Thunderbolt: Causal Concurrent Consensus and Execution

Junchao Chen
Exploratory Systems Lab
University of California, Davis

Alberto Sonnino
Mysten Labs
University College London (UCL)

Lefteris Kokoris-Kogias
Mysten Labs
IST Austria

Mohammad Sadoghi
Exploratory Systems Lab
University of California, Davis

ABSTRACT

In the realm of blockchain systems, smart contracts have gained widespread adoption owing to their programmability. Consequently, developing a system capable of facilitating high throughput and scalability is of paramount importance. Directed acyclic graph (DAG) consensus protocols [15, 33, 33, 34, 53, 54] have demonstrated notable enhancements in both throughput and latency, however, the serial execution is now becoming a bottleneck. Numerous works have endeavored to execute by constructing a dependency graph to trace concurrent transactions [19, 41, 63, 64]. However, approaches prove impractical for smart contracts by assuming that read/write sets are known in prior [10, 55].

This paper introduces Thunderbolt, a novel architecture based on DAG-based protocols, that aims to furnish a scalable and concurrent execution for smart contract transactions. Inspired by Hyperledger [6], Thunderbolt also expands Execute-Order-Validate architecture in which transactions are distributed into distinct replicas, with execution outcomes determined prior to ordering through the DAG-based protocol. Existing protocols adopt serial executions after the ordering to avoid non-determinism. However, Thunderbolt provides parallel pre-execution before the ordering as well as parallel verifications once any source of non-determinism is removed. Each replica validates the transaction results during the construction of the DAG other than after the ordering following the construction to improve the latency. In an effort to enhance smart contract execution, we implement an execution engine that constructs a dependency graph to dynamically assign transaction orders, thus mitigating abort rates due to execution conflicts. Additionally, we introduce a novel shard reconfiguration to withstand malicious attacks by relocating replicas from the current DAG to a new DAG, and rotating the shards among different replicas.

Our comparison of the results on SmallBank with serial execution on Narwhal-Tusk [15] revealed a remarkable 50 times speedup with 64 replicas.

KEYWORDS

concurrency control, two-phase locking, smart contract

Conference'17, July 2017, Washington, DC, USA
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

ACM Reference Format:
Junchao Chen, Alberto Sonnino, Lefteris Kokoris-Kogias, and Mohammad Sadoghi. 2024. Thunderbolt: Causal Concurrent Consensus and Execution. In . ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Consensus protocols have gained significant attention in the field of distributed systems as a means of constructing reliable systems. These protocols are considered a universal primitive and have been extensively researched to achieve higher throughput, lower latency, and scalability. When working on blockchain use cases, delivering high-performance and scalable solutions for engineers and scientists is crucial. This is particularly important in the context of smart contracts which is first proposed in [56] and have been widely used in industries [14, 20, 60]. Smart contracts are self-executing contracts with codes containing user functions written in a stack-based bytecode language. Executing the contracts needs to analyze the bytecode, leading to lower performance than executing the native transactions. Thus, providing a high-performance and scalable system to execute smart contracts is crucial.

Recently, directed acyclic graph (DAG) consensus protocols [7, 8, 15, 33, 34, 38, 50, 52–54] have exhibited significant advancements in both performance and robustness to asynchrony and leader failures.

One of the unique characteristics of these protocols is their ability to enable every replica to generate blocks in rounds that reference the blocks in previous rounds, forming a DAG. Furthermore, these protocols separate data dissemination from the core consensus logic, which is the primary bottleneck for leader-based protocols. This enables all replicas to disseminate data simultaneously while the consensus component only orders a smaller amount of metadata. One of the significant advantages of this architecture is that it delivers remarkable throughput and supports scaling out participants by adding more workers as part of a single validator.

However, integrating these protocols into an end-to-end smart contract platform is challenging for the following reasons. Firstly, malicious clients may degrade the goodput of the system by submitting the same transaction to all replicas. This is a known caveat of these systems [10]. Secondly, the excellent scalability of DAG-based systems enhances data dissemination parallelism. However, transaction execution remains a bottleneck, as it executes transactions in a linear order after the consensus logic. Finally, their architecture requires replicas to reach agreement over blocks of transactions rather than over-execution results and thus their safety strongly relies on the determinism of the smart contracts. Despite most smart contract platforms being theoretically deterministic, a common source of bugs can cause non-deterministic behaviors [61].

Recent works have attempted to address these issues. Sui [10] prevents clients from degrading the goodput of the system by introducing an extra round of consistent broadcast before proposing a transaction to the consensus protocol. This extra round is used to de-duplicate transactions from malicious clients. However, this approach introduces an extra round of communication which can degrade the latency of the system. Several works have attempted to improve the execution of smart contracts by constructing a dependency graph between transactions [19, 41, 63, 64]. Unfortunately, this approach requires transactions to provide their read/write sets in advance to eliminate non-determinism, which is impractical when dealing with smart contracts. As a result, smart contracts cannot benefit from improvements that require prior knowledge of read/write sets. In contrast to other blockchain frameworks, Hyperledger [6] has introduced an innovative framework, called Execute-Order-Validate, that enables transaction execution prior to reaching a consensus. In order to enhance parallelism, a set of executors executes transactions locally using Optimistic Concurrency Control (OCC) [36]. The results are then gathered by a primary node, which runs a consensus protocol to establish a global order. After the order is determined, the transactions will be validated and committed. However, as isolated executors execute these transactions, transaction conflicts are possible.

In this paper, we propose Thunderbolt, a new architecture based on DAG-based protocols that addresses all the above challenges. Inspired by Hyperledger, Thunderbolt adopts the executions before the consensus. However, Thunderbolt shards transactions into distinct shards and transactions in each shard will be executed by a replica, called a shard leader, to avoid contention between shards and natively prevent malicious clients from degrading the goodput of the system by submitting the same transactions to all replicas. We also leverage round-robin scheduling to rotate the shard leader periodically or do so on demand if a malicious shard leader is detected to enhance the system's security and liveness. Thunderbolt leverages the properties of DAG to migrate the current DAG to a new DAG without a hard stop to rotate the leaders of each shard. This DAG switching mechanism allows our protocol to be built on top of any DAG protocols.

Thunderbolt only focuses on single-shard transactions in which transactions in different shards are disjoint. However, Thunderbolt does not exactly have physical shards but it is logical because every replica contains all the data.

Thunderbolt runs in rounds and in each round, transactions in each shard will be executed in parallel by the assigned leader of each shard before generating a block containing those transactions. Then each block will be delivered to the DAG-based protocol, like Narwhal [15] and Bullshark [53], to broadcast to all other replicas and obtain a total order. The DAG will preserve the causal order of the transactions in each shard by linking the blocks in round r to the blocks in the previous round. Different from Hyperledger, in which the verification is triggered after the ordering, replicas in Thunderbolt verify the transaction results while generating the nodes in the DAG to improve the latency. This architecture allows the system to reach an agreement over execution results as well as over the order of the transactions. As a result, the safety of the system is not anymore dependent on the determinism of the smart contracts.

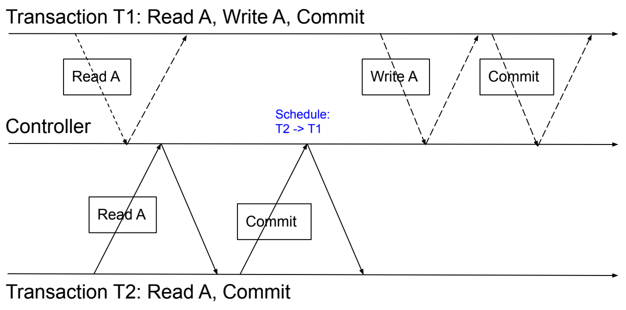

Transaction T1: Read A, Write A, Commit
Read A
Write A
Commit
Controller
Schedule:
T2 -> T1
Read A
Commit
Transaction T2: Read A, Commit

Figure 1: Nondeterministic Ordering of Thunderbolt.

Moreover, to enhance the execution of smart contracts which does not provide any read/write sets beforehand, we implement a novel concurrent executor that generates a dependency graph dynamically by the shard leader to allow non-conflict transactions to be processed in parallel and later be verified in parallel by all the replicas. This engine will reassign the execution order based on the state between transactions to reduce the number of aborts. For example, in Figure 1, T_2 will be assigned before T_1 to avoid being aborted although T_2 arrives later than T_1. This smart execution reduces the conflict rate among the transactions, increasing the throughput of the system.

The challenges in Thunderbolt are as follows. First, in Thunderbolt, transactions are distributed and processed by specific replicas to ensure optimal efficiency. If a replica becomes malicious, it becomes necessary to reassign shard leaders. Most selection mechanisms rely on timers and an additional consensus protocol to reach an agreement on the new leader after the timers expire. Thus, the primary challenge lies in devising a method to select the new logical shard leader without waiting for the timer to expire and avoiding the need for additional consensus, while ensuring that the new leader possesses the most recent state. Thunderbolt halts the current DAGs by broadcasting Shift blocks and utilizes a round-robin scheduling to rotate logical shard leaders in the new DAGs. The new DAG will not initiate until the majority of the replicas have entered, thereby ensuring that each replica in the new DAG obtains the most recent state of each shard, as the old shard cannot progress without obtaining the votes from the majority.

Second, the execution engine is a critical component within Thunderbolt, tasked with generating a dependency graph to facilitate the execution of transactions without prior knowledge of read/write sets. The primary challenge faced by the engine lies in effectively scheduling transactions within the graph to minimize the occurrence of aborts. To address this challenge, we implement a dynamic dependency graph that leverages the data accessed by the transactions. Upon the addition of a new operation (read/write), we meticulously adjust the graph by rescheduling transactions to ensure minimal aborts.

In summary, this paper makes the following contributions.

- Thunderbolt is the first sharded consensus protocol built on any DAG protocols, allowing transactions to be executed in parallel with undetermined order before consensus within the shards. Thunderbolt also leverages DAG to rotate the shard leaders without additional time-out messages to avoid malicious replicas by starting a new DAG.

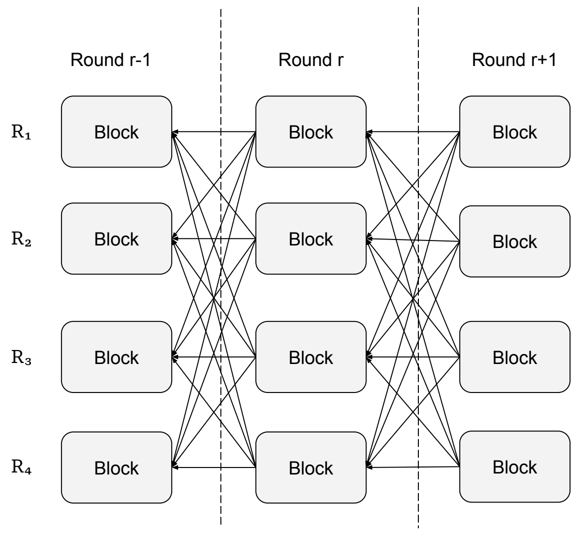


Figure 2: Overview of a DAG-based protocol.

- We implemented an execution engine to improve the parallelism of executing smart contracts without any read/write sets knowledge. The execution engine dynamically arranges the transactions based on the current assessments to reduce the abortion rates due to the conflicts.
- Our evaluation of Thunderbolt yields a remarkable 50x speedup over a sequential execution with SmallBank workload on 64 replicas built on Apache ResilientDB (Incubating) [1, 28].

2 DAG-BASED BFT CONSENSUS

We introduce DAG-based BFT consensus protocols and their properties that Thunderbolt leverages.

DAG as Mempool. DAG-based BFT consensus protocols aim to decouple the network communication from the consensus. These protocols, such as Narwhal [15, 53], BBCA-Chain [38], Shoal/Shoal++[7, 52], Mysticeti [9], Cordial Miners [34], and Motorway [22] propose blocks in rounds and each block consists of a collection of transactions and references to previous blocks. These blocks form an ever-growing DAG, with blocks serving as vertices and the references between blocks serving as edges. The causal history of a block B refers to the sub-graph that starts from v. During processing the block B_r in round r, blocks in previous rounds in the causal history of B_r will also processed implicitly. Figure 2 illustrates an overview of a round-based DAG.

Narwhal [15] is one of the DAG-based protocols that introduces a certificate block to reduce the payload of references in the blocks needed to be broadcast. In Narwhal, each data block including transactions must obtain at least $2f+1$ certificates from different replicas to generate a certificate block in each round r. This certificate block provides proof of its validation among the majority of the replicas. Then, each replica R must obtain at least $2f + 1$ certificate blocks from distinct replicas belonging to round $r - 1$ and include these blocks in the references of the data block in round r. Next, R broadcasts the data block and obtains its certificates.

BBCA-LEDGER [54] is a Byzantine log replication technology enabling blocks to be broadcast in parallel using Byzantine consistent broadcast [11] and a DAG is created to address the empty slots. Cordial Miners [34] and Mysticeti [9] reduce the latency from the reliable broadcast by only taking a single round of communication per DAG round.

DAG as Consensus. As each edge in the DAG represents a vote, the DAG also serves as a consensus protocol. Thus, each replica can determine the total order of all blocks in the DAG without communication with others. The consensus protocol guarantees that all the replicas will receive the same order of the committed blocks.

Tusk [15], Bullshark [53], Shoal [52], and Shoal++ [7] are algorithms built on Narwhal. DAG-Rider and Tusk introduce waves, which interprets every 4 rounds in DAG-Rider and 3 rounds in Tusk, and leverage random coins to select a replica as a leader and commit its block and its causal history in the first round of each wave. Bullshark provides a deterministic protocol variant with a lower-latency ordering rule relying on partial synchrony for liveness to improve the long tail latency from Tusk. Shoal and Shoal++ further improve this latency through pipelining and a reputation-based leader election module. Additionally, BBCA-LEDGER also leverages the ordering framework of Bullshark to drive the fallback consensus on top of its DAG.

DAG-based protocol properties.

- Validity: if an honest replica R has a block B in its local view of the DAG, then R also has all the causal history of B.
- Block Consistency: if an honest replica R obtains a block B_r in round r from replica P, then eventually all other honest replicas will have B_r.
- Completeness: if two honest replicas have a block B_r in round r, then the causal histories of B_r are identical in both replicas.

3 THUNDERBOLT OVERVIEW

Thunderbolt is designed to improve the efficiency of the smart contract execution by 1) introducing sharded concurrent execution. 2) live migration to reconfigure shards. Thunderbolt relocates the execution before ordering, thereby eliminating the sequential execution from the total order created from consensus protocols (section 2).

Thunderbolt comprises three major components, namely preplay, execution scheduling, and validation, which are illustrated in Figure 3. During each round r, a replica R executes a batch of transactions and generates a block B_r, which contains the execution outcomes, then R transmits B_r to other replicas via the DAG-based consensus protocol (section 3.2). Any other replica P will validate B_r in parallel during the consensus and persist the results once B_r is committed.

To eliminate contention between transactions in different replicas, as outlined in section 4, transactions are distributed into different shards. Each shard is responsible for maintaining disjoint transactions and will be assigned to a replica to address those transactions related to that shard. This paper only focuses on concurrently executing single-shard transactions. We shift the responsibility of

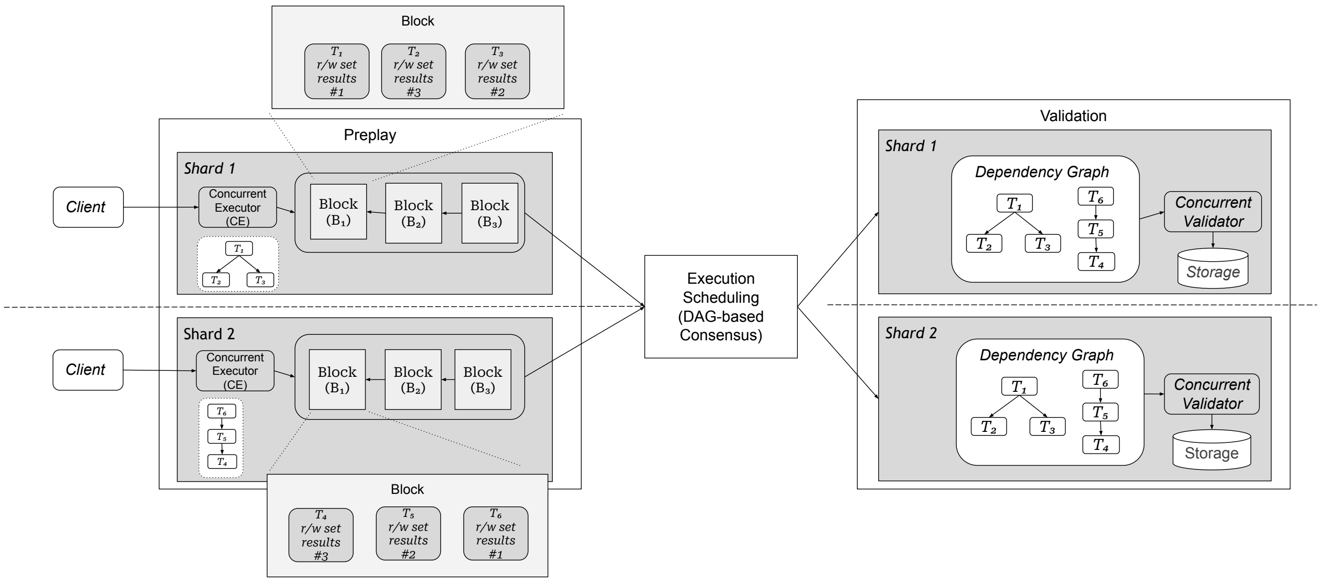


Figure 3: Transactions are partitioned into two individual shards in two replicas. Transactions in each shard will be executed and obtain their execution outcomes while creating the blocks. Next, the execution outcomes from all the replicas will obtain a global order through a DAG-based consensus protocol and will be validated by each replica before being stored globally.

handling cross-shard transactions to the users. Users must split the cross-shard transactions into smaller disjoint transactions and handle the consistency among these transactions.

Thunderbolt also live migrates each shard to another replica letting transactions be executed in different replicas if a malicious shard leader is detected, to protect from malicious replicas (section 4). This approach ensures that Thunderbolt remains efficient, secure, and reliable.

Thunderbolt is a versatile protocol that can be directly applied to any DAG-based consensus protocol, such as any Narwhal-based protocol, BBCA-Ledger, Mysticeti, and Cordial Miners, requiring only minor adjustments in the generation of the DAG nodes

3.1 System Model, Goals, and Assumptions

Threat model. We assume a set of n replicas, of which at most f are faulty, $n = 3f+1$. The f faulty replicas can perform any arbitrary (Byzantine) failures, while the remaining replicas are assumed to be honest and follow the protocol's specifications at all time. We assume an eventually synchronous network [18] that messages sent from a replica will eventually arrive in a global stabilization time (GST), which is unknown to the replicas. We also assume communications between replicas go through authenticated point-to-point channels, and messages are authenticated by a public-private key pair signed by the sender.

Design goals. We guarantee basic serializability, Safety and liveness properties. Intuitively, serializability means that execution produces the same result as a sequential execution across all the replicas. Safety means that every correct node receiving the same sequence of transactions performs the same state transitions. Liveness means that all correct nodes receiving a sequence of transactions eventually execute it.

Definition 1 (Seriazability). An honest replica holds the same validation outcomes when executing the same block of transactions.

Definition 2 (Safety). All honest replicas agree on the same block of transactions in each round.

Definition 3 (Liveness). Each honest replica will eventually decide a block of transactions.

Appendix B proves that Thunderbolt satisfies these properties.

Data model. The data model assumes that each transaction includes a contract code with functions to access data in the shard belonging to the sender. The contract involves two types of operations: $<Read, K>$ and $<Write, K, V>$. Here, K represents the key required for access, and V is the value that needs to be written to the key K. The contract code is Turing-complete and any information could not be obtained without execution.

Thunderbolt groups users and their data into distinct shards. Thunderbolt only focuses on single-shard partitionable workload containing keys within the same shard. To distribute the data into different shards, users need to assign the transaction a shard id SID to indicate where the keys should be accessed.

3.2 DAG Integration

Preplay. In Thunderbolt, each replica R as a shard leader is responsible for executing transactions and obtaining execution outcomes while generating a block B_r prior to disseminating it through DAG at round r. The process for creating blocks is illustrated in Figure 4.

```
Processing the transactions of shard $S$ on replica $R$ :
Let txn_list be a transaction list storing the transactions from clients.
event Receive a transaction $T$ of shard $S'$ from client do
    if $S \neq S''$ then
        Redirect $T$ to the shard leader of $S'$.
        Return.
    Append $T$ to txn_list.

event Receive a batch $B$ of transaction from txn_list do
    outcomes $O$ = Execute ($B$)
    Deliver $B_r =< B, O, r >$ to DAG($B_r, r$)
event Receive a Shard reconfiguration $S'$ do
    Start a new DAG
    Start to process the transactions of shard $S'$.
```

Figure 4: Preplay.

```
event Receive a valid block $B_r$ of shard $S$ sent from replica $R$ at round $r$ from DAG do
    if $R$ is not the shard leader of $S$ at round $r$ then
        Return invalid
    Build the dependency graph $G$ based on the read/write set $S$ in $B_r$.
    Execute the transactions simultaneously using $G$ and verify the results.
    if All the results are matched with the ones included in $B_r$ then
        Return valid
    else
        Return invalid

event Commit blocks $B$ from the committer at round $r$ from consensus do
    for Each block $b$ from round $r'$ of sub-DAG $j$ in $B$ do
        if $b$ is a Shift block then
            num_committed_shift_block += 1
            Continue
        Update the values in the write sets to the storage.
    if num_committed_shift_block == $2f + 1$ then
        Reconfig the shard leader to the next shard.
        num_committed_shift_block=0
```

Figure 5: Validate blocks.

It is worth noting that R only executes transactions assigned to the shards it leads. For transactions that are not assigned to R, it redirects them to the corresponding shard.

Each replica runs a concurrent executor (CE) to execute transactions in batches and generates detailed outputs for each transaction. These outputs include read/write sets, scheduled order, and operation results. The scheduled order determines the execution order of the transactions inside the batch results that should returned to the users during the execution, while the read/write sets provide the keys that each transaction accessed.

Execution Scheduling. Thunderbolt supports any DAG-based data dissemination layer equipped with a consensus protocol (section 2) to determine the total order of blocks among replicas. In each round r, R delivers B_r to the DAG to generate a node in the graph that contains edges to all the blocks in previous rounds, including the ones that R proposed in round $r - 1$ (section 2). Since the block needs to obtain votes from a majority of the replica, each replica will validate the results included in the block before sending their votes. It's essential to note that from the completeness property (section 2), block B_{r-1} in round r should be validated before block B_r from the same shard since block B_r will have a link to the block B_{r-1} from the same shard to the execution.

```
Proposing a block $B$ at round $r$ on replica $R$ :

event Receive $2f + 1$ blocks $\{B\}$ at round $r - 1$ do
    need_shift = false
    if $\{B\}$ contains $f + 1$ Shift blocks then
        need_shift = true
    else
        for each replica $R$ do
            if Do not receive any block from $R$ after round $r - K$ then
                need_shift = true
            if $K'$ blocks have been proposed then
                need_shift = true
    if need_shift = true and $R$ does not send $B_{shift}$ in the current DAG then
        Generate $B_{shift}$ and deliver to other replicas
    else
        Deliver block $B$ to other replicas
    Go to the next round
```

Figure 6: Broadcast the Shift block at round r if blocks from some replicas are missing from round $r - K$ or K' blocks have been proposed to make a periodical rotation.

Validation. Upon receiving block B_r of round r through the DAG, the verification process is initiated by the system to ensure the execution results of each transaction within the block are correct. This process is accomplished via the read/write sets to construct a dependency graph locally, as depicted in Line 4 in Figure 5. The graph serves to identify transactions that can be processed in parallel instead of sequentially validating them, thereby improving the overall performance.

To verify these transactions, Thunderbolt leverages the native OCC protocol [36], but not limited to [36], which executes a set of validators to verify the transactions in parallel. If the transaction fails to pass the verification process, it is discarded.

Thunderbolt verifies the blocks during the data dissemination, validating the execution outcomes when receiving the blocks, and thus in parallel with the consensus protocol to improve the latency. Data in the write sets are persisted in the storage once it is committed by the consensus protocol (Line 15).

It should be noted that a valid dependency graph consistently generates the same results on the read sets that obtain the same values on the keys and the final values written on each key are the same as the ones recorded in the block that the transactions provide. Thus, if a mismatch in the values from the read sets is produced, the block will be disregarded and the shard leader will be flagged as faulty.

4 SHARDS RECONFIGURATION

Thunderbolt segregates users and their associated data into distinct shards, with each shard being allocated to a specific replica serving as a shard leader. Each shard leader only proposes the transactions related to that shard. Thunderbolt employs a round-robin selection mechanism [49] to rote shard leaders when a leader fails to propose transactions for K rounds. Additionally, Thunderbolt implements a rotation of shard leaders to preempt potential harm from malicious leaders at intervals of K' rounds, where $K' > K$. This is the key technique enabling Thunderbolt to prevent malicious clients from submitting the same transactions to all replicas of the system and degrade its goodput. Each shard leader can locally perform transactions deduplication to prevent the same transaction from

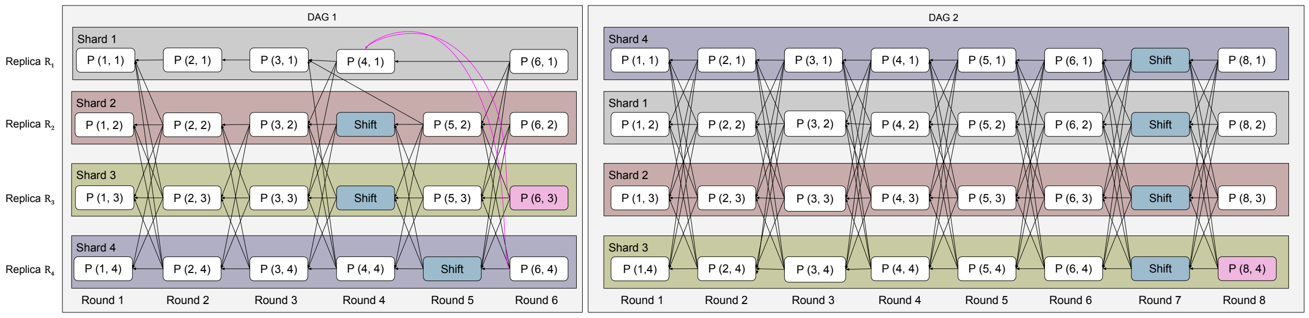


Figure 7: Each replica will propose a Shift block at round r if the block from a shard leader from round $r-2$ ($K=2$) does not arrive or has received 2 Shift blocks at round $r-1$. Shard reconfiguration is triggered when a block (pink block at round 6), including Shift blocks from 3 replicas, is committed, and a new DAG (DAG 2) is generated. All the results from the uncommitted blocks, like $P(6,1)$, will be discarded before starting the transactions in DAG 2. In DAG 2, although every replica proposes blocks in every round, Shift blocks will still be sent after $K^{'}=6$ blocks have been proposed to rotate the shard leaders to avoid censorship attacks.

being proposed multiple times, which is a key open challenge of DAG-based protocols [8, 10, 53].

In a Byzantine environment, the security and integrity of a replica may be compromised by malicious attacks. Once a replica falls under the control of malicious actors, the transactions within the assigned shard may become susceptible to censorship attacks, such as the dropping of blocks post-execution or avoiding proposing selected transactions. Thunderbolt implements a strategy of rotating the shard leader regularly or if a malicious leader is detected.

Diverging from traditional consensus protocols that depend on notification messages to alter primary nodes, Thunderbolt introduces an innovative mechanism that leverages the underlying DAG protocols to facilitate the seamless transition of live migration to reconfigure the shard leaders. The selection of a new leader is based on a round-robin approach that if the current shard leader is replica R_i, the subsequent leader will be $R_{(i \ mod \ n)+1}$.

However, the transmission of blocks to a new leader may experience delays or omissions due to network issues or the actions of a malicious leader. If the new leader for round r is unable to receive the proposal committed in round $r-1$ from the previous leader, operations will be halted until the block arrives to ensure safety.

In addressing this challenge, Thunderbolt implements Shift blocks to facilitate agreements among replicas regarding when a shard reconfiguration should be initiated and switch to a new DAG to process further transactions. The implementation is shown in Figure 6.

A replica R broadcasts a Shift block in round r under the following conditions:

(1) R does not receive any block of a replica after round $r-K$.
(2) R has proposed blocks for at least $K^{'}$ rounds.
(3) R received $f+1$ Shift blocks from distinct replicas at round $r-1$.
(4) R does not broadcast the Shift block before.

In the depicted scenario shown in Figure 7 where $K=2$ and $K^{'}=6$, a replica triggers the broadcasting of a Shift block. During round 4, replica R_2 and R_3 do not receive any blocks in rounds 2 and 3. Subsequently, R_2 and R_3 broadcast a Shift block to other replicas. During round 5, despite replica R_4 having received blocks at round 4 from replica R_1, it still broadcasts the Shift block because it has received 2 Shift blocks at round 4 to ensure the liveness.

After broadcasting the Shift block to claim a malicious replica is detected, each replica will continue broadcasting valid blocks with transactions to the current DAG and consensus. To guarantee the safety of the system, each replica must switch to the new DAG at the same round. We leverage the first commit block that includes $2f+1$ Shift blocks to be the ending round for each replica working on the current DAG. Since all the honest replicas will commit the same block at the same round (section 2), they will start the new DAG at the same following round. Thus, each replica will keep proposing transactions until it has received a committed block that contains $2f+1$ Shift blocks in its causal history. For example, R_2 will propose $P(5,2)$ at round 5 after proposing a Shift block at round 4. Finally, the block $P(6,3)$ from R_3 at round 6 is selected as the committer during the consensus and commits all the history including the Shift blocks from other replicas. Then all the replicas will switch to the new DAG (DAG 2) and start executing the transactions within the new shard.

Prior to entering the new DAG, each replica needs to wait for all the transactions in the history of the committed block to be committed and all the uncommitted transactions will be discarded and re-executed in future rounds. Like $P(6,1)$ in Figure 7. Then all the new shard leaders will execute the transactions within the new DAG. It is noted that, some blocks proposed after the Shift block still will be committed, like $P(5,2)$ and $(5,3)$ at round 5.

In the new DAG, like DAG 2 in Figure 7, each shard leader consistently proposes regular blocks for every round. However, to safeguard against censorship attacks, each shard leader will additionally propose a Shift block to signal $K^{'}=6$ blocks have been proposed and notify the need for rotation, like the Shift blocks at round 7 in the new DAG. Upon the commitment of $2f+1$ Shift blocks, such as by $P(8,4)$, the existing shard leaders will be rotated, and

Time	Transactions	Operations	Dependencies	Commit Order
0	Initial DB	$A = 0, B = 1, C = 0$ $D = 3, F = 5$	{}	{}
1	T_3:(W, D, 3)	T_3 writes $D = 3$	$\{T_3\}$	{}
2	T_5:(R, D, 3)	T_5 reads D on T_3: $(D = 3)$	$\{T_3 \rightarrow T_5\}$	{}
3	T_6:(R, D, 3)	T_6 reads D on T_3: $(D = 3)$	$\{ T_3 \rightarrow T_5$ $T_3 \rightarrow T_6 \}$	{}
4	T_6: Commit	Wait for T_3	$\{ T_3 \rightarrow T_5$ $T_3 \rightarrow T_6 \}$	{}
5	T_3:(W, D, 5)	T_3 writes $D = 5$. Abort T_5, T_6	$\{T_3\}$	{}
6	T_6:(R, D, 5) (Re-execute)	T_6 reads D on T_3: $(D = 5)$	$\{T_3 \rightarrow T_6\}$	{}
7	T_3: Commit	Commit T_3	$\{T_3 \rightarrow T_6\}$	$\{T_3\}$
8	T_6: Commit	Commit T_6	$\{T_3 \rightarrow T_6\}$	$\{T_3, T_6\}$
9	T_5: (W, D, 3)	Invalid and re-execute		
10	T_5:(R, D, 5) (Re-execute)	T_5 reads D on T_3: $(D = 5)$	$\{T_5\}$	$\{T_3, T_6\}$
11	T_5: (W, D, 2)	T_5 writes $D = 2$	$\{ T_3 \rightarrow T_5$ $T_3 \rightarrow T_6 \}$	$\{T_3, T_6\}$
12	T_5: Commit	Commit T_5	$\{ T_3 \rightarrow T_5$ $T_3 \rightarrow T_6 \}$	$\{T_3, T_6, T_5\}$

Table 1: An example of executing transactions generating a dependency graph of $\{T_3, T_5, T_6\}$ in Figure 9.

the new shard leaders will take on the responsibility of proposing new blocks in the future DAG.

Reconfiguration after K' rounds serves to prevent Thunderbolt from frequently shifting the DAG or K rounds if a malicious leader is detected and provides an opportunity for transaction execution.

5 CONCURRENT EXECUTOR

The concurrent executor (*CE*) is a crucial component that enables Thunderbolt to process transactions concurrently. As a standalone component within Thunderbolt, *CE* offers a sequential order, read-/write sets, and execution results to the DAG consensus protocol (section 3.2). The outputs of *CE* can also be verified by each replica in Thunderbolt. The sequential order can be an arbitrary order different from the arrival order of the transactions.

The architecture of *CE* is illustrated in Figure 8, where a group of executors executes transactions, and Concurrency Controller (*CC*) oversees the data execution process. Within *CE*, transactions undergo a two-phase data flow process, which involves an execution phase and a commit phase.

During the execution phase, the executors access the data within *CC* directly. *CC* maintains a dependency graph to keep track of the relationship between transactions. It should be noted that *CC* lacks information about the whole read/write sets that transactions will access and can only maintain the graph based on what the transactions have accessed. If two transactions T_1 and T_2, with T_1 arriving before T_2, having accessed the same keys, a dependency edge from T_1 to T_2 is created, with T_2 depending on T_1.

During the commit phase, the executor informs *CC* that all the operations have been completed. However, a transaction can only be committed if all its dependencies have been successfully committed.

Once the committed transactions are allowed to be stored into the storage, the execution orders are assigned. Then the outcome is applied to the storage asynchronously. If *CC* terminates the execution due to conflicts with other executors, *CC* notifies the executor to restart the process. Table 1 provides an example of executing transactions $\{T_3, T_5, T_8\}$ and generates a dependency graph in Figure 9 to illustrate these two phases in the following sections. The implementation of *CE* can be found in section A.

5.1 Execute Phase

During the execution phase, *CC* will verify each operation sent by the executors on its key, denoted as O_k, by checking the relationships among the transactions. If the transaction T conflicts with other transactions or has been aborted by other transactions, the operation O_k will not be considered valid. For instance, T_5 at time 9 in Table 1 is an example of an invalid operation, as it was aborted by T_3 at time 5 due to its outdated read on D. In such cases, the transaction T will be aborted and require re-execution.

However, if the operation O_k is valid, it will be added to the dependency graph (section 6.1) and obtain the operation result, such as the value V that O_k intends to read. Operation results will be obtained from other transactions directly based on the dependency graph to allow for reading uncommitted data, like T_5 reads D on T_3 at time 2.

5.2 Finalization Phase

Once transaction T is completed without any conflict, the executor requests *CC* to persist all of its operations to the storage. However, if T has any dependent transactions, *CC* will defer the writing to the storage until all of the dependent transactions have either been committed or aborted. It is imperative to note that even after receiving the persistent request from the executor, T could still be aborted due to the abortions from its dependent transactions. If T remains valid after this waiting period, it will be persisted. For example, T_6 at time 4 needs to wait until T_3 persists but is aborted at time 5.

Once T is persisted, the commit order of T will be finalized. Additionally, the commit order of the transactions is the sequential order generated from the dependency graph.

6 PREPLAY CONCURRENT EXECUTION

This section describes the dependency graph G at the heart of the *CC* component, which plays a crucial role in maintaining the causal relationship between transactions during the replay in *CE*. The execution results are also stored in the graph, and *CC* ensures that the sequential order of the execution generated by G is a 'valid' order.

6.1 Dependency Graph Construction

A Dependency Graph (*DG*) is a graph $G(V, E)$ that plays a crucial role in tracking the causal relationship between transactions in *CC*. Each node $v \in V$ represents a specific transaction. Additionally, each edge $e(u, v, k) \in E$ indicates a connection between two transactions u and v on a key K. This relationship is represented as $u \longrightarrow_k v$. For example, in Figure 9, transaction T_5 generates an edge $e(T_3, T_5, D)$ from T_3 because T_5 acquires the value 3 of key D from T_3.

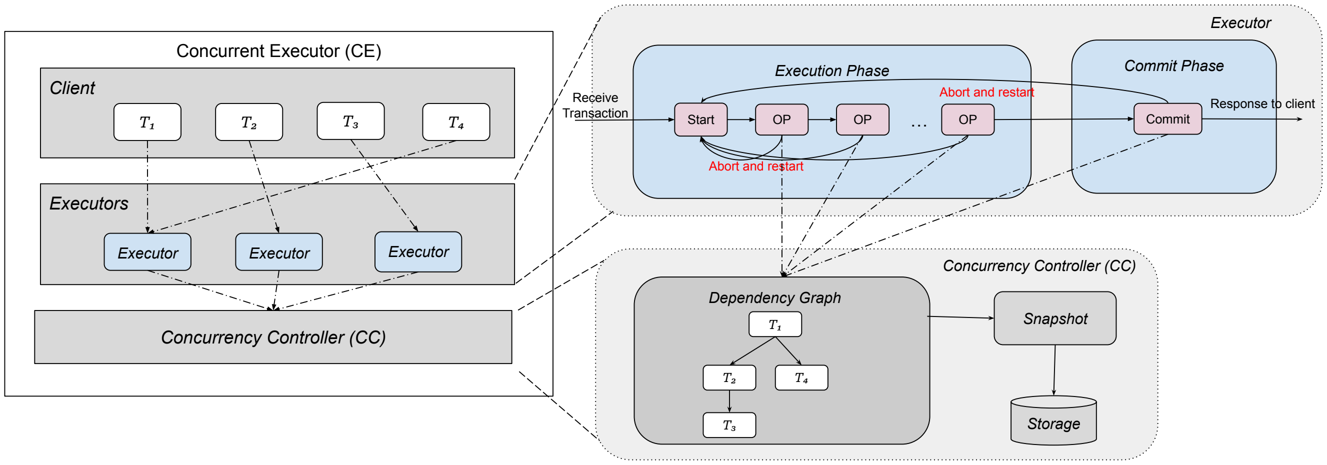


Figure 8: Execution Engine of Thunderbolt. Each executor executes a transaction in prepare and commit phases and the concurrency controller uses a dependency graph to determine the execution results.

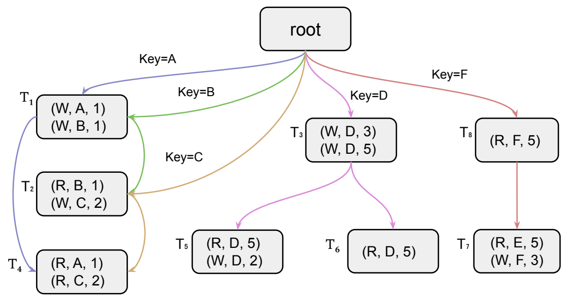
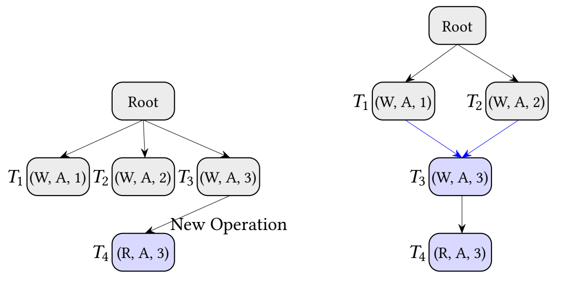


Figure 9: Dependency Graph on *Thunderbolt*. Edges with the same color represent a dependency graph with a specific key.



(a)

Add a new node for T_4 and read the results from T_3.

(b)

Remove the edge of T_3 from root and add edges from T_1 and T_3.

Figure 10: Example to add a new read operation.

Without loss of generality, we have assigned a root node denoted as R and added edges $e(R, u, k) \in E$ for each $u \in V$ that accesses key K but does not have any incoming edge on key K. For example, T_7 and T_8 in Figure 9 do not have any incoming edge on key F since they read key F from the database, the edges $e(R, T_7, F)$ and $e(R, T_8, F)$ are added.

If the graph G is acyclic, then a sequential order can be established by generating a topological order from it. As outlined in section 3.1, it is crucial that every transaction obtains the same causal order in any topological order from G to ensure consistency. Therefore, G is considered a valid graph only if any sequential order generated from the topological order is a valid serialization order and produces the same outcomes. By following any correct order, all transactions will yield the same execution results.

6.2 Graph Node Types and Records

Each node u maintains all the records of the operations triggered by transaction u, including the resulting value of each operation. The collection of operations in each node represents the value state from applying the individual operations. It is possible to simplify the in-node states by combining or removing internal operations. However, to trace the conflicts between two nodes, we must retain the first operation if it is a read and the last operation if it is a write, to ensure that the causal relationship is not lost. Thus, each node contains at most two operations. For example, suppose a transaction T_2 writes a new value on X after reading the old value on T_1. If the read operation of T_2 is removed by a later write, it becomes unclear whether T_2 should be aborted when T_1 is aborted because T_1 only aborts the transactions that have read the values on it.

To help illustrate the algorithm, we define the types of each node which contains at most two operations on a key:

- A node $v \in V$ is a read node R_v^k if the first operation on key K is a read.
- A node $v \in V$ is a write node W_v^k if v contains write operations on key K.
- The root node R is a write node.

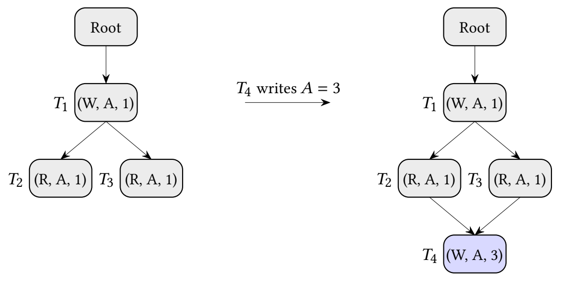


Figure 11: Example to add a new write operation that adds dependencies from all the read nodes.

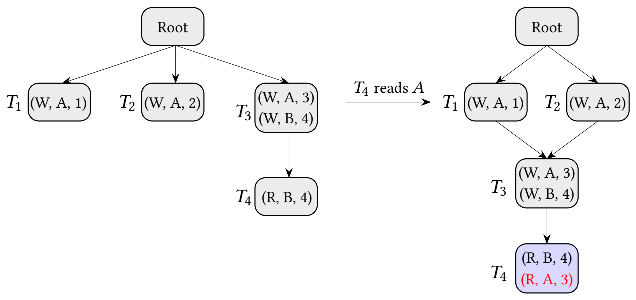


Figure 12: T_4 reads key A on its existing node and obtains the value from T_3. Then modify the graph to guarantee the graph is valid.

6.3 Generating New Nodes

This section presents the process of adding operations from a new transaction to the dependency graph G. The implementation also is shown in section A.2.

Whenever a operation O_k is received from a new transaction T, *CC* will create a new node if T is a new transaction.

If O_k is a write operation, T needs to establish a connection to each casual relation. To avoid pointing to the root and assuming that the earlier transaction will commit first, the non-write nodes v on key K, which only contains reads, without any outgoing edges (not dependent by other nodes) are selected, and edges $e(v, u, k)$ are added, pointing to u (Figure 11).

On the other hand, when a read operation is performed, it is imperative to select the latest write node u, in order to obtain the latest value. In the event that no write nodes exist, the root should be selected to read the data value from storage. Once the write node u has been selected, it is crucial to ensure that all other write nodes also contain a path to u to guarantee the correctness of read after write between u and v.

Finally, the operation and its result <Type, Key, Result> will be written into the node u.

The following is an example of adding a new read operation on A from T_4, as depicted in Figure 10. In part (a), T_4 selects T_3 to read, resulting in obtaining $A = 3$. Thereafter, a record <R, A, 3> is logged down in the node. To ensure the correctness of the read and write, after the modification, it is imperative to reassign the edges. This is achieved by removing the edge between the root R and T_2, and adding two edges $e(T_1, T_3, A)$ and $e(T_2, T_3, A)$ from T_1 and T_2, respectively, as illustrated in part (b) of Figure 10.

6.4 Operations on Existing Nodes

When receiving an operation O_k for key K from an existing transaction T, *CC* will select the corresponding node u to append the record. If O_k constitutes a read and T contains the record for key K, the result will be directly retrieved. If T does not contain any record for key K, it will proceed with the new node operation as specified in section 6.3 to choose a previous one to access the value. Figure 12 illustrates an instance where T_4 reads key A as its second operation and retrieves the value from T_3. If O_k is a write operation, the operation will be appended to the node.

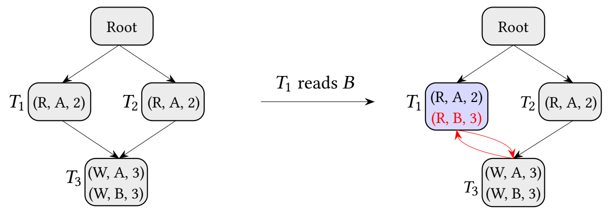

Root
T_1 (R, A, 2)
T_2 (R, A, 2)
T_3 (W, A, 3) (W, B, 3)
T_1 reads B
Root
T_1 (R, A, 2) (R, B, 3)
T_2 (R, A, 2)
T_3 (W, A, 3) (W, B, 3)

Figure 13: T_1 reads B and adds a dependency from T_3 following the rules in §6.3, which results in a cycle of conflict.

This process will also establish dependencies on the existing nodes discussed in section 6.3 to ensure the graph's validity and to simplify the modification, like modifying the previous nodes in Figure 12. Otherwise, we need to find out all the nodes in the graph that do not have a path on key K to u and add dependency edges.

6.5 Conflict Detection

Since we follow the rules outlined in section 6.3 for simplifying the graph modifications, an operation will always identify the most recent nodes and establish a dependency if they access the same keys. However, this approach may lead to the creation of a dependency cycle. Figure 13 depicts a scenario in which T_1 attempts to retrieve the most recent value of B from T_3, which has established a dependency from T_1 due to key A, thereby resulting in the creation of a dependency cycle.

Once conflicts are detected, *CC* triggers an abort process:

(1) If T only contains read operations, abort T itself.
(2) If T contains write operations, cascading abort from T.

In Figure 14, we need to abort T_2 and T_3 since T_1 contains a write operation. However, in Figure 13, we only need to remove T_1 and keep T_3 alive.

6.6 Asynchronous Commit

Upon committing a transaction T, it is important to note that any uncommitted transactions with dependent edges to T may lead to a delay on T. Consequently, the ability of T to successfully commit is contingent upon the absence of such dependencies. Once T has

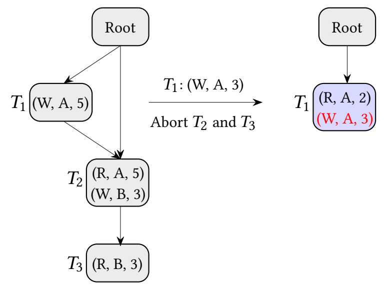


Figure 14: Cascading aborts from T_1 when T_1 wants to write A that conflicts with T_2.

been successfully committed, the resulting data will be written to data storage.

Each node in G stores the outcomes of all the operations from a transaction, enabling quick and easy access to value with a specific key during a read operation. We define the incoming node as follows:

Definition 4. An incoming node of a read node u on key K is defined as the node v, which contains records on the given key K and has a dependency path to u. If node v is both an incoming node and a read node on key K of u, it is an incoming read node on key K of u. Similarly, if node v is both an incoming node and a write node on key K of u, it is an incoming write node on key K of u.

Definition 5. An direct incoming node of a read node u on key K is defined as the node v, which is an incoming node of u on key K and there is no other incoming node w on the dependency path from v to u.

As a result, all operations can be executed in memory, in which each read operation can obtain the result from its direct incoming node, resulting in a streamlined process that can be committed asynchronously without incurring high latency costs from the storage. G also guarantees that each node only contains one direct incoming node on each key.

As the commit order of each transaction is also a valid order generated from G, they can be added to a commit queue. This can help save updates to the storage in a slow path and release more resources to each executor.

7 EVALUATION

This section presents an evaluation of Thunderbolt by measuring the performance on *CE* and the Thunderbolt framework. We implement Thunderbolt on Apache ResilientDB (Incubating) [1, 28]. Firstly, we will compare *CE* to two baseline protocols: OCC, and 2PL-No-Wait. Additionally, we will study the performance of Thunderbolt against the sequential execution built on Tusk [15]. To evaluate the performance, we will use SmallBank [3], a benchmarking suite that simulates common asset transfer transactions.

7.1 Baseline Protocols

We implement OCC and 2PL-No-Wait to compare the performance against our concurrent executor.

7.1.1 OCC. Each executor is responsible for locally executing transactions. When an operation within a transaction T requires reading the value of a key K that the executor has not previously accessed during the execution, the executor will retrieve the value from the storage. Each value also contains a version to indicate the time the value was obtained. For any write operation, it will update the values locally. Upon completion of T, all the updated values will be forwarded to a central verifier. The verifier will cross-check the value versions by comparing them with the current versions in the storage. If there is a mismatch, the commit will be rejected, necessitating the re-execution of T.

7.1.2 2PL-No-Wait. Each executor performs transactions by directly accessing the storage via a central controller. When an operation within a transaction T requires the read or update of a key K, the controller will lock K to prevent conflicts. If an operation seeks to access K but discovers it to be locked by another executor, the executor will release all locks and re-execute T. Upon the completion of T, all the results will be transmitted to storage, and all locks will be released.

7.2 System Setup

We set up our experiments on AWS c5.9xlarge consisting of 36 vCPU, 72GB of DDR3 memory. We use LevelDB as the storage to save the balance of each account.

7.3 Experiment Setup With Smallbank

SmallBank [3] is a transactional system that comprises six distinct transaction types, five of which are designed to update account balances, while the remaining transaction is a read-only query that retrieves both checking and saving the account details of a user. Our focus is on two transaction types: SendPayment and GetBalance, which are used to transfer funds between two checking accounts and retrieve account balances, respectively. Our objective is to evaluate the performance under varying read-write balance workloads. During a SendPayment transaction, account balances are updated by reading the current balance and then writing the new values back. We have created 10,000 accounts and conducted each experiment 50 times to obtain the average outputs.

We evaluated the impact of parallel execution. We measured the performance by uniformly selecting GetBalance with a probability of P_r and SendPayment with $1 - P_r$. To select accounts as transaction parameters, we followed a Zipfian distribution and set the Zipfian parameter θ. The value of θ determines the level of account contention, with higher values leading to greater contention. We only focus on the data workloads with high contention by setting $\theta = 0.85$.

7.4 Impact from Concurrent Executor

We first evaluate the impact of increasing the number of executors to execute the transactions then measure the aborts produced by each protocol. We ran three batch sizes $b100$, $b300$, and $b500$ for each protocol: Thunderbolt-b100, Thunderbolt-b300, Thunderbolt-b500, OCC-b100, OCC-b300, OCC-b500, 2PL-No-Wait-b100, 2PL-No-Wait-b300, and 2PL-No-Wait-b500.

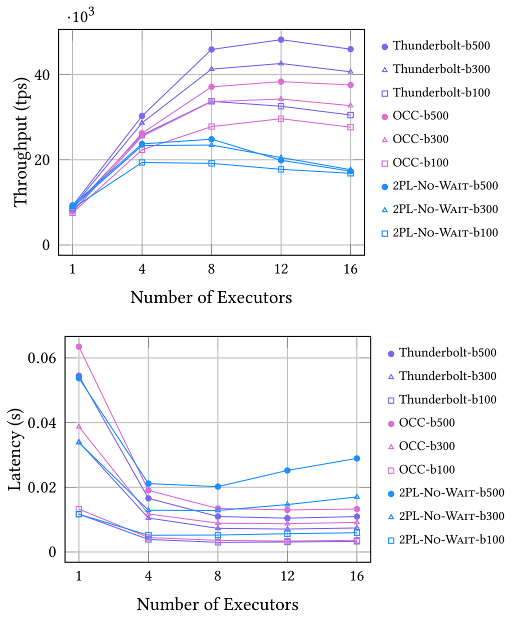


Figure 15: Throughput and latency of different numbers of executors in SmallBank with $\theta = 0.85$ and $P_r = 0.5$.

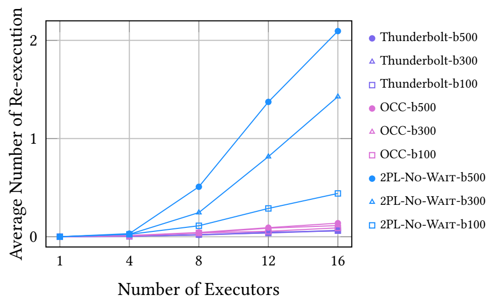


Figure 16: Average retry time of different numbers of executors in SmallBank with $\theta = 0.85$ and $P_r = 0.5$.

7.4.1 Number of Executors. We conducted experiments to compare the performance of Thunderbolt with OCC [36] and 2PL-No-Wait [51]. We set $P_r = 0.5$ to measure a read-write balanced workflow and $P_r = 0$ on an update-only workflow.

In the read-write balanced workflow, the results depicted in Figure 15 show that Thunderbolt with batch size 100 (Thunderbolt-b100) and 2PL-No-Wait protocols with different batch sizes all experience a drop in performance when increasing the number of executors beyond 8. However, Thunderbolt-b300, Thunderbolt-b500, and OCC protocols with all the batch sizes obtain their highest throughput on 12 executors and maintain stable throughput.

In the update-only workflow, the results shown in Figure 17 indicate that OCC and 2PL-No-Wait stopped increasing earlier at 4 executors while Thunderbolt provides a peek throughput on 12 executors.

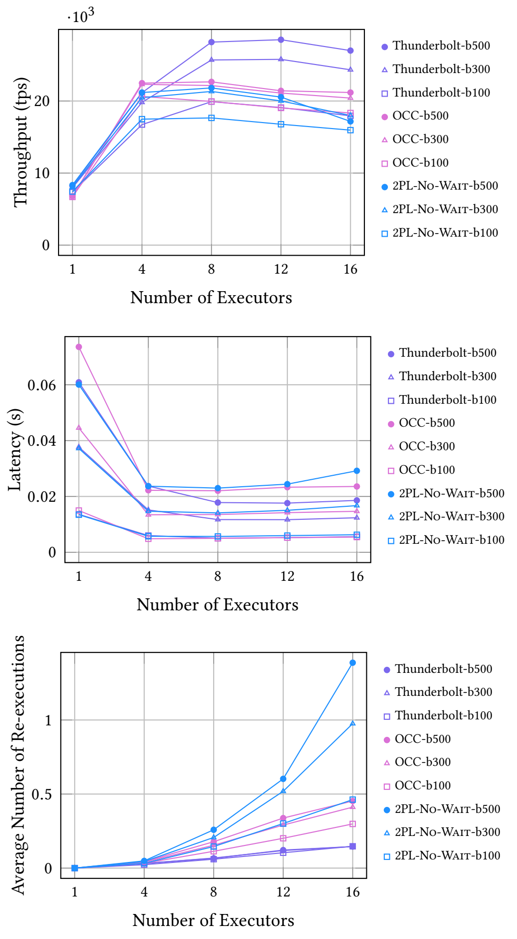


Figure 17: Evaluation of different numbers of executors in SmallBank with $\theta = 0.85$ and $P_r = 0$.

These experiments demonstrate that all the protocols do not obtain significant benefits for a large number of executors in a high-competition workflow. However, Thunderbolt still can achieve more parallelism with more executors.

7.4.2 Evaluation of Abort Rates. As we increase the number of executors, we have also been measuring the average number of re-executions of transactions. The results in Figure 16 and Figure 17 indicate that when the number of executors goes beyond 8, all 2PL-No-Wait protocols experience a significant increase in the rate of abortions, leading to a drop in throughput from $24k$ to $18k$ in the read-write balanced workflow, and from $22k$ to $17k$ in the update-only workflow for 2PL-No-Wait-b500. While OCC protocols provide a lower rate within the read-wirte balanced workflow, the rates still increase in the update-only experiments, resulting in a decrease in throughput from $22k$ to $21k$ when setting the batch size as 500 (OCC-b500). However, Thunderbolt achieves the lowest abortions, with Thunderbolt-b500 reducing 50% of the abortions from OCC-b500 and 90% from 2PL-No-Wait-b500 in all the experiments.

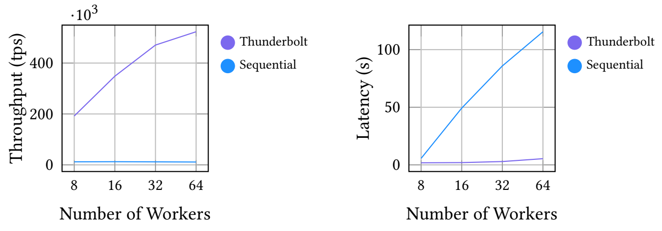


Figure 18: Throughput and average latency within SmallBank.

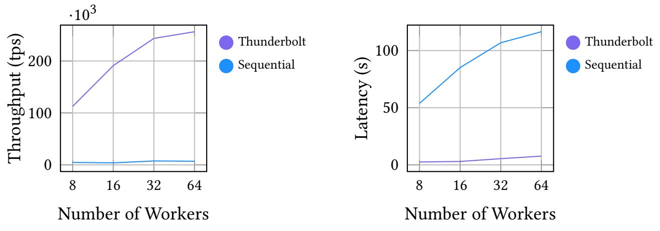


Figure 19: Throughput and average latency within payment in TPC-C.

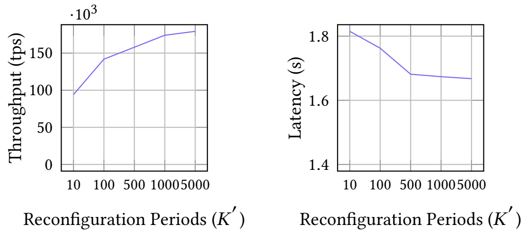


Figure 20: Throughput and average latency within different reconfiguration periods in SmallBank.

7.5 System Evaluation

In this section, we conduct evaluations to determine the impact of Thunderbolt on a Tusk [15]. We implemented Thunderbolt using Apache ResilientDB (Incubating) [1, 28]. The performance tests were conducted on AWS EC2 with c5.9xlarge instances.

In our evaluation, we compared the performance of Thunderbolt with sequential execution, which executes transactions after reaching a total order after DAG protocols. We used two data workloads, SmallBank [3] and TPC-C [2].

For each replica, we set up an *CE* with 16 executors to execute the transactions with a batch of 500 and 16 validators to validate the block after consensus. We scaled the system from 8 replicas to 64. By default, we have set K' to a large value to prevent rotation. At the end of our evaluation, we will assess the effects of various K' (§7.5.3).

7.5.1 SmallBank. Within the experiments with Smallbank workload, we only focus on the read-write balanced scenario ($P_r = 0.5$) that half of the transactions are read-only. The addresses in the transactions are selected in 1000 users with $\theta = 0.85$ to simulate a high contention workload. We compare the results among Thunderbolt and sequential execution. The sequential execution executes the transactions in the total order given by the consensus protocol. The results in Figure 18 show that Thunderbolt achieved higher throughput than the sequential order which speeds up 50X that Thunderbolt obtained $500K$ TPS while sequential execution only obtained $11K$ TPS. The results also demonstrate while increasing the replicas, the throughput increased when moving the executions ahead.

7.5.2 TPC-C. The TPC-C [2] benchmark is a widely recognized industry standard that simulates a commerce system with five transaction types involving customers, orders, warehouses, districts, stock, and items. Data in TPC-C is represented in nine normalized tables. Since we aim to test the performance within a high data contention environment, we focused only on the payment transaction, which updates the customer's balance and reflects the payment on the district and warehouse sales table.

To conduct our test, we assigned one warehouse for each group within 100 districts and randomly located 1000 customers across these districts. We utilized a value of $\theta = 0.85$ to select customers and subsequently updated their balances, as well as the related warehouse and district table.

Figure 19 demonstrates that Thunderbolt outperformed sequential execution. Thunderbolt achieved 36 times speed-up again sequential execution that Thunderbolt obtains $256K$ TPS while sequential execution only obtains $7k$ TPS with 64 replicas. Thunderbolt also obtained 0.06X less latency from sequential execution.

7.5.3 Reconfiguration Periods. Now, we analyzed the performance using different reconfiguration periods K' to transition the shard leaders into a new DAG on 8 replicas. Figure 20 demonstrates that Thunderbolt exhibited lower performance with smaller K' values ($80K$ TPS with $K' = 10$), attributed to the costly transition between DAGs. Conversely, when K' was increased to over 1000, Thunderbolt demonstrated significantly improved stability, achieving a throughput of $180K$ TPS. Additionally, the average latency decreased as K' increased from $1.9s$ to $1.7s$.

8 RELATED WORK

We discuss other prior works relevant to Thunderbolt and the comparison.

8.1 Sharding

Numerous studies [4, 16, 29, 31, 35, 43, 59, 66] has been carried out on the necessity of sharding to improve the scalability in blockchain systems.

Omiledger [35], Dang [16], and RepidChain [66] propose the provision of shards by generating committees with random coins. However, these existing protocols rely on a random assignment based on a distributed random coin generation on the committees and re-assign the committee members in each epoch.

In contrast, Thunderbolt takes a different approach by making each replica as a shard leader and leveraging the DAG to live migration to a new DAG to rotate the leader if the replicas detect malicious attacks. The assignment and reconfiguration in Thunderbolt are much more efficient than previous work.

8.2 Execute-Order-Validate

The Execute-Order-Validate (EOX) framework was originally introduced by Hyperledger Fabric [6] offering the advantage of allowing transactions to be optimistically executed by a subset of executors before obtaining a global order. However, the parallel execution may lead to conflicts between two transactions, causing the validation to abort the transactions after being ordered.

Various techniques have been proposed to improve the bottleneck on Hyperledger, including optimizing the peer process, replacing the state DB with a hash table, and pipelining the execution [24, 57, 58] Other platforms like Fabric++ [46], XOXFabric [23], and FabricSharp [44] have also been developed to reorder transactions within a block by analyzing their dependencies after the consensus, which reduces the abort rate.

Thunderbolt, inspired by Hyperledger, has implemented an execution model that allows transactions to be executed before ordering. However, different from Hyperledger, Thunderbolt distributes transactions into different shards and transactions in each shard will be executed by a shard leader. This approach enables each shard leader to leverage the concurrent executor to execute transactions in parallel while also reordering them to improve performance and reduce the abort rates.

8.3 Concurrent Execution

Deterministic approaches have been proposed [19, 41, 64] to make the execution of transactions more efficient. These methods involve constructing a dependency graph to allow transactions to be executed concurrently without causing conflicts. Transaction Chopping [47, 48], SChain [13], and Caracal [42] go one more stop by dividing transactions into smaller pieces before building the dependency graph. Non-volatile main Memory is also introduced in [62] to address long latency transactions.

However, these techniques have some limitations, such as the need to obtain transaction execution read/write sets in advance.

In contrast, Thunderbolt does not rely on the assumption of read/writes, assigns the order dynamically, and minimizes conflicts between transactions.

CHIRON [40] utilizes BlockSTM [21], which provides a non-deterministic execution to extract dependencies (hints) from smart contracts on Ethereum for the acceleration of straggling and full nodes. Block-STM allows smart contracts to be executed in parallel in some order and outputs the read/write sets as results. However, the execution order is based on the arrival time of the transactions.

In contrast, Thunderbolt does not rely on the assumption of arrival time, assigns the order dynamically, and minimizes conflicts between transactions.

8.4 Transaction Reordering

BCC [65] proposed a method to minimize the number of aborted transactions by checking the commit time to adjust the commit order. A directed graph within a batch is constructed in [17] and a greedy algorithm is introduced to reorder transactions to address the NP-hard problem for highly contentious cases. Similar techniques have been applied in [23, 44, 46]. However, these approaches have a higher latency as they aim to find the best global order on a deterministic graph built from the read/write sets obtained from the execution.

In contrast, Thunderbolt builds the graph dynamically and maintains it online. This ensures that the graph is always up-to-date and the order of transactions can be adjusted as needed to achieve the desired results.

8.5 Concurrent Consensus

Blockchain fabric with high performance and scalability is crucial [5, 25, 27, 45]. PoE [26] reduced one phase from Pbft [12] by introducing a speculative execution. RCC [28], FlexiTrust [30], and SpotLess [32] extend the single-leader protocols to multi-leaders to improve parallelism. However, these protocols do not support reconfiguration.

9 CONCLUSIONS

We have developed Thunderbolt, a sharding system that is built on a DAG-based protocol to enhance the serial execution of smart contracts. Thunderbolt shards transactions into distinct shards and transactions in each shard will be executed by a shard leader, to avoid contention between shards and natively prevent malicious clients from degrading the goodput of the system by submitting the same transaction to all replicas. Thunderbolt leverages the properties of DAG to migrate the current DAG to a new DAG without a hard stop to rotate the leaders of each shard if a malicious shard leader is detected. We also implement a concurrent executor engine to enhance the execution of smart contracts. by generating a dependency graph dynamically without any read/write sets known prior.

Our performance evaluation results have demonstrated that Thunderbolt can deliver a 50 times speed-up compared to the native execution running on a popular DAG-based consensus system.

10 ACKNOWLEDGEMENTS

This work is partially funded by NSF Award Number 2245373.

REFERENCES

[1] [n. d.]. Apache ResilientDB (Incubating). https://resilientdb.incubator.apache.org/

[2] [n. d.]. TPCC. https://www.tpc.org/tpcc/

[3] 2019. smallbank benchmark. http://hstore.cs.brown.edu/documentation/deployment/benchmarks/smallbank/

[4] 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).

[5] Mohammad Javad Amiri, Chenyuan Wu, Divyakant Agrawal, Amr El Abbadi, Boon Thau Loo, and Mohammad Sadoghi. 2022. The Bedrock of BFT: A Unified Platform for BFT Protocol Design and Implementation. *CoRR* abs/2205.04534 (2022). https://doi.org/10.48550/arXiv.2205.04534 arXiv:2205.04534

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

[7] Balaji Arun, Zekun Li, Florian Suri-Payer, Sourav Das, and Alexander Spiegelman. 2024. Shoal++: High Throughput DAG BFT Can Be Fast! *arXiv preprint arXiv:2405.20488* (2024).

[8] 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).

[9] 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).

[10] 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).

[11] Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. 2011. *Introduction to reliable and secure distributed programming.* Springer Science & Business Media.

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

[13] Zhihao Chen, Haizhen Zhuo, Quanqing Xu, Xiaodong Qi, Chengyu Zhu, Zhao Zhang, Cheqing Jin, Aoying Zhou, Ying Yan, and Hui Zhang. 2021. SChain: a scalable consortium blockchain exploiting intra-and inter-block concurrency. *Proceedings of the VLDB Endowment* 14, 12 (2021), 2799–2802.

[14] Mario Ciampi, Fabrizio Marangio, Giovanni Schmid, and Mario Sicuranza. 2021. A Blockchain-based Smart Contract System Architecture for Dependable Health Processes. In *Italian Conference on Cybersecurity*. https://api.semanticscholar.org/CorpusID:244895448

[15] 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*. Association for Computing Machinery, New York, NY, USA, 34–50. https://doi.org/10.1145/3492321.3519594

[16] Hung Dang, Tien Tuan Anh Dinh, Dumitrel Loghin, Ee-Chien Chang, Qian Lin, and Beng Chin Ooi. 2019. Towards scaling blockchain systems via sharding. In *Proceedings of the 2019 international conference on management of data*. 123–140.

[17] Bailu Ding, Lucja Kot, and Johannes Gehrke. 2018. Improving optimistic concurrency control through transaction batching and operation reordering. *Proceedings of the VLDB Endowment* 12, 2 (2018), 169–182.

[18] 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.

[19] Jose M Faleiro, Daniel J Abadi, and Joseph M Hellerstein. 2017. High performance transactions via early write visibility. *Proceedings of the VLDB Endowment* 10, 5 (2017).

[20] Zaleha Fauziah, Haznah Latifah, Xavier Omar, Alfiah Khoirunisa, and Shofiyul Millah. 2020. Application of Blockchain Technology in Smart Contracts: A Systematic Literature Review. *Aptisi Transactions on Technopreneurship (ATT)* (2020). https://api.semanticscholar.org/CorpusID:225417313

[21] 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. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 232–244.

[22] Neil Giridharan, Florian Suri-Payer, Ittai Abraham, Lorenzo Alvisi, and Natacha Crooks. 2024. Motorway: Seamless high speed BFT. *arXiv preprint arXiv:2401.10369* (2024).

[23] Christian Gorenflo, Lukasz Golab, and Srinivasan Keshav. 2020. XOX Fabric: A hybrid approach to blockchain transaction execution. In *2020 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. IEEE, 1–9.

[24] Christian Gorenflo, Stephen Lee, Lukasz Golab, and Srinivasan Keshav. 2020. FastFabric: Scaling hyperledger fabric to 20 000 transactions per second. *International Journal of Network Management* 30, 5 (2020), e2099.

[25] Suyash Gupta, Mohammad Javad Amiri, and Mohammad Sadoghi. 2023. Chemistry behind Agreement. In *13th Conference on Innovative Data Systems Research, CIDR 2023, Amsterdam, The Netherlands, January 8-11, 2023*. www.cidrdb.org.

[26] Suyash Gupta, Jelle Hellings, Sajjad Rahnama, and Mohammad Sadoghi. 2021. Proof-of-Execution: Reaching Consensus through Fault-Tolerant Speculation. In *Proceedings of the 24th International Conference on Extending Database Technology*.

[27] Suyash Gupta, Jelle Hellings, and Mohammad Sadoghi. 2021. *Fault-Tolerant Distributed Transactions on Blockchain.* Morgan & Claypool Publishers. (2021).

[28] Suyash Gupta, Jelle Hellings, and Mohammad Sadoghi. 2021. RCC: Resilient Concurrent Consensus for High-Throughput Secure Transaction Processing. In *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021.* IEEE, 1392–1403. https://doi.org/10.1109/ICDE51399.2021.00124

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

[30] Suyash Gupta, Sajjad Rahnama, Shubham Pandey, Natacha Crooks, and Mohammad Sadoghi. 2023. Dissecting BFT Consensus: In Trusted Components we Trust!. In *Proceedings of the Eighteenth European Conference on Computer Systems, EuroSys 2023, Rome, Italy, May 8-12, 2023*, Giuseppe Antonio Di Luna, Leonardo Querzoni, Alexandra Fedorova, and Dushyanth Narayanan (Eds.). ACM, 521–539.

[31] Jelle Hellings and Mohammad Sadoghi. 2023. ByShard: sharding in a Byzantine environment. *VLDB J.* 32, 6 (2023), 1343–1367.

[32] Dakai Kang, Sajjad Rahnama, Jelle Hellings, and Mohammad Sadoghi. 2024. SpotLess: Concurrent Rotational Consensus Made Practical through Rapid View Synchronization. In *40th IEEE International Conference on Data Engineering, ICDE 2024, Utrecht, Netherlands, May 13-17, 2024.* IEEE.

[33] Idit Keidar, Eleftherios Kokoris-Kogias, Oded Naor, and Alexander Spiegelman. 2021. All you need is dag. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*. 165–175.

[34] Idit Keidar, Oded Naor, Ouri Poupko, and Ehud Shapiro. 2022. Cordial miners: Fast and efficient consensus for every eventuality. *arXiv preprint arXiv:2205.09174* (2022).

[35] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. 2018. Omniledger: A secure, scale-out, decentralized ledger via sharding. In *2018 IEEE symposium on security and privacy (SP)*. IEEE, 583–598.

[36] Hsiang-Tsung Kung and John T Robinson. 1981. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)* 6, 2 (1981), 213–226.

[37] Leslie Lamport. 2019. Time, clocks, and the ordering of events in a distributed system. In *Concurrency: the Works of Leslie Lamport*. 179–196.

[38] Dahlia Malkhi, Chrysoula Stathakopoulou, and Maofan Yin. 2023. BBCA-CHAIN: One-Message, Low Latency BFT Consensus on a DAG. *arXiv preprint arXiv:2310.06335* (2023).

[39] Microsoft. [n. d.]. eEVM. https://github.com/microsoft/eEVM

[40] Ray Neiheiser, Arman Babaei, Giannis Alexopoulos, Marios Kogias, and Eleftherios Kokoris Kogias. 2024. CHIRON: Accelerating Node Synchronization without Security Trade-offs in Distributed Ledgers. *arXiv preprint arXiv:2401.14278* (2024).

[41] Thamir M Qadah and Mohammad Sadoghi. 2018. Quecc: A queue-oriented, control-free concurrency architecture. In *Proceedings of the 19th International Middleware Conference*. 13–25.

[42] Dai Qin, Angela Demke Brown, and Ashvin Goel. 2021. Caracal: Contention management with deterministic concurrency control. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 180–194.

[43] Sajjad Rahnama, Suyash Gupta, Rohan Sogani, Dhruv Krishnan, and Mohammad Sadoghi. 2022. RingBFT: Resilient Consensus over Sharded Ring Topology. In *Proceedings of the 25th International Conference on Extending Database Technology*. OpenProceedings.org, 2:298–2:311. https://doi.org/10.48786/edbt.2022.17

[44] Pingcheng Ruan, Dumitrel Loghin, Quang-Trung Ta, Meihui Zhang, Gang Chen, and Beng Chin Ooi. 2020. A transactional perspective on execute-order-validate blockchains. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 543–557.

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

[46] Ankur Sharma, Felix Martin Schuhknecht, Divya Agrawal, and Jens Dittrich. 2019. Blurring the lines between blockchains and database systems: the case of hyperledger fabric. In *Proceedings of the 2019 International Conference on Management of Data*. 105–122.

[47] Dennis Shasha, Francois Llirbat, Eric Simon, and Patrick Valduriez. 1995. Transaction chopping: Algorithms and performance studies. *ACM Transactions on Database Systems (TODS)* 20, 3 (1995), 325–363.

[48] Dennis Shasha, Eric Simon, and Patrick Valduriez. 1992. Simple rational guidance for chopping up transactions. In *Proceedings of the 1992 ACM SIGMOD International Conference on management of Data*. 298–307.

[49] Madhavapeddi Shreedhar and George Varghese. 1995. Efficient fair queueing using deficit round robin. In *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*. 231–242.

[50] Nibesh Shrestha, Rohan Shrothrium, Aniket Kate, and Kartik Nayak. 2024. Sailfish: Towards Improving Latency of DAG-based BFT. *Cryptology ePrint Archive* (2024).

[51] Eljas Soisalon-Soininen and Tatu Ylönen. 1995. Partial strictness in two-phase locking. In *International Conference on Database Theory*. Springer, 139–147.

[52] Alexander Spiegelman, Balaji Aurn, Rati Gelashvili, and Zekun Li. 2023. Shoal: Improving dag-bft latency and robustness. *arXiv preprint arXiv:2306.03058* (2023).

[53] Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. 2022. Bullshark: Dag bft protocols made practical. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 2705–2718.

[54] Chrysoula Stathakopoulou, Michael Wei, Maofan Yin, Hongbo Zhang, and Dahlia Malkhi. 2023. BBCA-LEDGER: High Throughput Consensus meets Low Latency. *arXiv preprint arXiv:2306.14757* (2023).

[55] Sui. 2024. Build Beyond. https://sui.io.

[56] Nick Szabo. 1996. Smart contracts: building blocks for digital markets. *EXTROPY: The Journal of Transhumanist Thought,(16)* 18, 2 (1996), 28.

[57] Parth Thakkar and Senthilnathan Natarajan. 2021. Scaling blockchains using pipelined execution and sparse peers. In *Proceedings of the ACM Symposium on Cloud Computing*. 489–502.

[58] Parth Thakkar, Senthil Nathan, and Balaji Viswanathan. 2018. Performance benchmarking and optimizing hyperledger fabric blockchain platform. In *2018 IEEE 26th international symposium on modeling, analysis, and simulation of computer and telecommunication systems (MASCOTS)*. IEEE, 264–276.
[59] Gang Wang, Zhijie Jerry Shi, Mark Nixon, and Song Han. 2019. Sok: Sharding on blockchain. In *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*. 41–61.
[60] Hao Wang, Chaonian Guo, and Shuhan Cheng. 2019. LoC — A new financial loan management system based on smart contracts. *Future Generation Computer Systems* 100 (2019), 648–655. https://doi.org/10.1016/j.future.2019.05.040
[61] Shuai Wang, Chengyu Zhang, and Zhendong Su. 2019. Detecting nondeterministic payment bugs in ethereum smart contracts. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–29.
[62] Yu Chen Wang, Angela Demke Brown, and Ashvin Goel. 2023. Integrating Non-Volatile Main Memory in a Deterministic Database. In *Proceedings of the Eighteenth European Conference on Computer Systems*. 672–686.
[63] Zhaoguo Wang, Shuai Mu, Yang Cui, Han Yi, Haibo Chen, and Jinyang Li. 2016. Scaling multicore databases via constrained parallel execution. In *Proceedings of the 2016 International Conference on Management of Data*. 1643–1658.
[64] Chang Yao, Divyakant Agrawal, Gang Chen, Qian Lin, Beng Chin Ooi, Weng-Fai Wong, and Meihui Zhang. 2016. Exploiting single-threaded model in multi-core in-memory systems. *IEEE Transactions on Knowledge and Data Engineering* 28, 10 (2016), 2635–2650.
[65] Yuan Yuan, Kaibo Wang, Rubao Lee, Xiaoning Ding, Jing Xing, Spyros Blanas, and Xiaodong Zhang. 2016. Bcc: Reducing false aborts in optimistic concurrency control with low cost for in-memory databases. *Proceedings of the VLDB Endowment* 9, 6 (2016), 504–515.
[66] Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova. 2018. Rapidchain: Scaling blockchain via full sharding. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*. 931–948.

Appendix A CONCURRENT EXECUTOR IMPLEMENTATION

We demonstrate the implementation of each component in *CE* in this section.

A.1 Execution on the Executors

Figure 21 shows the pseudocode code of the executors in *CE*. When *CE* receives a batch *B* of transactions, it initiates a set of executors and runs the eEVM [39], a tool to execute smart contracts, to execute the contract code. eEVM provides the read and write callback functions for the developers to implement their implementations to read and write the values for each key. When the read and write functions are triggered, *CE* will send the operation to *CC* (Line 26 and 28). If an abort of a transaction T is received from *CC* due to the conflict with other transactions, the execution will be aborted and T will be re-sent to the executor to re-execute. When eEVM completes the execution, the executor will commit the transaction (Line 15).

As discussed in section 5.2, the transaction is not actually committed after sending the commit request to *CC*. *CC* will send a notification after the transaction is finalized and *CE* can obtain the results (Line 17). Once all the transactions are committed, *CE* returns their scheduled orders (the commit order), read/write sets, and operation results.

A.2 Concurrency Controller

Concurrency Controller (*CC*) will receive four types of operations: Write, Read, Commit. *CC* will check if the transactions have been aborted before processing. The algorithm is shown in Figure 22.

A.2.1 Write OP. When *CC* receives a write operation O_k from a transaction T with key K to update the value to V, If O_k is a new record, *CC* will add the necessary edges to the graph by linking all the non-write nodes (defined in section 6.2) to ensure the graph is valid as depicted in section 6.3. If adding the record failed due to a cycle detected, abort T by removing the node or processing cascading abort (Line 41).

A.2.2 Read OP. Similar to the write OP, if the read operation O_k from T is a new record, *CC* returns the value and adds the necessary edges to the graph by linking all the write nodes. While adding the record, the graph will return the node u' the record refers to (Line 19) which is the node T reads K from. If u' does not exist due to conflicts detected, abort T.

A.2.3 Commit. When receiving a commit request to inform the execution is done for transaction T, *CC* will place T to a pending list (*commit_list*) if there is any dependency for T in the graph (Line 33). Otherwise, commit the nodes and generate the read/write sets RWS and the values of all the reads V. The commit order cid which is the scheduled order for the transactions is obtained. Then *CC* will notify the executors that transaction T has been committed, with the outcomes $<T, cid, RWS, V>$.

Upon a transaction T has been committed, it will check if there is any other transaction waiting in *commit_list* that it is allowed to commit as its dependency has been removed (Lines 37-40).

```
Let cc be the instance of $CC$.
Let $result\_list$ be the execution outcomes for the transactions.
Let $cid$ be the cid assigned by $CC$ if the transaction is read-only.
Initial $cid = -1$.

function Execute (Batch $b$)
    Initial the $result\_list$: $result\_list$.clear()
    Initial a new instance $CC$: cc.init()

    for Each transaction $T$ in Batch $b$ do
        Deliver $T$ to the executor to execute
    Wait for the results for all the transactions in
    Return $result\_list$

event Receive an abort of transaction $T$ from $CC$ do
    Abort $T$ if the executor of $T$ is alive
    Deliver $T$ to the executor to re-execute
    Commit $T$: cc.commit($T$)

event Receive a commit notification with <$T$, $cid$, $RES$, $V$> do
    result.append(<$T$, $cid$, $RES$, $V$>)

Executor :
event Receive a transaction $T$ do
    Notify $CC$ to clean the state of $T$: cc.StartTxn($T$)
    Leverages eEVM to execute $T$

event Read key $K$, transaction $T$ do
    if $T$ is a read-only transaction then
        $V$, $cid$ = cc.readonly($K$, $T$, $cid$)
        Return $V$
    else
        Return cc.read($K$, $T$)

event Write key $K$ with value $V$, transaction $T$ do
    Return cc.write($K$, $V$, $T$)
```

Figure 21: Executor Implementation.

```
Let $G$ be the dependency graph.
Let $commit\_list$ be the list saving the transactions waiting for commit.

function StartTxn (transaction $T$)
    Clean the abort state of $T$ if it is aborted.

function Write (key $K$, value $V_{new}$, transaction $T$)
    if $T$ has been aborted then
        Return Fail
    Let $u$ be the node of transaction $T$.
    if If $u$ is a new node on $K$ then
        if $G$.AddNewWriteRecord($u$, $V_{new}$, $K$) returns Fail then
            AbortNode($u$)
            Return Fail
    Return Success

function Read (key $K$, transaction $T$)
    if $T$ has been aborted then
        Return Fail
    Let $u$ be the node of transaction $T$.
    if $u$ is a new node on $K$ then
        $u^{'}$ = $G$.AddNewReadRecord($u$, $K$)
    else
        $u^{'} = u$:
    if $u^{'}$ does not exist then
        AbortNode(u)
        Return Fail
    $V$ is the lastest updating value key $K$ on $u^{'}$
    Add a new record <$R$, $K$, $V$> to $u$
    Return $V$

function Commit (transactioin $T$)
    if $T$ has been aborted then
        Return Fail
    Let $u$ be the node of transaction $T$.
    if $u$ contains dependency edges in $G$ then
        $commit\_list$.append($T$)
    else
        CommitNode($u$)
        Remove $T$ from $commit\_list$
        for each transaction $T^{'}$ in $commit\_list$ do
            Let $u^{'}$ be the node of transaction $T$.
            if $u^{'}$ does not have dependency in $G$ then
                Commit($T$)

function AbortNode (node $u$, transaction $T$)
    if $u$ is not a write node on every key then
        Remove($u$)
    else
        CasadingAbort($u$)
    Mark $T$ is aborted.
    Send notification to the executors.

function CommitNode (node $u$, transaction $T$)
    $RWS$ is the read/write sets
    $V$ is all the read results
    for Each record $< Type, Key, Value >$ in $u$ do
        $RES$.append($< Type, Key >$)
        if $Type = R$ then
            $V$.append($< Key, Value >$)
    $cid$ = Commit($T$)
    Notify $CE$ with <$T$, $cid$, $RES$, $V$>
```

Figure 22: *CC* Implementation.

A.3 Dependency Graph

Figure 23 shows how the Dependency Graph is constructed in CC. When adding a write record to node u on key K, the graph will ensure all the non-write nodes on key K, which only reads the value on K, have the paths to u (Line 7). After adding the edges, a cycle check will be triggered to detect the conflict (Line 12).

On the other hand, when adding a read record on node u, u tries to obtain the value from an incoming node x (Line 30). If no such node, a write node of key K will be searched or using the root node instead. When adding the edges from node x, some other edges will be added to ensure the graph is valid (Lines 23-25). Finally, returning the referring node x to CC.

Appendix B THUNDERBOLT SECURITY ANALYSIS

We provide the proofs for the security of Thunderbolt we have discussed in section 3.1. Suppose Thunderbolt generates a sequential order $SO = [T_1, \ldots, T_n]$ and produces an outcome $OUT = [OUT_1, \ldots, OUT_n]$ for a set of transactions. Let SE be the sequential execution in SO. Let the outcomes $OUT^{'}$ be the outcomes of SE.

B.1 Causal Ordering

Before giving proofs, we define notions for the causal ordering. Causal ordering of transactions in distributed systems is introduced by Lamport by defining a well-known "happened before" relation, denoted $\longrightarrow$ [37]. If A and B are two events, then $A \longrightarrow B$ if and only if one of the following conditions is true:

(1) A occurs before B in the same location;
(2) A is an outgoing message, and B corresponds to the response message;
(3) There is an existing event C that $A \longrightarrow C$ and $C \longrightarrow B$;

```
Let $root$ is the root node.

function AddNewWriteRecord (node $u$, value $V$, key $K$)
    $link\_to\_root = true$
    for each non-write node $v$ containing records on $K$ do
        if $v$ does not contain any outgoing edge on $K$ then
            /* No other nodes depending on $v$ on $K$ */
            Add edges($v$, $u$, $K$)
            $link\_to\_root = false$
    if $link\_to\_root == true$ then
        Add edges($root$,$u$,$K$) /* Link to root */
    else
        if Contain a cycle on $u$ then
            Return Fail
    Return Sucess

/* Find a write node $x$ on $K$ and depend on $x$ then return the value from $x$ */
function AddNewReadRecord (node $u$, key $K$)
    /* Find a incoming node $x$ on key $K$ */
    $x$ = GetIncomingNode($u$, $K$)
    if $x! = None$ then
        $x$ = GetWriteNodeToRead($u$, $K$)
    if $x! = None$ then
        Add edges($x$,$u$,$K$) /* Link to $x$ */
        for Each write node on $w$ key $K$ do
            if No path from $w$ to $x$ on $K$ then
                Add edges($w$,$x$,$K$)
        Return $x$
    Add edges($root$,$u$,$K$) /* Link to root */
    Return $root$

/* Find a incoming node $x$ on key $K$ */
function GetIncomingNode (node $u$, key $K$)
    if $u$ contains any incoming read node $v$ of $u$ on key $K$ then
        if CheckCycle($v$, $u$, $K$) == Fail then
            Return $v$
    if $u$ contains any incoming write node $v$ of $u$ on key $K$ then
        if CheckCycle($v$, $u$, $K$) == Fail then
            Return $v$
    Return None

/* Find a write node $x$ to read on key $K$ */
function GetWriteNodeToRead (node $u$, key $K$)
    for each write node $v$ containing records on $K$ do
        if CheckCycle($v$, $u$, $K$) == Fail then
            Return $v$
    Return None

/* Check if a read node $v$ can read values on node $u$ on key $K$ */
function CheckCycle (node $u$, node $v$, key $K$)
    if $u$ is not a write node on key $K$ then
        /* read values from a node without any update will not affect the graph */
        Return Fail
    for Each write node on $w$ key $K$ do
        /* need to ensure $u$ is the last update */
        if Contain a path from $v$ to $w$ then
            /* a cycle occurs $u \rightarrow v \rightarrow w \rightarrow u$ */
            Return Fail
    Return Success
```

Figure 23: Dpendency Graph Implementation.

We extend the causal ordering from events to transactions in concurrency control to define a causal relationship between two transactions A and B:

Definition 6. If transaction A needs to be executed before B, $A \rightarrow B$, if and only if B reads the data updated by A.

Definition 7. A and B can be executed concurrently only if $A \nrightarrow B$ and $B \nrightarrow A$.

Definition 8. If A has a causal relationship with B, either $A \rightarrow B$ or $B \rightarrow A$.

Definition 9. If $A \rightarrow B$ and $B \rightarrow C$, then $A \rightarrow C$.

B.2 Proof of Serializability

If Thunderbolt is serializable, $OUT = OUT'$.

Definition 10 (Read-Complete). If T_i reads a value from T_j in the execution in Thunderbolt, T_i will also read the value from T_j in SE.

Definition 11 (Write-Complete). If T_i and T_j both write new values on K but T_i commits before T_j in the execution in Thunderbolt, T_i will also write the values on K before T_j when in SE.

Theorem 12. *Thunderbolt is both Read-Complete and Write-Complete if the dependency graph G is always valid.*

Proof. Firstly, if G is valid, we know that if there is a read node R_v^k reading a value from a write node W_u^k on key K, all the write nodes writing values on K either have a path to u or having a path from v to guarantee the correctness of read after write. Therefore, if transaction T_i reads values on T_j on key K, all other transactions writing values will not be assigned an order between T_i and T_j. Thus T_i will read the same value on T_j and Thunderbolt is Read-Complete.

Secondly, since SO is the commit order in Thunderbolt, if T_i commits before T_j, T_i will be assigned before T_j in SO. Thus, T_j will update the values after T_i in SE and Thunderbolt is Write-Complete. □

Theorem 13. *Thunderbolt is serializability iff Thunderbolt is both Read-Complete and Write-Complete.*

Proof. For any transaction T_i in SO, if T_i reads some values on the transactions $T_j \leq T_i$ in Thunderbolt, T_i must read the same values in SE since Thunderbolt is Read-Complete. If T_i writes some new values, since Thunderbolt is Write-Complete, T_i will write the same values in SE and all the transactions $T_j < T_i$ have been committed. Thus, transactions will produce the same outcomes in Thunderbolt and SE: $OUT = OUT'$. □

B.3 Proof of Safety

Thunderbolt produces the sequential order SO as well as the read-/write sets of each transactions. If the read/write set RS_i of transaction T_i overlaps the read/write set RS_j of transaction T_j and $T_i \rightarrow T_j$ in SO, T_j has a dependency on T_i. Therefore, if T_i dependents on T_j in one validator, other validators will have the same dependency. Finally, all the validators contain the same dependency graph generated by the read/write sets and SO. Thus, following the dependency graph to execute the transactions leading them to obtain the same outcomes.

B.4 Proof of Liveness

If all the replicas behave well, they will keep proposing the blocks in the same DAG. All the blocks proposed by each shard leader will be committed eventually. When a malicious replica is detected, honest replicas will propose a Shift block. If less than $2f + 1$ Shift blocks are proposed, the DAG will not be switched and all the replicas will stay in the current DAG and keep proposing the new blocks. If there are $2f + 1$ Shift blocks, all the honest replicas will switch to the new DAG at the same round (section 2). After at least $2f + 1$ honest replicas have relocated to the new DAG, they are able to propose the new blocks.

If all the replicas behave properly, they will consistently propose blocks within the same DAG. Each shard leader will eventually have the blocks they proposed committed Upon detection of a malicious replica, honest replicas will propose a Shift block. If fewer than $2f+1$ Shift blocks are proposed, the DAG will remain unchanged, and all replicas will persist within the current DAG, continuing to propose new blocks. If there are $2f + 1$ Shift blocks, all honest replicas will transition to the new DAG at the same round (section 2). Following the relocation of at least $2f + 1$ honest replicas to the new DAG, they will have the capability to propose new blocks.