisfies the linearizability property (Definition 2.2 of Section 2.3). 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* exec(*Tx*, *O*) *(Line 9 of Algorithm 5) 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 [20], the SVM [34], and both the MoveVM [15] (used by the Aptos blockchain [39]) and the Sui MoveVM [27] (used by the Sui blockchain [8]). All execution engines except the Sui MoveVM also fulfill Assumption 2 (Section 5 and Appendix D remove this assumption to make Pilotfish compatible with Sui).

We rely on the following lemmas to prove linearizability in Theorem B.11.

Lemma B.9. *Given the same sequence of transactions* [$Tx_1, \ldots, Tx_n$], *all correct validators build the same execution schedule (that is, they build same queues Pending).*

Proof. We argue this property by construction of Algorithm 2 and Algorithm 3. Since all validators receive the same input sequence [$\mathsf{Tx}_1, \ldots, \mathsf{Tx}_n$] and Algorithm 2 respects the order of transactions (Line 8), all correct validators create the same sequence of ProposeMessage. Lemma B.3 then ensures that Algorithm 3 processes each ProposeMessage respecting the transaction order. Finally, Lemma B.4 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 B.10. *No two correct validators creating the same ResultMessage (Line 13 of Algorithm 5) obtain a different object state Objects.*

Proof. We argue this property by construction of Algorithm 6 and by assuming that the communication channel between all ExecutionWorkers of each validator preserves the order of messages[7]. Once a validator creates a ResultMessage (Line 13 of Algorithm 5), it is processed by Algorithm 6. This algorithm first calls TryAdvanceExecWatermark(·) (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 B.11 (Pilotfish Linearizability). *No two correct validators that executed the same sequence of transactions* [$Tx_1, \ldots, Tx_n$] *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 B.9 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 3), all correct validators create a ReadyMessage referencing Tx_n and the set of objects *O* (Line 17 of Algorithm 4). 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 5 with the same input and thus execute the same sequence of transactions. By construction of Algorithm 5 and Assumption 2, they all call calls to *exec*(Tx, *O*) (Line 9 of Algorithm 5) with the same inputs Tx and *O*. Given that Assumption 1 ensures that all calls to *exec*(Tx, *O*) are deterministic, all correct validators thus create the same ResultMessage (Line 13). Finally, Lemma B.10 ensures that all validators creating the same ResultMessage obtain the 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 4) and ResultMessage (Line 13 of Algorithm 5) 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 2.3 of Section 2.1). Intuitively, this property guarantees that valid transactions (Definition B.12) are eventually executed. The proof argues that (i) all transactions are eventually processed (Definition B.13), and (ii) among those transactions, valid ones are not aborted.

Definition B.12 (Valid Transaction). A transaction *T* with index $idx = \textsc{Index}(T)$ is valid if all objects referenced by its read and write set are created by a transaction T' with index $idx' < idx$.

Definition B.13 (Processed Transaction). A transaction *T* is said *processed* when it is either executed or aborted and the object state Objects is updated accordingly.

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

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 B.18.

Lemma B.14. *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 $\mathsf{Tx}_i \rightarrow \mathsf{Tx}_j$ 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 G contains no cycles to prove liveness. That is, it is sufficient to show that G contains no edges $\mathsf{Tx}_j \rightarrow \mathsf{Tx}_i$, where $j > i$. Let's assume by contradiction that G has an edge $\mathsf{Tx}_j \rightarrow \mathsf{Tx}_i$, where $j > i$. Then, by rule (1) of the construction of G, Tx_j and Tx_i must conflict on some object *oid*. Lemma B.4 ensures that Tx_j is placed in the queue in Pending[*oid*] after Tx_i . Lemma B.6 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 $\mathsf{Tx}_j \rightarrow \mathsf{Tx}_i$ 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 B.15. *If a transaction T is processed (Definition B.13), 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 4 and Algorithm 6. 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 4 to effectively release read locks. This call occurs right after creating a ReadyMessage referencing T (Line 17), a necessary step to trigger Algorithm 5 and thus process T. (ii) The second call occurs at Line 14 of Algorithm 6 to effectively release write locks. This call occurs right after updating Objects[*oid*] and thus terminating the processing of T. □

Lemma B.16. *If the sequence of transactions $[\mathsf{Tx}_1, \ldots, \mathsf{Tx}_n]$ is processed (Definition B.13), the watermark **j** (Line 1 of Algorithm 4) is advanced to n.*

Proof. The processing of T involves updating the object state Objects (Line 4 of Algorithm 6). However, the watermark **j** is only updated upon calling TryAdvanceExecWatermark(Tx_i) at Line 3 of this same algorithm. Thus, by construction, the buffer **E** contains every processed transaction Tx_i (Line 38 of Algorithm 4), and once the sequence $[\mathsf{Tx}_1, \ldots, \mathsf{Tx}_n]$ is processed, the watermark **j** is advanced to $\mathbf{j} = \max\{\text{Index}(\mathsf{Tx}_1), \ldots, \text{Index}(\mathsf{Tx}_n)\} = n$ (Line 39 of Algorithm 4). □

Lemma B.17. *A transaction T is eventually processed (Definition B.13) if it has neither missing dependencies nor missing objects that could be created by earlier transactions. That is, T is eventually processed if Algorithm 4 creates a ReadyMessage referencing T.*

Proof. We argue this lemma by construction of Algorithm 5 and Algorithm 6. Algorithm 5 receives a ReadyMessage from all ExecutionWorkers and the check Line 5 of Algorithm 5 passes. Transaction T is then either executed (if the check Line 16 passes) or aborted (if the check Line 16 fails), and Algorithm 5 creates a ResultMessage referencing T (Line 13). Algorithm 6 then receives this ResultMessage and accordingly updates its object Objects (after an infallible call to TryAdvanceExecWatermark(T) Line 39). □

Lemma B.18 (Eventual Transaction Processing). *All correct validators receiving the sequence of transactions $[\mathsf{Tx}_1, \ldots, \mathsf{Tx}_n]$ eventually process (Definition B.13) all transactions $\mathsf{Tx}_1, \ldots, \mathsf{Tx}_n$.*

Proof. Lemma B.14 ensures that the transaction scheduling process is deadlock-free (no circular dependencies) and Pilotfish thus triggers their execution (Line 3 of Algorithm 4). We are then left to prove that these scheduled transactions are processed (Definition B.13). Since Theorem B.8 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 B.15 thus ensures these transactions are removed from all queues Pending[·]. As a result, when triggering the execution of Tx_n (Line 3 of Algorithm 4), the check HasDependencies(Tx_n) (Line 5 of Algorithm 4) returns **False** (since $\forall oid \in \mathcal{R}(\mathsf{Tx}_n) \cup \mathcal{W}(\mathsf{Tx}_n) : \text{Pending}[oid][0] = \mathsf{Tx}_n$). Furthermore, since all transactions Tx_i with $i \leq n-1$ are already processed, Lemma B.16 ensures that the watermark $j = n-1$ (Line 1 of Algorithm 4) and thus MissingObjects(Tx_n) returns $\emptyset$. Finally, Algorithm 4 creates a ReadyMessage referring Tx_n and thus Lemma B.17 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 4); and Lemma B.17 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 B.21.

Lemma B.19. *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 B.13).*

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 6. This call is thus after Algorithm 6 updates Objects[*oid*] (at Line 4) and thus after the transaction is processed. □

Lemma B.20. *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 B.13). In which case, Lemma B.6 states that both T and T' are placed in the same queue Pending[oid] and Lemma B.19 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 4). □

Lemma B.21. *A valid transaction T it is never aborted; that is the call AbortExec(T) (Line 8 of Algorithm 5) 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 5 found at least one missing object ($\perp$) referenced in the read or write set of T, and thus that Algorithm 5 received at least one ReadyMessage message referring $\perp$ instead of an object data. However, Lemma B.20 ensures that the call to Objects[oid] (Line 15 of Algorithm 4) never returns $\perp$ if oid is referenced by the read or write set of a valid transaction, hence a contradiction. □

Livness proof. We finally combine Lemma B.18 and Lemma B.21 to prove liveness.

Theorem B.22 (Liveness). *All correct validators receiving the sequence of transactions $[Tx_1, \ldots, Tx_n]$ eventually execute all the valid transactions of the sequence.*

Proof. Lemma B.18 ensures that all correct validators eventually process (Definition B.13) all transactions $\mathsf{Tx}_1, \ldots, \mathsf{Tx}_n$. Lemma B.21 ensures that valid transactions are never aborted and thus executed. □

C DETAILED RECOVERY PROTOCOL

This section completes Section 4 by providing the algorithms allowing ExecutionWorkers to recover from crash-faults and proving the security of Pilotfish in this setting.

C.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. This is achieved through a call to $(w, \mathsf{TxIdx}^*) \leftarrow$ GetStatus(x.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 NewReader(w, 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.

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:        Syncrhonize()
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
```

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 LoadCheckpoint(w) procedure.

Finally, e brodcasts a synchronization completion messages to all peers and awaits their responses. If an incoming SyncComplete

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*])
```

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
```

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.

C.2 Proofs Modifications

We show that Pilotfish satisfies the security properties defined in Section 3 despite the crash-failure of f_e out of $2f_e + 1$ ExecutionWorkers in all shards.

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 (Partial Synchrony). *The network between ExecutionWorkers is eventually synchronous [16].*

Assumption 5 (Eventually Strong Failure Detector). *There exists an eventually perfect failure detector $\Diamond\mathcal{S}$ with the following propertiy.* Strong completeness: *Every faulty process is eventually permanently suspected by every non-faulty process.*

Serializability and linearizability proof. We argue both the serializability and linearizability 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 linearizability then follow from the proofs of Appendix B.

Lemma C.1. *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 C.2. *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 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 C.3. *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 C.1) nor processes any transaction twice (Lemma C.2). 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 C.7 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 C.4. *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 C.5. *A call NewReader(w, ·) (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(·) (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 C.6. *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 C.4, 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** \ *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** \ *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 C.7. *A call to Recover(·) (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 C.4). 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 C.5. 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 C.6 then guarantees that the call to Synchronize (Line 18) eventually terminates. The ExecutionWorker then successfully terminates. □

D DETAILED DYNAMIC OBJECTS PROTOCOL

This section completes Section 5 by providing the modifications to the algorithms of Appendix A and proving the security of Pilotfish while supporting dynamic reads and writes.

D.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 4 of Appendix A is the removal of Line 22. Instead of immediately clearing the read locks after accessing the read set's objects, Algorithm 6 removes all read and write locks later.

The main change between Algorithm 11 and Algorithm 5 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 re-scheduled for execution.

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

D.2 Proofs Modifications

We specify the modifications to the proofs of Appendix B to prove the serializability, linearizability, and liveness (Section 3) 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.

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-elected 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)
```

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 4
```

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

The Sui MoveVM [27] (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.

Serializability. We replace Lemma B.5 (Appendix B) with Lemma D.1 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 D.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 D.1 (Unlock after Processing). *If a transaction Tx is removed from the pending queue of an object oid then Tx has already been processed (Definition B.13 of Appendix B).*

Proof. We argue this lemma by construction of Algorithm 6. By definition (Definition B.13), the processing of Tx terminates at Algorithm 6 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 D.1 and facilitates the proofs presented in the rest of the section.

Corollary D.2 (Simulateous Removal). *A transaction Tx is removed from all queues at Line 14 of Algorithm 6.*

Proof. We observe that the proof of Lemma D.1 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 6. □

Linearizability. The call to $exec(\mathsf{Tx}, O)$ at Line 9 of Algorithm 5 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 D.3 (Consistent Dynamic Execution). *If a correct validator successfully calls exec(Tx, O) with an dynamically accessed object $o' \in O$ s.t. $o' \neq \perp$ then no correct validators calls exec(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 B.4 (Appendix B) ensures that they are both placed in the same queue Pending[oid] (with $oid = \text{Id}(o)$). Lemma B.9 (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 D.1 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 B.7 (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 D.3 replaces the reliance on Assumption 2 in the proof of Theorem B.11 (Appendix B).

Liveness. Lemma B.18 of Appendix B assumes that all calls to $exec(\mathsf{Tx}, \cdot)$ are infallible. However, supporting dynamic objects requires us to modify Algorithm 5 as indicated in Algorithm 11 and make $exec(\mathsf{Tx}, O)$ fallible. The final Lemma D.6 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 D.4 (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 B.7 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 D.5 (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 B.4 (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 D.2 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 D.4 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 D.6 (Unlock after Processing). *All objects dynamically accessed by Tx are eventually discovered. That is, eventually $exec(Tx, \cdot) \neq (\bot, \cdot)$.*

Proof. Lemma D.1 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 D.5 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 $exec(\mathsf{Tx}, \cdot) \neq (\bot, \cdot)$. □

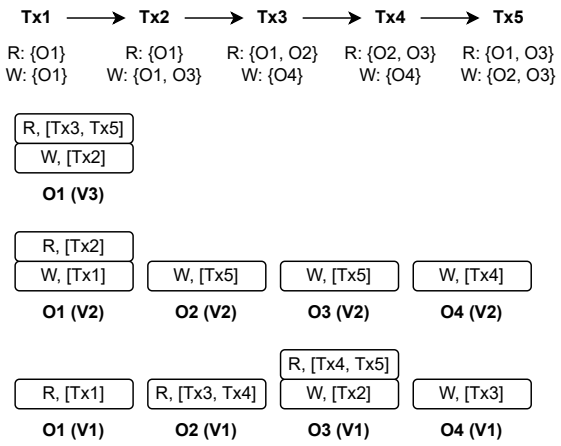


Figure 15: Example of per-object-version queues using the same transactions as in Figure 3.

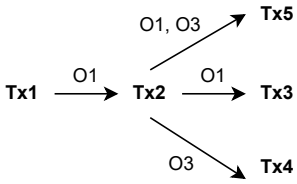


Figure 16: Example of the happens-before/waiting-on relationship resulting from the per-object-version queues of Figure 15. Edge labels indicate which object is responsible for the dependency.

D.3 Versioned Queues Scheduling

This section shows the necessary changes to the algorithms of Appendix D.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$[oid, \text{Version}] \rightarrow o$ 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 15. Also, the resulting dependencies between transactions can be seen in Figure 16. 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

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 4
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 4
```

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
```

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] → TxIdx keeps track of which transaction is writing the most recent version of any object, and (b) WaitingOn[TxIdx] → [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.