Fair Transaction Ordering on DAGs

Preventing MEV extraction without sacrificing practicality

Giulio Segalini

Fair Transaction Ordering on DAGs

Preventing MEV extraction without sacrificing practicality

by

Giulio Segalini

to obtain the degree of Master of Science
at the Delft University of Technology,
to be publicly defended on Thursday, July 10th, 2025 at 15:00.

Student number:	5326648	
Project duration:	November 1st 2024 – July 10th 2025	
Thesis committee:	Dr. J. Decouchant,	TU Delft, supervisor
	Prof. dr. G. Smaragdakis,	TU Delft, chair
	Dr. B. K. Özkan,	TU Delft

Cover:	Giulio Segalini
Style:	TU Delft Report Style, with modifications by Daan Zwaneveld

An electronic version of this thesis is available at http://repository.tudelft.nl/.

TU Delft

Preface and Acknowledgements

This document contains the Master Thesis to fulfill the graduation requirements for the special program in Cyber-Security at Delft University of Technology. This has been a long process that has showed me aspects of research I did not expect.

It was very rewarding to see how my design decisions influenced the real-world behavior of the protocol. The trade-off between real-life applicability and usability, and raw performance was a big point of discussion during the design process.

It was quite demoralizing seeing how common it is to prefer high numbers and theoretical guarantees instead of novel and realistic protocols. I decided to strongly oppose this view. While theoretical soundness is important, science is supposed to produce output that is useful for the progress of society. If researchers only try to maximize specific metrics or only care for theoretical elegance, we lose the objective of creating something meaningful.

I want to greatly thank professor Jérémie Decouchant for his continuous support. I never felt like I was left alone in working on this project, and he was always available when necessary.

I want to thank Jianting Zhang for providing the code used in their paper "No Fish Is Too Big for Flash Boys! Frontrunning on DAG-based Blockchains" [83]. This allowed me to evaluate my work on a set of attacks already shown to be effective on top of Narwhal and Tusk.

I also want to thank Professor Mohammad Sadoghi, Dakai Kang and Shaokang Xie from the University of Davis, California. While they did not directly contribute to this work, discussing and confronting on similar topics and technologies showed me how vast are the possibilities of this field. I hope we will be able to collaborate in the future, as I'm pretty sure this will not be the last time we will see each other.

I hope the readers will enjoy these pages. The amount of effort put into this work is considerable, and I hope this shows in the words I chose.

Giulio Segalini
Delft, July 2025

Summary

Blockchain technologies enable the decentralized storage and verification of records, such as financial transactions. Systems like Bitcoin [56] and Ethereum [78] see a considerable usage and have market values in the order of 10s of billions of dollars[1]. A recent evolution of blockchain consensus systems, which have traditionally been maintaining a chain of blocks of transactions, is the introduction of Directed Acyclic Graph (DAG)-based algorithms [75]. In these new systems, popularized by companies such as Sui[2] and IOTA[3], transactions are positioned in rows of blocks that are connected in a DAG. The participants of these systems add blocks to the DAG in parallel, which has been shown to result in higher throughput, lower latency and higher decentralization [75].

Some blockchains, including DAG-based ones, support smart contracts [72], which enable the execution of (almost) arbitrary code with correctness, verifiability, and consistency guarantees. A natural usage of smart contracts is Decentralized Finance (DeFi) [65]. DeFi implements services that are offered in traditional centralized finance, such as loans and currency exchanges, in a decentralized fashion [65]. For example, instead of a central authority possessing different currencies and fixing their exchange rates through supply and demand of their clients, a decentralized exchange (DeX) can match buyers and sellers and update the exchange rate on the fly based on the supply of the two tokens.

The promise of safely removing the need for a central authority and regulations is however undermined by current blockchain implementations. Intuitively, the decentralized nature of DeFi should prevent malicious behavior, but in practice validators (i.e., the entities that maintain the blockchain) have the possibility to do so. For example, in Ethereum, miners have full control over the content of blocks [22]. This means that they have the power to reorder and inject transactions. The profit that is obtained this way is called Miner Extractable Value (MEV) [80].

MEV is not desirable because it introduces hidden costs for correct users: higher fees, worse exchange rates and network congestion [22]. Different mitigation techniques have been proposed over the years [80]. The most commonly used in practice are systems that allow the re-distribution or an easier extraction of MEV [25, 29, 30, 27]. These have significant ethical concerns, as they increase resource consumption while still maintaining the negative side effects of MEV on the average consumer.

It is also possible to anonymize the content of transactions until after their execution, making it more difficult for attackers to exploit their contents for profit [51, 80]. While anonymizing transactions makes an attacker's task more difficult, it requires the use of expensive cryptographic primitives or the presence of a trusted third-party, and transaction metadata might still be exploited to enable MEV.

The most interesting concept from the academic community against MEV is **Order Fairness**, which promises that transactions will be executed in an order that accurately represents the actual order in which the transactions were made public [85, 42]. These systems have been mainly studied as an extension of existing linear consensus systems, but have been scarcely applied to DAGs. Indeed, DAGs have been thought to have some inherent resistance to MEV extraction, but recent work by Zhang et al. [83] has shown otherwise.

To the best of our knowledge, the only system that combines DAG-based consensus and Order Fairness is FairDAG [39]. While this protocol provides strong theoretical guarantees, we will show that it lacks real-world applicability and contains some suboptimal design decisions.

In this work we propose Tilikum, the first DAG-based consensus protocol that upholds Order Fairness through Ordering linearizability without sacrificing usability and improving average throughput by about 15 times.

[1] https://coinmarketcap.com/
[2] https://sui.io/
[3] https://www.iota.org/

Contents

Nomenclature

Abbreviations

Abbreviation	Definition
BFT	Byzantine Fault Tolerance
DeFi	Decentralized Finance
DeX	Decentralized Exchange
MEV	Maximum Extractable Value
SMR	State Machine Replication
ASR	Attack Success Rate
TEE	Trusted Execution Environment
PoW	Proof-of-Work

1 Introduction

Since the introduction of Bitcoin [56], cryptocurrencies and blockchain technologies have remained a continuous focus of academic research and industrial development. Bitcoin and other cryptocurrencies enabled only one specific use of blockchain technology, i.e., payments, until Ethereum [78] was designed in 2014. Ethereum introduced smart contracts, which allow generalized applications to run on top of blockchains, achieving strong security guarantees, such as integrity, transparency, pseudo-privacy, decentralization, etc. One particular application of smart contracts that has gained traction is Decentralized Finance (DeFi), systems that offer traditional financial services without the need for trusted intermediaries [65].

Transactions performed through a DeFi smart contract are often order-dependent, as the outcome and magnitude of inputs/outputs of a set of transactions depend on the status of previous transactions. This enables transaction reordering attacks where actors try to extract value by manipulating the order of other parties' transactions or by inserting their own at specific locations. This is mainly performed by miners of blocks and so the amount of extractable profit takes the name of *Blockchain Extractable Value (BEV)* or *Miner/Maximal Extractable Value (MEV)*. Miners have an inherent advantage in performing these attacks, as the contents of the mined blocks fully depend on what they decide should be included and in which order.

Simple transactions and exchanges between two parties are not susceptible to these attacks, but other applications are. Some major examples are lending protocols, which dynamically adjust interest rates depending on previous interactions, and decentralized exchanges, where buyers and sellers of different tokens are automatically matched based on previous market interactions. EigenPhi[1] computed that more than $2M\$$ were extracted in the last 7 days. The Flashbot organization, through their MEV-Explore platform [26], in 2022 estimated that up to $100M\$$ per months were extracted. These numbers result in an opaque tax paid by users directly or indirectly through higher fees or worse exchange rates [22].

Many solutions have been proposed to mitigate the problems arising from MEV attacks, but they differ greatly in their design philosophy. Some of them try to democratize the process by allowing everyone to reap profits from positioning their transactions in favorable positions, others achieve fair transaction ordering guarantees. A promising method involves computing what would be a fair way to order transactions based on the order in which different replicas received them [42, 85]. These systems first define a notion of fairness, either using some absolute time indicators or relative orderings between pairs of transactions, and then extract a fair ordering from each replica's proposals.

These methods have been mainly designed for the linear blockchains such as Ethereum, as they are currently the main supports for smart contracts, but many blockchains based on different technologies and block structure exists and have been growing in importance. For example, some systems place their transactions/blocks in a Directed Acyclic Graph (DAG) [75], with promises of higher scalability and transactions completed in order of magnitudes faster times without security compromises.

[1] https://eigenphi.io/, accessed in May 2025

As shown by Wang et al. [75], there are multiple ways in which DAGs can be used to implement blockchain systems, as nodes in the graph can represent individual transactions or batches of transactions. Moreover, the graph structure itself can have different topologies and the consensus algorithms may employ different security systems and guarantee different properties. This also does not include systems where DAGs may be involved different protocol layers. For example Narwhal [23] is a DAG-based mempool which can be combined with a consensus protocol to implement a distributed ledger.

The first studies to analyse the susceptibility of DAG-based ledgers to MEV attacks have been released in 2025 by Zhang et al. [83] and Mahe et al. [49]. These studies showed that a specific, although very popular, DAG-based design paradigm is vulnerable to a new set of attacks. The number of consensus protocols based on DAGs that implement a fair ordering mechanism is also limited. The state-of-the-art approach, FairDAG, was published in April 2025 by Kang et al. [39]. We will however discuss why we believe that this system made some design choices that limit its use in practice.

In this work we propose Tilikum, a DAG-based ledger protocol based on Narwhal and Tusk [23] that implements transaction fair ordering through the ordering linearizability property [85] while also maintaining the applicability and performance characteristics of currently deployed systems. We thoroughly evaluated Tilikum, and compared its design decisions to the state-of-the-art baselines. Moreover, we showcase Tilikum's ability to defend against attacks that have been shown to be successful on DAG-based protocols.

The development on Tilikum, and this report as a whole, aim to answer the following research questions:

- **RQ1:** Can a DAG-based protocol implement order fairness?
- **RQ2:** What is the performance overhead of order fairness on this protocol?
- **RQ3:** Is this protocol able to defend against DAG-specific reordering attacks, such as the ones from Zhang et al. [83]?
- **RQ4:** How does this protocol compare to FairDAG [39], the state-of-the-art fair ordering solution for DAGs?

2

Blockchain and DeFi

This chapter describes and defines the important terminology and the background necessary to understand the remainder of this work. First, we describe some concepts related to linear blockchains that enable smart contracts. Then we talk about some applications of Decentralized Finance that are susceptible to MEV extraction. This chapter concludes with a deep dive into Narwhal and Tusk [23] as they serve as the base for our solution.

2.1. Linear Blockchains

In this section, we first describe the general structure and procedures necessary for transaction inclusion in blockchains. The second section discusses the most common application for blockchain technologies: cryptocurrencies. The third section defines the concept of smart contracts and why new security issues arise from their use. The last section explains the difference between the consensus and execution steps.

2.1.1. Transaction Proposal and Inclusion

When a user plans to execute a transaction, they broadcast it through the peer-to-peer network, entering it in the **mempool**, which is the public waiting area for transactions. All transactions that are not yet included in a **block** are present in the mempool.

Blocks are the main components of the blockchain, which contain information about confirmed transactions and the hash of the previous block. Blocks are built by **miners** (or validators), which are actors in the system that use their processing power to create new blocks and make the whole system progress. Miners do not do this without incentives, and they receive rewards for their participation. These rewards may come from clients whose transactions are included (fees) or from the network itself (block rewards).

When a transaction is inserted in the mempool, the user indicates the **fee** they are willing to pay. When a miner includes the transaction in a block, they obtain (part of) the fee. Miners usually try to prioritize transactions with high fees as they will obtain higher profits. It is important to notice that the mempool is public, and that miners have complete control over which transaction they want to include and in which order, which is why MEV attacks are possible.

2.1.2. Cryptocurrencies

The use of Blockchain technologies was popularized by Bitcoin [56], which is a Proof-of-Work system that enables the usage of a fully decentralized currency. In Bitcoin, miners need to perform random but computationally-intense operations to identify values that allow them to create new valid blocks. These operations are hash reversing operations where they need to find a pre-image starting with a certain number of 0s. When such a value is found, a miner can choose a set of transactions from the mempool to be included in the new block. As explained before, miners tend to select transactions that give the highest fee to maximize their own profit.

In a simple system such as blockchain, which only allows users to exchange one type of currency, there is no dependency between transaction order. If client A is sending some funds to user B and B is sending some funds to C, assuming all parties have enough assets to perform these transactions, there is no difference in the final balances if the first transaction executed before the second one. We will see later how this is not the case when we introduce more complex systems that allow code execution on the blockchain.

2.1.3. Smart Contracts

The Ethereum [78] blockchain is the most used chain for DeFi applications [36], as it was the first to implement the concept of **Smart Contract** in a real system. Smart Contracts were proposed by Szabo in 1997 [72] as a transaction protocol that executes the terms of an agreement [86]. Through Blockchain Technologies smart contracts came to life and the current Ethereum implementation allows for arbitrary code to run.

The code running on smart contracts usually implements some system to track resources and assets owned by users that can be exchanged and transferred according to different rules. These programs enable the existence of **Decentralized Finance** (DeFi), online systems that provide services comparable to the ones provided by traditional financial institutions. Section 2.2 explains two common DeFi applications.

The introduction of smart contracts enabled a new type of attack: transaction ordering manipulation. If before, the only way transactions could depend on each other was related to the presence of sufficient funds for future transactions (i.e., a transaction increased the balance enough to perform a second one), now smart contracts may change their internal state after execution of a specific transaction, changing the outcome for future ones. Let us illustrate this point with a simple example. A smart contract implements a simple cryptocurrency system where users do not choose the fee, but it is instead calculated based on how many transactions were executed in total. The first transaction has fee of 0, the second of 1, the third of 2, etc. It is clear that in this case users would want their transactions executed before others, as they would pay the least amount of fees, and that the execution of a transaction can influence the result of future ones.

2.1.4. Consensus vs. Execution

In the majority of the blockchain systems, after a block and its transactions have been confirmed enough times and consensus has been reached, all its contents can be executed. In this context, **Execution** means carrying out the operations described by the contents of a transaction (that is, exchanging currency or executing code in the context of a contract) and applying them to the global state of the ledger.

Many blockchain systems decouple execution from consensus to better abstract these two concepts from each other, as they do not necessarily coincide. The consensus step is interested in making sure that every replica sees the same blocks in the same order and that their contents are available to everyone. The execution step, which can only happen after consensus, is responsible for the actual changes that the agreed blocks will bring to the state.

2.2. Decentralized Finance Basics

This section presents two applications of Decentralized Finance: exchanges where different tokens and assets can be swapped, and platforms that enable loaning of assets.

2.2.1. Decentralized Exchanges

Decentralized Exchanges (DeXs) are peer-to-peer applications and protocols that enable parties to exchange cryptocurrencies without the need for intermediaries [48]. If traditional exchanges operate using the limit order book mechanism [32], DeXs implement using smart contracts systems called *Automated Market Makers (AMM)*. AMMs automatically compute the current exchange rate at which currencies should be traded depending on the trade size and the current amount of liquidity present for both currencies [1, 84].

As exchange rates are dynamically updated after every transaction, the amount of output tokens received for the same input continuously changes, even if by small amounts. This is the key detail that enables

profit extraction, as the order in which transactions are executed influences the exchange rate.

2.2.2. Lending Platforms

DeFi lending platforms enable parties to borrow currencies from each other without intermediaries. Loaners deposit the excess cryptocurrency they want to lend in a liquidity pool, and borrowers can withdraw currencies while paying lenders interest. Borrowers also need to deposit assets as collateral.

In the most common lending protocols, the collateral must be at least as big as the loaned amount [15], making all loans over-collateralized. In case the collateral value drops below a pre-determined threshold, the protocol *liquidates* the loan. There are mainly two liquidation strategies: auction liquidation and fixed spread liquidation [36, 64]. The first method, used by MakerDAO [50], puts the loan for liquidation in an auction system. The older version utilized an English-style auction, where the price grows upward until only a single bidder is willing to pay. Newer versions of the protocol use the Dutch auction system, where the price decreases until the first buyer available for purchase confirms it.

The second liquidation strategy makes the liquidated loan available immediately for purchase at a pre-determined discount [64, 53]. Both strategies may suffer from MEV attacks, as attacker can prioritize their bids over others.

2.3. DAG-based Blockchains

Traditional blockchain structures connect blocks linearly and, using the concept of Nakamoto Consensus [56], the longest chain among multiple valid ones will survive. This usually results in very low transaction throughput and scalability, and motivated the development of blockchains that rely on Directed Acyclic Graphs (DAGs) such as IOTA [63], Spectre [68], GHOST [69], Prism [8] and Sui [11]. It is important to notice that even if all these blockchains share a DAG-like structure, their design philosophies significantly vary. As summarized by Wang et al. [75] there are mainly 6 types of chains, with the most significant distinctions being the graph topology and the structure of a singular unit (node) in the graph. This taxonomy does not include more recent systems like Sui [11] where DAGs are present only in parts of the system, in this case the mempool.

There are many design decisions behind DAG-based ledgers, but the main common advantage is the decoupling of consensus and block dissemination. By removing the competitive element in block building, instead of a single block being added for each round, we can have multiple validators build a graph that can later be traversed for transaction execution.

The first design that proposed this separation was DAG-Rider [40]. DAG-Rider achieves block dissemination using reliable broadcast, which is used to build a graph of blocks. Then using a global perfect coin, obtained through replicas sharing randomness in their blocks, the system reaches consensus over blocks.

Each replica is allowed to submit one block per round r (rounds are logical) and this block needs to refer to at least $2f+1$ blocks from round $r-1$ and up to f blocks from rounds from $r-2$ or earlier. The first type of connection is called a **strong edge**, while the second is a **weak edge**.

DAG-Rider was the theoretical basis on which Narwhal and Tusk [23] were built. Narwhal and Tusk [23] are the two protocols used in the first implementation of Sui [11], the most popular DAG-based ledger. [1]

A natural evolution of Narwhal and Tusk is Bullshark [70], which assumes that nodes go through synchronous phases. This is achieved through the implementation of a fast path that exploits the synchronous periods and removes the need for view-change and view-synchronization mechanisms [70].

The current implementation of Sui[2] utilizes Mysticeti [7], which still separates block dissemination to build the DAG, and consensus, but also leverages Threshold Clocks to introduce a new committing rule. This rule can be pipelined and supports multiple leaders, which is shown to greatly decrease latency.

[1] Sui, a blockchain and a token implemented on top of Narwhal, is the most popular DAG-based token on https://coinmarketcap.com/. We describe those protocols in more details in the following section. Sui has now moved away from Narwhal, but its influence on its current design is still noticeable.

[2] https://sui.io/

Narwhal and Tusk served as the theoretical basis for more protocols. Shoal [71] integrated leader reputation and pipelining to decrease commit latency. These additions are protocol-agnostic and can be adapted into other works as well. Shoal exploits the fact that Byzantine faults are more rare than slow or crashing nodes. Leaders in a round are dynamically adjusted according to a function that takes into account the fault rate of each node so far.

Shoal++ [5] expanded upon Shoal by building a novel protocol with three key concepts. First, the commit rule is optimized for the common case. Next, leader election latency is eliminated by attempting to elect every possible node as leader, which is then dynamically adjusted using a method similar to Shoal's reputation system [71]. Finally, multiple DAGs are actually instantiated in parallel, with their final logs intertwined, enabling missed transactions to be included in a different graph.

Mahi-Mahi [38] is a protocol proposed by Jovanovic et al. which uses an uncertified DAG to achieve lower latency. A new commit rule which reduces the expected delay was developed, as blocks are not reliably broadcast in this new DAG construct. This rule uses a 5-rounds wave to implicitly certify blocks. The blocks in the latest round of the wave are called certifiers, while the ones in the round just before are voters. A voter block is a vote for a block in the first round of the wave if this block appears in its casual history. Blocks in the first round of the wave are considered certified if a certifier block contains at least $2f+1$ voter blocks in its casual history.

Starfish [62] is a partially synchronous DAG-based BFT protocol with linear amortized communication complexity. It leverages a push-based system where validators disseminate full transaction data for their own blocks, but only encoded shards for others. When $f+1$ shards are collected, it is then possible to reconstruct the full data. It also uses an extension of the uncertified DAG from Mahi-Mahi [38]. The commit rule is expanded to verify data availability and possibly reconstruct needed blocks.

Ladelsky et al. [47] performed an analysis on the difference in quorum sizes between different DAG-based protocols, focusing in particular on DAG-Rider [40], Narwhal and Tusk [23], and Bullshark [70]. They found out that all properties of DAG-Rider are actually maintained when $N = 2f+1$, assuming there exists a method achieving reliable broadcast with such quorum size. For the asynchronous version of the other protocols, it is still required that $N > 3f$, while the partially synchronous version of Bullshark has the same requirements as DAG-Rider.

Fides [79] achieves DAG-based BFT consensus in the case of $N = 2f+1$ through the use of Trusted Components, TEEs. The system builds on top of DAG-Rider [40] and Tusk [23]. TEEs assist in Reliable Broadcast, Vertex Validation, Leader Election and Transaction Disclosure. As transactions are kept private until commitment, this system ensures censorship-resistance and implicitly defends against MEV extraction.

The rest of this section focuses on Narwhal and Tusk [23], as it became the foundation for many more DAG-based protocols, including Bullshark [70] and Mysticeti [7].

2.3.1. Narwhal and Tusk

In this section we describe the design of Narwhal, a DAG-based mempool, and of Tusk, a consensus protocol to commit and order transactions in the Narwhal mempool [23]. First we provide a high-level description of the system and its advantages. Then a detailed explanation of the Narwhal Mempool and the Tusk consensus protocol follows. The section ends with some observations and considerations on the security and fairness of the protocols.

High-Level Description

The main contribution of the paper is proposing to separate the transaction dissemination task from transaction ordering and commitment. This is achieved through a novel mempool protocol called Narwhal, which positions transactions in a DAG, that can be combined with a consensus protocol to decide on a total order of blocks in the graph.

Traditional mempools are implemented through a best-effort gossip system [56, 78] where clients submit transactions to a subset of nodes who then broadcast it to all other participants. While this is a simple system, it leads to the double transmission of transactions, once for mempool dissemination and a second time when a block containing them is mined and shared. Narwhal tries to solve this double transmission problem while also enabling scalability [23].

In this mempool nodes transmit blocks instead of individual transactions. Blocks are consistently broadcast to the network resulting in the creation of an availability certificate in a sufficient amount of nodes ($2f+1$). Blocks are also integrity-protected by the mempool through digital signatures certified by other nodes. They also need to refer to enough previous block availability certificates, increasing the utility of certificates of availability. Instead of only confirming the presence of a single block they now testify that the block's casual history can be downloaded as well if necessary.

This design enables the mempool to be continuously expanded without influencing the consensus layer, blocks inserted in the mempool will eventually be committed and ordered. However, it does not address the fact that malicious nodes can spam the network with blocks, forcing other nodes to download large amounts of data and stalling legitimate transactions from getting confirmed. To address this issue Narwhal uses a round system in which each node can submit only one block per round, which also needs to include a quorum of certificates from the previous round [23]. Nodes also cannot advance rounds unless a majority of honest nodes has concluded the previous round.

While all these operations can be performed by a single process, it is possible to split block creation into smaller batches who are handled by worker nodes and combine these batches into a single block by a primary node. Narwhal leverages this separation to scale the system efficiently.

The Narwhal mempool can be combined with a consensus protocol (e.g., HotStuff [81]) to provide a functional distributed ledger. To better take advantage of the Narwhal design the authors also developed Tusk, which is an asynchronous consensus protocol requiring no extra communication. Tusk is built on top of DAG-Rider [40] and allows nodes to take the local view of the casually-ordered DAG provided by Narwhal and extracts a consistent total-order of its blocks.

Narwhal Mempool

Narwhal is a mempool and as such is required to provide specific abstraction to the involved parties. These abstractions mainly refer to a set of operations any node can perform on the DAG that will return the same result for every correct replica. These are:

1. $write(d, b)$: Store a block b with digest d in the mempool. This returns a value $c(d)$, an unforgeable certificate of availability. The $write$ operation succeeds if a valid certificate is returned;
2. $read(d)$: Returns the block b associated with digest d, if and only if a previous $write(d, b)$ operation succeeded;
3. $valid(d, c(d))$: a boolean function returning whether the certificate is valid;
4. $read\_causal(d)$: returns all blocks b such that there is a happened-before relation to the block returned by $read(d)$.

The Narwhal mempool operates under the assumption of N participating node (also referred to as validators) and at most $f < N/3$ faulty nodes, which have Byzantine behavior [13]. For simplicity, we will consider a validator to be a single complete entity, but the authors showed that to increase scalability and throughput it is possible and advised to separate nodes into smaller worker nodes that communicate with a primary one that handles communication with other parties.

Each validator maintains the current round number r and continuously listens to clients that want to submit transactions. Validators also listen for and collect certificates of availability from other nodes. When certificates from at least $2f+1$ distinct parties have been collected the node can advance to the next round, starting by increase the local round number.

The transactions collected so far can be assembled into a block which is then reliably broadcast [18] to all other validators. Narwhal does not implement a push strategy, but instead uses a pull strategy to reduce the communication overhead [23]. When a validator receives a block from another node during the previously described step they check if it is valid. For a block to be valid it needs to:

- Be correctly signed by its creator;
- Belong to the same round as the validator checking it;
- Either belong to round 0 or contain at least $2f+1$ block certificates from the previous round;
- Be the first and only block received from this creator during this round.

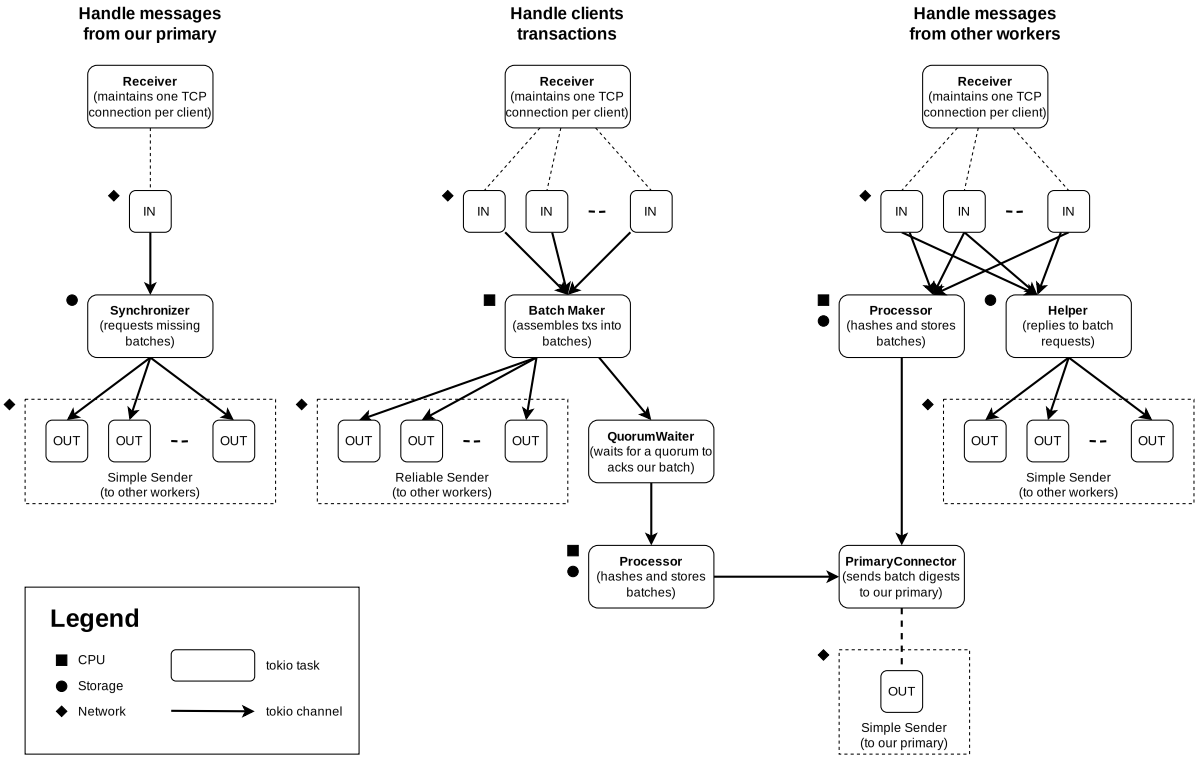


Figure 2.1: Worker architecture in the Narwhal protocol [23]

If all the conditions are met, the validator will store the block and respond with a signature certifying its digest, creator, and round. As blocks round and local round need to coincide, it is possible for older blocks to be completely dismissed by validators. However, as each valid block contains at least $2f+1$ previous certificates, the reception of even a single future block is enough to convince a node to advance its round.

Once a block creator gets $2f+1$ distinct signed acknowledgments for a block, it combines them into a block availability certificate, which includes the signatures, round number and info about the block and its creator. This certificate is then broadcast to be included in future rounds blocks.

To achieve the maximum potential of Narwhal some practical points need to be addressed: how to achieve reliable broadcast without the need for perfect channels and how to scale-out validators.

The first important intuition is to stop handling messages that are related to previous rounds. As there are certificates of availability for previous rounds, a certified block in round $r+1$ guarantees that all $2f+1$ blocks it refers to are also available and so can be requested from the validators that signed the certificate. At least $f+1$ honest nodes will store the block, yielding high chances of a positive response.

Scaling of validators is achieved by employing a simple primary-worker architecture. The worker process operates the architecture shown in Figure 2.1, while the primary operates the one shown in Figure 2.2.

Clients communicate with workers, who create transaction batches. These batches are then broadcast to other workers. When a quorum of acknowledgments is received, the worker can then send the batch (or for more efficient bandwidth usage, its hash) to its primary. Now the primary can run the normal Narwhal protocol, but instead of including transactions in blocks it will include hashes of batches from their workers. Other small modifications are necessary to make sure that primaries check that the batches contained in the blocks they receive to be certified are stored by their workers and, that if this is not the case, they can be pulled by workers from other workers.

The mempool can be combined with an eventually synchronous consensus protocol to totally-order the transactions in the DAG and implement a consistent distributed ledger, but Narwhal itself can be

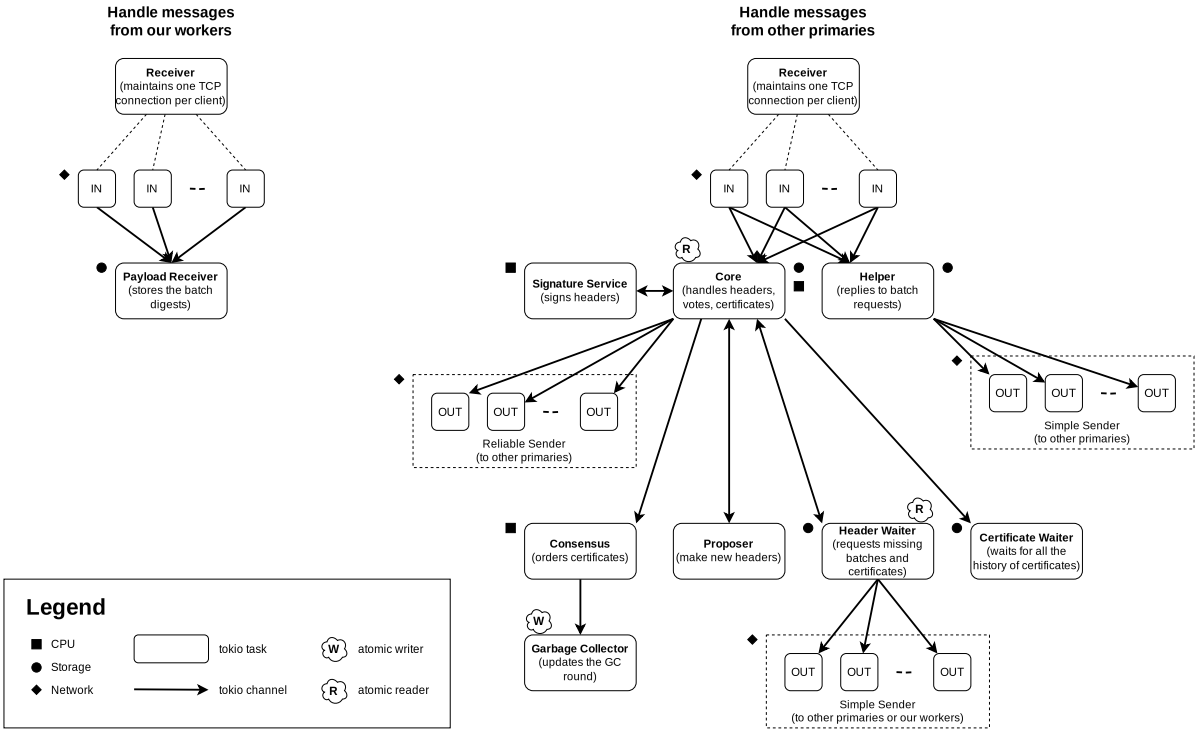


Figure 2.2: Primary architecture in the Narwhal protocol [23]

expanded to enable asynchronous consensus with no extra communication. This protocol is called Tusk and is explained in the next subsection.

Tusk Consensus

Tusk is an asynchronous consensus protocol based on DAG-Rider [40]. Any node participating in Tusk operates a Narwhal mempool, and include in each block some additional information that is necessary to create a perfect random coin. Validators conceptually divide the DAG into waves, each being 3 consecutive rounds. In the first round each node proposes its block, including its causal history. The second round has each validator vote on the proposals, by including them in their own block. The last round creates randomness to elect a random leader from the first round of the wave [23].

The leader is committed if there are at least $f + 1$ blocks referring to it in the following round (the second of the wave). The causal history of this leader block can then be ordered deterministically. As different validators may have different views of the DAG, it is possible that some leader blocks are not committed in all nodes. To make sure that these are eventually committed by every validator, when a wave has a successfully committed leader the validator starts a recursive process. The current leader block becomes a candidate and the previously committed leader block is searched. For every wave with an uncommitted leader in between these two blocks, the validator checks if there is a path between the uncommitted leader block and the candidate. If such a path exists, then this older block (and its causal history) is ordered before the candidate. This step is executed recursively with the older leader becoming the new candidate.

This mechanism is similar to DAG-Rider [40], but uses waves of three rounds instead of four, giving better termination guarantees in the case where messages are randomly delayed.

Censorship and Fairness in Narwhal and Tusk

Narwhal and Tusk do not explicitly try to address fairness. Validators still have full control over what is included in a block and in which order. Moreover, the system does not give any indications to clients on how they should transmit their transactions to make sure they are included.

One of the most important contributions that makes achieving the high throughput of Narwhal possible is removing the need for transaction retransmission. To maximize transactions throughput, and to

minimize latency, clients are encouraged to send their transactions to only one node. [3] While this leads to significant improvements in the usability of the system, it enables malicious nodes to fully censor some or all transactions they receive, as there is no guarantee that a correct replica will also be able to insert it in a block at some point.

To address this problem the authors recommend clients to submit their transaction to at least k nodes instead, as the probability of having at least one correct node receive the transaction increases exponentially. This mitigates the issue, but does not give any guarantees, and it does not address the possibility of re-ordering being performed by the validators.

[3] This is also the case in the experimental setup of Narwhal.

3

Transaction Reordering and Order Fairness

This chapter describes why transaction reordering is common, its consequences and existing mitigation techniques.

The first section describes the primitive operations that an attacker can perform to extract profits by reordering transactions. The second describes what attacks are enabled by these operations and how they work. The third section describes the current directions that defenses against transaction re-ordering attacks are taking. The fourth section explores in more detail the literature on fair ordering techniques. The last part of this chapter focuses on one of the recent protocols, Pompē [85], a fair ordering protocol for Byzantine ordered consensus. Pompē is one of the seminal works in the field and its definitions and properties are fundamental to this work.

3.1. Transaction Reordering Attack Primitives

In any blockchain where miners control transaction orders, there are four main types of possible transaction reordering attack strategies [36, 80]. These operations enable an attacker to exploit fluctuations in market prices caused by the underlying smart contract implementations.

Front-Running

An attacker can place their transaction (or someone else's they are colluding with) before a victim transaction. This is usually performed to obtain favorable rates or to snatch valuable deals. Front-running is illegal in many jurisdictions for the traditional financial market [52, 66, 54], but the unregulated and online nature of cryptocurrencies makes this type of control hard, if not impossible.

Fatal Front-Running

This is a special case of Front-Running in which the introduction of one or more transactions before the victim's one makes the latter fails. This special type of attack is typically targeted towards specific entities or used in situations where a transaction not happening is necessary. Any type of transaction that removes an asset from the market, for example acquiring a unique product such as an NFT, is subject to this type of attacks, as a successful front-run transaction obtaining this resource will block anyone else from the purchase. An example is shown in subsection 3.2.1.

Back-Running

Back-running is the practice of placing an attacker's transaction after a victim transaction. This type of attack is common when an attacker wants to be the first to claim a specific deal announced through another transaction, such as the liquidation of a loan or the sudden change in market valuation of specific assets. More details in subsection 3.2.2.

Sandwich Attack
With a combination of front- and back-running attacks, an adversary places at least two transactions around a victim transaction. The first set of transactions, happening before the victim's one, enables the attacker to acquire assets that change the status of the market advantageously for the attacker and in a possibly negative way for the victim. The second set of transactions allows the attacker to sell the previously acquired assets at a profit.

Bribery
All previously mentioned attacks assume a malicious miner that acts for their gain. It is also possible for miners to get bribed to perform attacks on someone else's behalf [80]. This is not a new attack strategy per se, but it is important to remember when defining threat models as a rational miner may not be willingly performing an attack, but just act in their best interest [77].

3.2. Reordering Attacks

All the previously described attacks are feasible on most blockchains, but some have more specific usages depending on the type of smart contract under attack. In this section, we first describe the attack possibilities when using (fatal) front-running, then the ones with back-running, following with sandwich attacks and an explanation of (cyclic) arbitrage. The section concludes with a small overview of recently proposed attacks that are specific to DAG-based systems that use DAG-Rider [40] or Narwhal and Tusk [23].

3.2.1. Front-Running Based Attacks

Eskandari et al. [24] defined three main types of front-running attacks.

Displacement attacks are specific to fatal front-running as they do not require the victim's transaction to successfully execute, but actually rely on it failing.

As an example, a miner, or someone that bribed them, could be able to snatch a deal before anyone else even if time-wise their transaction was not first. This is already known to have happened with the NFT "CryptoPunk 3860". [1] In this situation, the front-running transaction executing successfully will prevent anyone else from buying the same resource.

Insertion attacks aim to change the smart contract's state to a more favorable position so that profit can be extracted from the victim's transaction. For example, let's assume a victim has placed a transaction offering to buy an asset for price P while an attacker notices that the same asset is also available at price $Q < P$. They can then buy the asset at the cheaper price and place this transaction before the victim one, making a profit.

Suppression Attacks, where enough other transactions are front-placed to delay a victim's transaction as much as possible. This is similar to fatal front-running, but completely blocking the transaction is not the specific goal.

3.2.2. Back-Running Based Attacks

The main objective of back-running based attacks is to execute a transaction immediately after another specific one. One common situation is to take advantage of delays in exchange rate updates between different services. A new transaction T_1 can significantly increase the price of an asset on exchange E_1, while exchange E_2 will not update its rates until the transaction has been confirmed in a block. An attacker may use this information to back-run a transaction buying the asset at the cheaper price so that it can be sold later at the new higher price.

Another situation where back-running is common is when assets are liquidated as a consequence of an under-collateralized loan [64]. As explained in Section 2.2.2, some platforms liquidate loans through fixed spread liquidation and so the assets are sold at a discount immediately. This is a perfect scenario for an attacker to place their transaction buying the liquidated assets immediately after they are officially announced on the blockchain.

[1] https://etherscan.io/tx/0xb40fd0c9a2ba2d1d5e7ee5e322f9afc5e2ec1b7e2d520b638ea83dcc9c850d02

3.2.3. Sandwich Attacks

Sandwich attacks are common when an asset can be bought at a certain price in the front-run transaction and later sold at a higher price in the back-run transaction, as the sandwiched victim's transaction will increase the price of the good. This is especially common in DeXs, we explain why in the following. As explained in Section 2.2.1, AMMs regulate the exchange rate of currencies on the fly. The four biggest DeXs using the Ethereum blockchain all operate using AMMs, in particular, constant function market makers (CFMMs) [3]. These systems adjust trading rates by making sure that a specific function over the available amounts of two tokens stays constant.

A common implementation of CFMMs uses Constant Product Market Makers (CPMMs). In a CPMM the exchange will hold liquidity pools of different tokens and for each combination (X, Y) of two tokens the product between the amount of available liquidity is maintained constant by adjusting the exchange rate. If the current amount of tokens of type X at time t is X_t and the number of tokens Y is Y_t the market will always maintain at any given moment that $X_t \cdot Y_t = c$ where c is a constant. Assuming that a party wants to exchange δ_X tokens of type X and that the number of tokens of type Y they will receive is δ_Y, the new state of the market should stay constant, so $(X_t + \delta_X) \cdot (Y_t - \delta_Y) = c$. It is then easy to see that the amount δ_Y will be:

$$\delta_Y = \frac{\delta_X \cdot Y_t}{X_t + \delta_X}$$

Exchanges usually require a fee $0 < f < 1$ to operate, which can be obtained by applying a coefficient $1 - f$ to any occurrence of the term δ_X.

Let us assume that a malicious miner will notice that a victim is trying to exchange some tokens Y for some tokens X. The attacker can then front-run this transaction by acquiring some tokens X which will increase their price on the same exchange. The victim will then receive a worse deal than what they expected and further inflate the price of the tokens. The attacker can then back-run this transaction by selling the tokens X they acquired for a larger amount of token Y than they spent before the attack.

This type of transaction reordering is detrimental not only as it enables an attacker to obtain arbitrary profits, but also because it directly affects the user's exchange rates, moving people away from using these services and increasing prices.

3.2.4. Arbitrage

If sandwich attacks rely on specific transactions changing the exchange rates for a pair of tokens, it is possible to exploit differences in rates between different DeXs. This practice is called **Arbitrage** and exists in many traditional markets as well.

In the context of DeFi, the most common type is **Cyclic Arbitrage** and EigenPhi[2] estimates that in the last 7 days 4.5M of profits were extracted through it. If at least 3 exchanges can be identified with discrepancies in exchange rates, it is possible to extract profits. Let us assume that we find 3 markets A, B and C. The attacker starts with token X and notices the following exchange rates: market A exchanges token X to Y with rate $1 : P_{XY}$, market B exchanges token Y to Z with rate $1 : P_{YZ}$, and market C exchanges token Z to X with rate $1 : P_{ZX}$. If $P_{XY} \cdot P_{YZ} \cdot P_{ZX} \geq 1$, the attacker can obtain a profit by cycling the three exchanges. Another example with 4 tokens is shown in Figure 3.1.

3.2.5. Reordering Attacks on DAG-Based Blockchains

To the best of our knowledge, there is currently no large-scale study on the feasibility of different transaction re-ordering attacks specific to DAG-based blockchains. The first studies to conduct an analysis in this direction were released in 2025 by Zhang et al. [83] and Mahe et al. [49] and showed that a specific, although very popular, DAG-based design paradigm is vulnerable to a new set of attacks. The design paradigm is the one first proposed by DAG-Rider [40] and expanded upon by Narwhal and Tusk [23].

Zhang et al. [83] created three attack strategies with high success rates that are specific to the way Narwhal and Tusk are implemented. The attacks are: (i) the Fissure attack, where malicious actors try to disconnect victim blocks from the DAG; (ii) the Sluggish attack, where attackers delay their block

[2] https://eigenphi.io/, retrieved on 11/06/2025

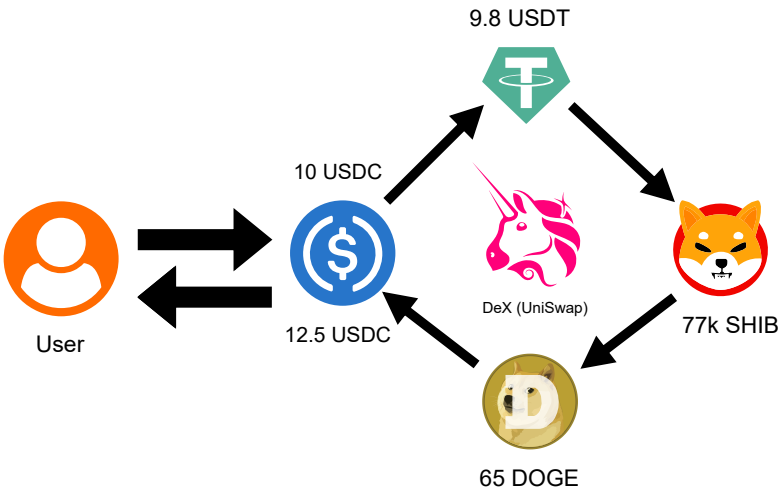


Figure 3.1: An example of a possible cyclic arbitrage (inspired from [76])

proposal so that they appear in a round before the victim; and (iii) the Speculative attack, where blocks with small digest values are built. These are explained in detail in Section 5.4 as they were used in the evaluation of Tilikum's robustness.

Mahe et al. [49] evaluated an attack strategy similar to the Fissure attack on DAG-Rider [40] and also measured the impact of Byzantine behavior applied to the consensus layer. They obtained similar results, showcasing the need for mitigations that are specific to the DAG structures. They also analyzed how various design decisions amplify the effect of Byzantine behavior on the system and how some mistakes in previous work could be addressed.

3.3. A Taxonomy of Transaction Reordering Defenses

Yang et al. [80] have defined four main categories of defenses and countermeasures against MEV extraction attacks.

MEV Auction Platforms

These systems try to make MEV extraction easy and more transparent, democratizing the process. They usually try to maintain two properties for transactions submitted to it: **Atomicity**, which means that transactions submitted together have to all be executed in the required ordering or all fail, and transaction privacy, so that regular users that do not want to submit their transactions to the public mempool can have them inserted in the ledger without revealing their content.

There are mainly three types of users benefiting from using these protocols: regular users, MEV searchers and validators. MEV searchers are users that explore the mempool and find MEV profit opportunities, but do not have the processing power to operate as validators. Through the platform, they can ask a validator to include the bundle of transactions they recognized, in exchange for a fee. Regular users can use the privacy guarantees to hide their transaction content from MEV searchers. Validators sell block space for a fee that users can buy for transaction inclusion.

Time-Based Ordering Fairness

As blockchains effectively implement state-machine replication, it is possible to extend the protocol to guarantee an extra property on the order of committed transactions. This is order fairness, which can be defined in multiple ways, such as agreeing on the order of transactions the way the majority of nodes received them [42, 43]. If the agreed definition of order fairness is respected, then it is possible to prevent transaction reordering.

Content-Agnostic Ordering

If time-based ordering fairness may use the content and metadata of a transaction to ensure a fair outcome, content-agnostic ordering determines the order of transactions independently of their content.

This is a weaker constraint than the one described before, but it can still help mitigate some attacks, as it can remove the capability of viewing transaction content before they are executed from malicious actors.

MEV-aware Application Design

The other types of solutions try to mitigate or exclude MEV attacks on existing blockchain technologies. However, it is possible to design new systems that are resilient to transaction reordering by construction or to design smart contracts that mitigate the effects of MEV extraction. Some examples of MEV-aware applications are described in subsection 3.3.1.

3.3.1. Protection Mechanisms

As shown in Section 3.3, there are four main types of defense strategies in linear blockchains like Ethereum. In this section, we go through some recent systems that implement these four solutions, while also comparing their security and usability implications.

MEV Auction Platforms

The main idea of MEV Auction Platforms is to enable regular users and MEV searchers (people who want to realize MEV profits) to place bids to have their transactions included by miners who sell their block space through the platform. This type of system needs to guarantee two key properties: privacy and atomicity. Privacy of transactions is necessary as miners could see which set of transactions an MEV searcher would perform and front-run similar or identical ones to obtain the profits themselves. Atomicity is necessary as MEV's success depends on multiple transactions, and performing only some of them may actually lead to losses.

The first MEV auction platform was Flashbot [27], which allowed users to submit bundles of transactions that need to be executed in a specific order together with a promised payment for the miner that will include these transactions in a block. The auction platform acts as a trusted third party for miners and users.

After Ethereum switched to Proof-of-Stake as their consensus mechanism, MEV auctioning became natively implemented in the protocol through Proposer-Builder Separation [25], which separates the role of Block Proposer and Block Builder. As this is not yet fully implemented, users can use MEV-Boost [29], which achieves similar results. This requires trust in the builder ecosystems, or in a specific set of builders that users choose to trust.

Another system similar to MEV Auctioning is MEV redistribution, which promises users the ability to capture some of the profits that MEV searchers create using their own transactions. Currently, there are two implementations, MEV Share [30] and BackRunMe [12], that require full trust from their users.

Finally, users may just require privacy over their transactions to avoid miners using them to extract MEV or to make sure that their MEV profit is not compromised. MEV auction platforms usually provide this service as well in the form of private channels, such as Flashbot Protect [28].

MEV auction platforms are commonly implemented and used in practice, so much so that around 90% of transactions involve one of these systems [80]. These solutions differ drastically from the rest but remain the most used. Some examples of currently used MEV auction platforms are bloXroute BackRunMe [12], the EDEN network [61] and the Osmosis ProtoRev by Skip Protocol [60].

Enforcing Order

By enforcing a fair time-based order, it is possible to fully eliminate transaction reordering attacks. The main idea behind these systems is to enforce some definition of fairness, such as receive-order fairness as defined by Kelkar et al. [42] or Ordering linearizability as defined by Zhang et al. [85].

While these systems have strong theoretical guarantees, in practice they are seldom used as they require moving the user base to new platforms, while also losing the large number of people exploiting MEV. As this topic is of particular interest for the rest of this work a section is dedicated to it in section 3.4.

Content-Agnostic Ordering

Content-agnostic ordering, also known as blind-order fairness, determines transaction ordering with complete disregard for its content. Implementations commonly use a commit-and-reveal protocol.

Table 3.1: Comparison between fair ordering protocols.

Protocol	Fairness Property	Fault Threshold	Network Synchrony
Sync Aequitas [42]	γ-Batch-Order-Fairness	$f \leq \lfloor N(\gamma - \frac{1}{2}) \rfloor$	Sync
Async Aequitas [42]	γ-Batch-Order-Fairness	$f \leq \lfloor \frac{N}{2}(\gamma - \frac{1}{2}) \rfloor$	Async
Permissionless Aequitas [41]	γ-Batch-Order-Fairness	$f \leq \lfloor \frac{N}{2}(\gamma - \frac{1}{2}) \rfloor$ (PoW/PoS)	Async
Themis [43]	γ-Batch-Order-Fairness	$f \leq \lfloor N \frac{2\gamma-1}{2\gamma+1} \rfloor$	Async
SpeedyFair [55]	γ-Batch-Order-Fairness	$f \leq \lfloor N \frac{2\gamma-1}{4} \rfloor$	Partially Sync
Pompē [85]	Ordering Linearizability	$f \leq \lfloor (N-1)/3 \rfloor$	Async
Lyra [82]	Ordering Linearizability	$f \leq \lfloor (N-1)/3 \rfloor$	Async
AOAB [33]	Ordering Linearizability (+ Commit-Reveal)	$f \leq \lfloor (N-1)/3 \rfloor$	Async
Ciampi et al. [20]	Median-Based (+ Commit-Reveal) with TEEs	$f \leq \lfloor (N-1)/2 \rfloor$ (PoW/PoS)	Async
Wendy [46, 45]	Probabilistic Order-Fairness	$f \leq \lfloor (N-1)/3 \rfloor$	Async
QuickOF [17]	κ-Differential Order-Fairness	$f \leq \lfloor (N-1)/3 \rfloor$	Async
Alpos et al. [2]	Random Ordering	—	—
FairDAG-AB [39]	Ordering Linearizability	$f \leq \lfloor (N-1)/3 \rfloor$	Async
FairDAG-RL [39]	γ-Batch-Order-Fairness	$f \leq \lfloor N \frac{2\gamma-1}{2\gamma+1} \rfloor$	Async
Tilikum (this work)	Ordering Linearizability	$f \leq \lfloor (N-1)/3 \rfloor$	Async

First, transactions are committed using some metadata, which identifies the transaction (e.g., a digest). Then, miners can determine some order. Only after the order has been confirmed transactions are fully revealed, and their metadata is matched to place them in the correct order. The difference between different implementations mainly resides in the technologies involved in the commitment step. Fino [51], Sikka [67], Osmosis [59], and the Shutter Network [58] all implement threshold encryption, which allows users to encrypt their transactions only to be decrypted by a trusted committee when necessary. TEX [44] uses timelock puzzles, a way to automatically decrypt the transaction when specific conditions are met. Tesseract [10] uses Trusted Execution Environments (TEEs) to guarantee secure channels between users and the exchange, hiding transactions from front-runners.

Content-agnostic ordering is promising as it effectively reduces the chances of MEV attacks happening, but it is vulnerable to metadata leakage.

MEV-Aware Design

It is also possible to fully design a system, or parts of it, to fully prevent MEV extraction.

LO [57] designs a new mempool data structure that forces miners to log all the transactions they receive so that they are processed in a verifiable way. CowSwap [21] implements Frequent Batch Auctions [16] similarly to centralized markets. Trades are batched in discrete time intervals, and transactions in the same batch are executed with the same rates, making it useless to manipulate the order inside the same batch. Rapidash [19] and He-HTLC [74] use Hashed Time-Locked Contracts to enable time-bound transactions, while also incentivizing miners to penalize cheating parties.

When executing a transaction users can set a slippage, which is a value indicating how much the rate of exchange at the time of transaction execution can deviate from when the transaction was issued. Low slippage can increase the chance of transaction failure, while high values enable higher MEV extraction. Heimbach et al. [35] propose an algorithm to compute the optimal slippage value to find the correct compromise.

All these design principles are resilient against transaction reordering, but the adoption of new systems that completely block MEV is slow and actively detracted by users profiting from MEV extraction.

3.4. A Deep Dive into Fair Ordering

This section analyzes in depth the different definitions and implementations of fair-ordering methods. A summary of the described method is presented in Table 3.1.

The first fair-ordering notion was introduced by Kelkar et al. [42]. This work defined the notion of **Receive-order-fairness** as enforcing that if a transaction tx has been received by a fraction γ of nodes before another transaction tx', then all correct nodes will output tx before tx'.

This property was then showed to be impossible because of a consequence of the **Condorcet Paradox** from social choice theory [31]. This conclusion from voting theory shows that it is logically impossible

to have a voted decision to have support from the majority of participants. In the context of distributed consensus over the order of transactions, let's take an example with three parties, A, B, and C. Each of them receives 3 transactions, x, y, and z. A receives them in the order $[x, y, z]$, B in the order $[y, z, x]$ and C in the order $[z, x, y]$. In this case a majority of nodes have received x before y, y before z and z before x, creating a transitive cycle, even when all parties are honest and rational.

This justified the definition of **Block-order fairness**, where if a transaction tx has been received by a fraction γ of nodes before another transaction tx', then all correct nodes will output tx in a block **not after** tx'.

They then developed Aequitas, a protocol that guarantees γ-block-order-fairness in different settings, synchronous/asynchronous and leader/leaderless.

Kursawe developed Wendy [46, 45], a set of blockchain-agnostic protocols that can implement different concepts of fairness. They also showed that the block-order-fairness definition does not guarantee termination as attackers can build an arbitrarily long chain of Condorcet Cycles that should all be inserted in the same block. To achieve termination they define Probabilistic Block Order Fairness, which guarantees that if a transaction tx has been received before tx' by every correct node then tx is scheduled in a block **not after** tx' with a probability of at least p, with p being fixed. A stronger liveness guarantee can be achieved with Timed Order Fairness. Assuming all parties have access to a local clock, which is not necessarily synchronized with other clocks, then if there exist a time t such that tx was seen before t by all honest parties and tx' was seen after t by all honest parties, then tx must be scheduled before tx'. Scheduling a transaction before another still implies that it can be in the same block and as timestamps are included in the block, the ordering of requests inside it can be performed locally after delivery.

Cachin et al. [17] first defined a new notion of Order-Fairness called κ-Differential Order-Fairness. Consider a system with f malicious parties and two transactions tx and tx'. If bm_1, m_2 is the number of correct processes that broadcast m_1 before m_2 than the system is κ-Differentially Order-Fair if $b(tx, tx') > b(tx', tx) + 2f + \kappa$, implies that no correct process delivers tx' before tx.

Kelkar et al. [41] also analyzed order-fairness in the permissionless setting. They proposed two protocols, based on the concept of semantic chains. Each node would be mining and maintaining multiple (d) chains at the same time, with each chain representing a different order and each transaction appearing once in each chain. An existing implementation of an algorithm that can extract a fair ordering from the semantic chains, such as Aequitas [42] can then be employed. This method has constant storage and bandwidth overhead, but latency is increased by a factor of $\frac{5}{2}$ and throughput is decreased by a factor of d.

Zhang et al. defined Byzantine Ordered Consensus as a new primitive that extends Byzantine State Machine Replication to include guarantees about the produced total order [85]. Nodes can associate ordering indicators, such as timestamps, to the events they submit that need to be taken into account when producing a totally ordered sequence. They also propose to clearly separate the process of ordering from consensus, eliminating a leader's ability to fully control the position of commands in the ledger. They propose **Ordering Linearizability** as a fair-ordering property. Through this property, if the highest correct ordering indicator of a transaction t_1 is lower than the smallest correct ordering indicator of a transaction t_2, then t_1 is guaranteed to be ordered before t_2. Pompē is proposed as a protocol that satisfies these properties by collecting $2f+1$ timestamps and using the median value as the assigned timestamp for ordering. This is both upper and lower-bounded by correct values, and so respects Ordering Linearizability. The median value is then broadcast to obtain at least $2f+1$ approvals, which confirms the timestamp value as the one to be used for ordering. The assigned value is approved only if it would not be lower than the one from previously known stable events.

Themis is a scheme for introducing fair ordering of transactions into (permissioned) Byzantine consensus protocols with at most f faulty nodes among $N \geq 4f + 1$ [43]. It uses an order-fairness definition similar to Aequitas [42], but considering only a fraction of honest when checking if enough nodes approve of a specific order. All replicas send their local ordering to a leader, who then proposes an order that should be verified as fair by all other parties. The ordering may produce Condorcet cycles which are handled using deferred ordering. These transactions may be present in enough replicas, and so required to be immediately included, but the leader might not have enough information to order them. A partial ordering is then submitted and completed later, possibly by a different leader, when enough orderings

contain this transaction. This work also showcased how the lack of triangle inequalities in network communications enables attacks on systems such as Themis and Pompē.

Lyra [82] is a protocol that implements Ordering linearizability similarly to Pompē [85], while also introducing a commit-and-reveal scheme based on Verifiable Secret Sharing. Ordering linearizability is achieved by only accepting transactions which timestamp does not exceed the expected timestamp from the sending node by a certain predetermined amount (the security parameter). The expected timestamps can be computed during previous rounds by measuring the distance between nodes. This is achieved by piggybacking the perceived sequence number of a previously received message with the next messages and computing the difference between the actual value and the perceived one. Having obfuscated transactions payload removes the possibility of exploiting triangle inequalities from attackers, but does not protect against the use of transaction metadata [80].

Attackers can retrieve partial information about transactions even without knowing their contents. For example, the transaction sender is usually public so that a fee can be charged to avoid spammers in the network. In some scenarios, knowing transaction contents is also of no value, as blind front-running will still result in transactions getting executed before anyone else.

Gramoli et al. [33] developed the Asynchronous Ordered Atomic Broadcast (AOAB) protocol. The protocol in divided in two phases, ordering and consensus. In the ordering phase transactions are broadcast across nodes together with timestamps. Each node will then assign as the final timestamp of the transaction the median of at least $2f + 1$ received values. During this phase the protocol also makes sure that the transactions are stored in enough nodes, in case they need to be retrieved by other parties. The consensus phase is necessary to decide which transactions need to be included in the output. To achieve optimal communication complexity the computed median value to be used as timestamp is also signed using a threshold signature scheme that gives participants the guarantee that at least one correct node verified its timestamp. Transactions payloads can also be encrypted using a threshold encryption scheme to achieve stronger MEV protection.

Alpos et al. [2] proposed a procedure to generate a blockchain protocol that is resistant to sandwich MEV attacks by modifying any other blockchain protocol used as input. This is done by randomly permuting transactions inside a block using partial seeds that miners of previous blocks need to include to obtain the mining reward. To increase the cardinality of possible permutations, transactions can also be split into smaller transfers that sum to the same final amount. This is not without downsides, as splitting transactions into smaller ones might increase fees, latency and overall network consumption as more metadata is necessary (i.e., signatures). Moreover, this might enable split transactions to be partially executed for some period of time.

SpeedyFair is a protocol created by Mu et al. [55] that decouples ordering and consensus to achieve higher throughput and latency, compared to Themis. This is achieved by introducing a new sub-protocol called Optimistic Fair Ordering that performs the ordering process individually and consecutively. When the current round of fair ordering ends, it is possible to start a new one without having to wait for the consensus phase to end.

Ciampi et al. [20] proposed a TEE-based approach to provide order-fairness through transaction encryption, zero-knowledge proofs and an extension of the ledger. Their protocols can be adapted on top of any PoW Nakamoto Consensus systems. Transaction are encrypted and sent to a subset of validators. Nodes are able to decrypt the transaction only after a certain amount of time (measured in blocks) has passed. The blockchain is extended to include a new type of block, profile blocks, which includes timestamp information together with the encrypted transaction and a zero-knowledge proof. This extra information is called a tag for the transaction. If the majority of consecutive profile blocks contains a tag for a transaction, the transaction is included and its execution time will be the median of the timestamps included in the tags.

FairDAG was recently proposed by Kang et al. [39] and implements ordering linearizability and $\gamma-$batch-order fairness on top of DAG protocols such as DAG-Rider [40]. This is achieved by having all (correct) replicas include any transaction they receive from a client in their blocks, together with assigned ordering indicators. Once a transaction has been committed enough time, it is then possible to extract the fair ordering for execution. FairDAG was released towards the end of this work and its ideas where also explored during the design phase of Tilikum. As they try to achieve the same properties

on top of very similar DAG implementation, an in-depth analysis of FairDAG and comparison with Tilikum is performed in Section 5.6.

3.5. Pompē: Ordering Linearizability

This section summarizes the main contributions from the work of Zhang et al., where they defined Byzantine ordered consensus, a primitive that augments the correctness specification of BFT State Machine Replication to include guarantees on the ordering it produces [85].

First we define the problem of ordered consensus and show how social choice theory can be used to analyze the limitations of the system. The second part describes the Pompē protocol and how it implements Byzantine Ordered Consensus.

3.5.1. Byzantine Ordered Consensus Definition and Impossibility Results

Byzantine Ordered Consensus is a generalization of Byzantine Fault-Tolerant State Machine Replication (BFT SMR) that enables participants to express preferences on how commands need to be ordered. As with normal SMR, it assumes a system of N nodes with up to f faulty (Byzantine) nodes.

When participants propose commands/events to the protocol, they also associate an ordering indicator. This can be any type of value that has some type of *ordered-before* relationship ($\prec$). This means that there is a way to compare two ordering indicators o_1 and o_2, which are respectively the indicators of c_1 and c_2, to be able to say that if $o_1 \prec o_2$ then c_1 will be ordered before c_2. The simplest and most intuitive way to implement ordering indicators is to use timestamps. Each command can be associated a timestamp, for example the time when the command was first issued/seen, and the ordered-before relationship then becomes the less-than operation $<$.

Using results from social choice theory [4, 6, 14] the authors define how Byzantine nodes can influence the final ordering produced by the protocol, what properties are desirable in a fair-ordering protocol and which of these properties are actually achievable. The first concept is Byzantine Oligarchy, which is a property of a protocol that enables Byzantine nodes to order two commands c_1 and c_2 in a specific way (for example c_2 before c_1) even if all correct nodes agree on the opposite ordering (c_1 before c_2). This property is obviously not desirable, and a correct Byzantine Ordered Consensus protocol should not allow the faulty nodes to fully control the ordering, but should still allow every node to influence over the final decision.

This concept is called Free Will by the authors and has two consequences: no commands present in all correct nodes' ledgers can be arbitrarily excluded, and no trivial predetermined ordering can be allowed. The authors then proved that a protocol that upholds Free Will also upholds Byzantine Democracy, which is a property that allows Byzantine nodes to sway the ordering decision only in some specific situations. The next interesting point is trying to minimize the times in which these conditions apply and also to understand exactly in which cases it is impossible to disallow Byzantine influence.

Zhang et al. proposed two ordering properties definitions and analyze their possibility [85]. The first property is *ordering unanimity* that guarantees that if in every correct node's ledger that contains commands c_1 and c_2, c_1 is ordered before c_2, then every correct node agrees on this order.

Unfortunately the authors show that this property is compatible with Free Will because of the existence of Condorcet Cycles [31, 42, 85] and the impossibility of distinguishing faulty nodes from correct ones. It is possible to have situations in which $2f+1$ nodes prefer a specific order, while all the other f nodes disagree, for multiple pairs of commands that then construct a cycle.

The second ordering property is *Ordering linearizability*, inspired by the homonymous concept in Programming Languages theory [37]. In this case, all correct nodes will order command c_1 before command c_2 if all ordering indicators of c_1 in correct nodes are ordered-before all ordering indicators of c_2 from correct nodes. Intuitively, this means that the maximum ordering indicator of c_1 is less ($\prec$) than the minimum indicator of c_2. This property is able to avoid the issue of Condorcet cycles [85] and is what Pompē implements.

3.5.2. Protocol Details

Pompē is a protocol that solves Byzantine Ordered Consensus while upholding Free Will, and so Byzantine Democracy. The protocol is divided in two phases: an ordering phase and a consensus phase.

During the ordering phase, the node that proposed command c collects at least $2f+1$ signed timestamps (including its own) from other participants. The node will then compute the median t of these $2f+1$ values and assign this value as the timestamp of c. As the maximum number of faulty nodes is f, the median must be both upper- and lower-bounded by correct timestamps, which makes this value usable for ordering linearizability [85]. To lock the command c at the position of the assigned timestamp the node needs to have this value approved by at least $2f+1$ participants, nodes accept messages based on some conditions explained later.

Each node also maintains the following data structures: the highest timestamp they believe to be stable in the ledger `localAcceptThresholdTS`; the set of all locally accepted commands `localSequencedSet`; a vector `highTS` containing the highest timestamp received from every node (so `highTS[i]` is the highest timestamp of node i); and `highTSMsgs`, which contains the message containing the previously mentioned highest timestamp.

Nodes also have a way to update and synchronize their local timer, which involves two extra variables: `globalSyncTS`, the highest timestamp received in a synchronization message and `localSyncTS` which was the local timestamp at the time the message was received. Each node considers their $(f+1)$-th highest timestamp from `highTS` (from now called T) to be the minimum value their timestamp needs to be, so every time this value becomes higher than their local time they update the latter to be T. This value can also be periodically transmitted to signal other processes that this time is safe to move forward to.

When a node wants a command to be sequenced it first sends the command with a request for timestamps. After $2f+1$ values have been collected, the median is computed and a new message is broadcast requesting the assigned timestamp t (median) to be accepted. Other nodes accept the value if $t >$ `localAcceptThresholdTS`. An accepted command is acknowledged and added to `localSequencedSet`. A rejected command is not added to the set and an adequate response is sent.

After commands are accepted, which means that their timestamp is locked and ready to be ordered in the ledger, the consensus phase can begin. The main objective of this phase is to agree on stable prefixes of commands that will not need to be reordered. A consensus algorithm is used to agree on which commands need to be included for each time interval, also called time slots or consensus slots. If a node wants to act as the leader for a specific consensus instance over slot k that is bound between timestamps ts and ts', it starts by broadcasting such request and waiting for $2f+1$ approvals. Other nodes will approve this request if $ts' <$ `globalSyncTS` and the current timestamp is sufficiently larger than `localSyncTS`. If a request is ready to be approved, a node that received it will update its own `localAcceptThresholdTS` to max(ts', `localAcceptThresholdTS`) and then respond by sending all commands in `localSequencedSet` that belong to the current time slot.

The now approved leader is able to run a consensus algorithm over the union of all received sets of commands. As enough nodes will have sent their local commands to be sequenced, and subsequently updated their `localAcceptThresholdTS`, these transactions are now stable and can be safely ordered by sorting them over their timestamps, breaking ties deterministically (such as using hashes), and removing possible duplicates.

4

Tilikum: Adding Order Fairness to DAG-based Consensus

This chapter explains the necessary modifications to implement Tilikum[1], a protocol upholding ordering linearizability, like Pompē [85], on top of Narhwal and Tusk [23].

The first section proposes the system and threat model used for analysis and implementation. The second section explains the necessary modifications to prevent censorship of client's transaction by malicious nodes. The third and fourth sections are interested in how nodes can add timestamps to transactions, collect enough of them from other participants, and verify their validity. Section five describes how to work around the lack of full validity in Narwhal and Tusk. Section six summarizes the design of the protocol. Section seven presents and proves the properties that Tilikum upholds and analyzes the message complexity. The conclusion of this chapter discusses the design choices made and the limitations of Tilikum.

4.1. System Model

As the system is based on top of Narwhal and Pompē, our system and threat models are similar to theirs. We assume N nodes with a maximum of f faulty nodes, with $f < N/3$. Faulty nodes are also called malicious or corrupted and are Byzantine, which means that they can deviate from the protocol in any way. Malicious nodes are also able to collude, share information, and coordinate to perform more effective attacks.

We also assume eventually reliable communication links. Messages can be delayed indefinitely, but the number of lost messages is finite.

Each node also has access to some cryptographic primitives. In particular, collision-resistant hashes and unforgeable digital signatures. Each node has a unique private key that allows them to sign messages, and every node knows the public key of every other node to be able to verify signatures.

4.2. Solving Censorship in Narwhal

To achieve order fairness, it is also important to solve possible censorship issues. Indeed, we can see censorship as a special case of re-ordering. A malicious node may (indefinitely) delay specific transactions, which enables front-running. We then ask, **to how many nodes should clients send transactions to?**

The authors of Narwhal and Tusk discuss this limitation in their paper and come to the conclusion that to guarantee censorship resistance clients should send their transactions to $f + 1$ nodes. They also discuss the impracticality of this solution, as it would divide the bandwidth by an $O(n)$ factor [23]. A

[1]Keeping the tradition of calling protocols like whales, Tilikum was a male orca kept captive at SeaWorld, Orlando. He was featured in a documentary that shed light on the unfair living conditions of orcas in captivity.

proposed alternative is to send each transaction to a fixed number k of nodes, which exponentially increases the chances of finding a correct replica that will include the transaction in their next block.

While this functions in practice, and also does not reduce the throughput too much, it does not result in the best conditions for order fairness. Let us assume that the transaction is sent to $f + 1$ parties and each of them records the timestamp at which the transaction was received. As only one of these timestamps is guaranteed to come from a correct node, we can only obtain at best an upper or a lower bound on the correct timestamp values (using respectively the maximum or minimum value). This is not enough, as malicious nodes can still front-run in case we extract an upper bound or delay transactions in case we extract a lower bound. Pompē does need $2f+1$ timestamps to extract a meaningful timestamp. With this amount of timestamps, the median is both upper- and lower-bounded by correct values.

We then require transactions to be seen by at least $2f+1$ nodes, such that they are meaningfully assigned a timestamp. This does not necessarily mean that a client needs to send the transaction to at least $2f+1$ nodes. When nodes need to collect timestamps, to then compute the median as the assigned indicator, they also broadcast the transaction itself. Nodes receiving this timestamp request will also receive the transaction itself and can respond with the current time.

Having transactions being sent to everyone would, without additional modifications to the original protocol, result in redundancy in the DAG. Each (correct) node will try to include the transaction in a batch as soon as it sees, which would reduce the total processing capacity of the network. The next design step is to decide **how many nodes should include a transaction.**

We can employ a mechanism to assign transactions to some specific nodes. Each node knows in advance how many replicas are participating in the system and can then compute a hash value of the transaction, map it to the set of replica IDs and include the transaction in its own batch only if the resulting ID is in the assigned set. Note that this could lead to transactions assigned to faulty replicas never being included in the ledger.

We can then follow one of two approaches, either **we assign the transaction to a set of at least $f + 1$ nodes**, which will guarantee that at least one correct node will include it, or we employ some **mechanism for correct nodes to take over in case the transaction is not included.**

Correct nodes should store transactions that were not assigned to themselves and include them in a batch after some time has passed without those transactions being committed. To decrease the chances of transactions being unnecessarily included, this timeout value can be dynamically adjusted. Both strategies are valid and were implemented for evaluation.

4.3. Adding Timestamps to Batches

As previously established, it is necessary to collect $2f+1$ timestamps for each transaction so that these values can be agreed on during consensus. There are two strategies that we can employ, either **nodes bundle each transaction with $2f+1$ timestamps for it** or **all nodes include all transactions together with one timestamp per transaction**. We already discussed in Section 4.2 why we do not need full redundancy to solve censorship, and we describe in more detail why the second strategy is not ideal in Section 5.6.2. Bundling enough timestamps with every transaction in a block allows every transaction that appears in the DAG to be successfully executed.

We then require nodes that are proposing a batch of transactions to collect $2f+1$ timestamps per transaction. After collecting the timestamps, a correct node can compute their median and use it as the assigned timestamp of the transaction. This value also needs $2f+1$ approvals, which means that the timestamp respects some properties, in particular that it would not require re-ordering of already committed transactions.

The practical implementation of Narwhal involves workers exchanging batches to reliably store them between each other. This requires the batch to be broadcast and for $2f+1$ acknowledgments to be collected. This gives the guarantee that at least $f+1$ nodes stored the batch for later retrieval.

In the node implementation, each worker communicates with the workers with which it shares the ID and that are under a different primary. For example, in a system with 5 primaries and 4 workers per primary, worker 0 of primary 0 will communicate with all other workers 0 from primaries 1 to 4.

Workers exchange batches to reliably store them in at least $2f+1$ databases. This also removes the need to store full transactions in blocks and only requires the exchange of batches' references.

The current worker architecture (Figure 2.1) manages these steps in three components:

- The Batch Maker creates batches from transactions waiting in its internal queue;
- The Quorum Waiter receives batches from the Batch Maker and sends them to all other workers, waiting for $2f$ acknowledgment before relaying the batch to the Processor;
- The Processor stores batches, either its own ones that have been acknowledged enough times or other's batches for reliable storage.

To introduce timestamp sharing, a new step is necessary. After a batch is sealed by the Batch Maker, but before it is broadcast so that the Quorum Waiter can wait for acknowledgments for its storage, another round of broadcast is necessary. The worker will send a Timestamp Request to the corresponding workers of other nodes. The response to receiving this kind of message should include a signed vector of timestamps, one for each transaction in the batch. These timestamps indicate when each transaction was first seen by this worker, which may be the time of receipt of the Timestamp Request message, as the full transaction is sent at that time.

The new operations are handled by the previously described components and by a newly added one, the **Timestamp Collector**. After collecting $2f+1$ responses, the batch can be broadcast again with the purpose of being stored. From now on the program continues normally, with batches being reliably stored and then sent to the Primary node for inclusion in a block.

Another important detail is making sure that a primary only accepts blocks for which batches contain enough correctly signed timestamps for each transaction. This is achieved by adding a new condition to the four already existing ones. A valid block then must:

1. contain a valid signature from its creator;
2. be at the local round r of the validator checking it;
3. be at round 0 (genesis), or contain certificates for at least $2f+1$ blocks of round $r-1$;
4. be the first one received from the creator for round r;
5. **NEW:** contain $2f+1$ signed timestamp vectors of the correct length for each batch in the block.

To prevent attacks where signed vectors are re-used by malicious parties, the signature should sign the content of the batch as well, i.e., the signed digest is of timestamps and transactions.

Through these modifications, the Narwhal mempool is now populated by blocks containing batches of transactions with at least $2f+1$ signed timestamps per transaction. This is enough to calculate a median that is both upper- and lower-bound by correct values, just as Pompē [85]. We call the median of these values the **assigned timestamp** of the transaction. This implies that transactions ordered by these timestamps satisfy ordering linearizability.

The next step in the algorithm needs to order these transactions securely, in a way that does not lead to late transactions with old timestamps having to be discarded or forcing re-ordering of already executed ones.

4.4. Ordering Transactions based on Timestamps

Narwhal is still only a mempool abstraction, and even if it now contains enough information to order transactions, it is still necessary to reach a consensus on a block that validators can see and retrieve with its causal history.

Narwhal can be natively augmented to include information to build a perfect random coin and yield Tusk, an asynchronous consensus algorithm. This algorithm takes the local view of the DAG constructed by Narwhal and totally orders its blocks consistently. This order does not respect the transaction timestamps, and so transactions' execution needs to be delayed until it is safe to do.

Pompē normally implements this by only approving transactions that are more recent than a specific threshold, which is updated after every consensus round. Consensus is executed in time slots, so in

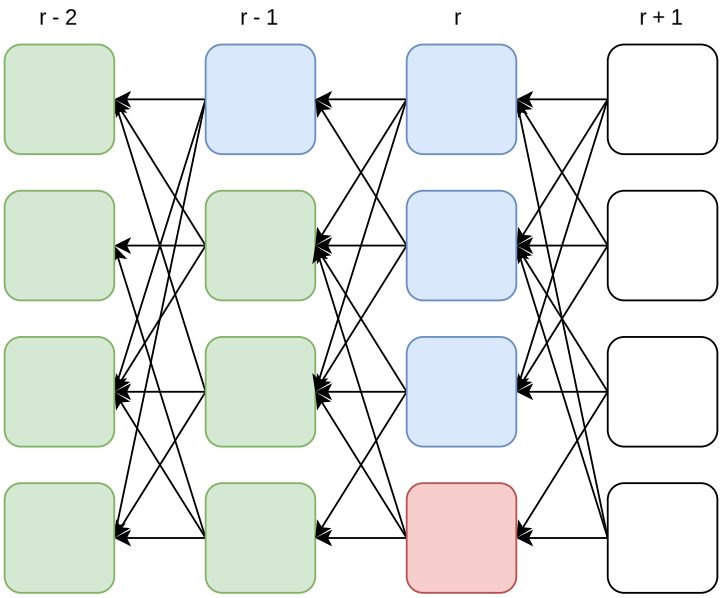


Figure 4.1: Example of consensus wave, the red block is the elected leader, which will be committed together with the green blocks. The blue blocks will not be committed yet.

each round transactions between time ts and ts' are agreed upon and then executed. The threshold is then updated to ts' (or kept if it was already higher) [85].

This is not desirable, as latency would then depend on an external factor (the size of the consensus slots) and does not only depend on the status of the network. We then propose a new method to order transactions safely that does not require the discretization of time slots.

4.5. Introducing Logical Time

When transactions need to be executed in the order expressed by their assigned timestamps, it is necessary to make sure that later in time no transaction that was supposed to be executed earlier can still arrive. Given how transactions are added and then committed in the DAG, there is no guarantee that this will be the case.

Let us take a simple example. In each consensus round, one leader is chosen from all the blocks of round r. The other blocks in round r will be committed **at least** in the next run of the consensus algorithm. This situation is shown in Figure 4.1, the red block is the leader block, green blocks will be committed, while blue nodes will be committed at least in the next wave. Blocks of correct nodes in the same round are likely to include timestamps of similar values. This means that in the following rounds of consensus, nodes will commit blocks that have timestamps that might be similar to what was already previously committed, including lower ones.

While the example above simplifies some behaviors, the nature of the DAG makes us commit sub-graphs that do not guarantee that blocks in the same rounds (which are then likely to have close timestamps) are committed at the same time.

We can visualize a similar scenario using Figure 4.2. In this case, we have 4 replicas that have seen 4 transactions, T_a, T_b, T_c and T_d. The committed sets of $2f+1 = 3$ timestamps for T_a, T_B and, T_c are highlighted in different colors (green, blue, purple). We can also notice that there is a valid set of 3 timestamps (highlighted in red) that could belong to T_d, but are not yet committed. A future appearance of T_d might result in an assigned timestamp lower than one of T_a, T_b or T_c, which, if executed immediately, would result in invalid ordering.

What is necessary from each node's point of view is to know a safe time threshold until which to safely execute transactions. The first necessary information to compute such a threshold is whether there exist transactions that have been assigned a timestamp, but have not been committed yet.

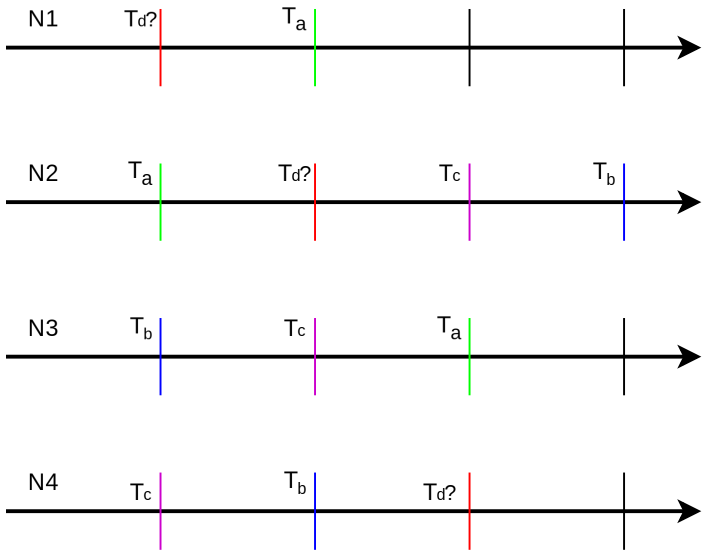


Figure 4.2: Example scenario where unseen timestamps (shown in red) could possibly belong to a valid future transaction T_d.

We can obtain this knowledge by changing the format of timestamps. When requesting one from other nodes, the response must include not only the current UNIX clock value, but also what we call a **Logical Clock** value. This is a simple monotonically increasing counter that represents the order in which transactions have been seen by a particular replica (combination of a primary and its workers). This new data structure that contains both a UNIX timestamp and the corresponding Logical Counter is called a **Timestamp Pair**.

When a specific transaction in a specific block and its $2f+1$ signed Timestamp Pairs are committed, nodes can then update for each replica that participated in the set of $2f+1$ timestamps which logical times have been committed. These values can be stored in a new data structure, abstracted as a map of lists, which we call the **Logical Table**.

Each value in the map, indexed by replica's identified (Public Keys), is a list of all seen Timestamp Pairs, ordered by their logical value. The main responsibility of the Logical Table is to track the missing timestamp pairs of each replica. We call such missing values **holes**. Each row of the table is a sorted list of Timestamp Pairs, with values for which nodes have received **all** previous pairs and their immediate successor removed. An example of a Logical Table is shown in Table 4.1. We can see a row for each replica with a hole immediately following the head value, plus some other pairs later.

Table 4.1: Example of Logical Table.

ID						
0	(4, 23)	(6, 28)	(7, 35)			(10, 54)
1	(5, 28)	(7, 39)	(8, 40)	(9, 47)		(11, 59)
2	(3, 22)	(6, 29)		(7, 33)	(8, 39)	(9, 44)
3	(7, 42)		(10, 60)	(11, 63)		(13, 70)

If nodes collect, for each replica, the smallest pair that does not miss any precedent value (i.e., the head of the list), they obtain a list of $3f+1$ Timestamp pairs that can always be executed. This is because, for each replica, nodes have also committed all transactions seen by that replica with a smaller timestamp than the one in the list.

If nodes take the $2f+1$ smallest associated UNIX timestamps and compute their median, they obtain a value that can be used as a threshold up until which execution is safe.

This value has some important properties:

1. If correct nodes are progressing and adding new blocks, then this value is also always increasing;

2. Every correct node will compute the same value, as they already agreed on a sequence of committed blocks;
3. This value will never be larger than the assigned timestamp of an uncommitted transaction;
4. Faulty nodes have limited influence on this value;
5. It is a fair representation of the point in time until which correct nodes had their values committed;

All of these properties are ideal for such a threshold as it gives correct nodes the guarantees necessary to execute transactions.

4.5.1. Challenges Related to the Execution Threshold

It is possible that specific timestamp pairs coming from specific replicas will never be inserted in the DAG, leaving a permanent hole in the logical table. This can happen for two reasons, either the replica's proposed timestamp was not included in the list of $2f+1$ values bundled with the transaction, or the block containing the timestamp pair was not referenced by enough other blocks in the DAG.

As Narwhal and Tusk do not implement weak edges [23], contrary to DAG-Rider [40], there is no guarantee that every certified block is eventually committed. This is because a block might be approved and be eligible for a certificate of availability, but the certificate might be received too late by other replicas and not be referenced by blocks of the successive round.

The authors of Narwhal and Tusk suggest that replicas re-inject batches of blocks that will not be committed in future blocks. This is still not a strong guarantee, but in expectation every batch will eventually be included.

The case where a specific timestamp pair was not selected in the set of $2f+1$ would still lead to permanently missing values in the Logical Table. To mitigate this issue, we introduce the concept of the **Hole Filler**, which is extra metadata added to blocks by replicas to make others catch-up on lost values that should fill the table.

Whenever a primary receives a request for a Timestamp Pair for a transaction, they should store this transaction-time pair for later use. After a block has been committed by the Consensus module, the primary should verify for which transactions in the blocks their assigned Timestamp Pair was **not** included. All these values should be retrieved from the local storage and included in their own row in the Logical Table. Nodes can then add as metadata in the next block the current head of the just updated row.

When processing a committed block, the Analyzer can safely advance the logical table row of the block's author such that the new head is equivalent to the hole filler's value. In case the row is already ahead enough, the value can be simply ignored.

4.6. Putting it All Together

Let us summarize our design decisions and how the protocol works.

We assume that clients that want their transaction to be (fairly) included in a block will send it to every replica, which means to one worker per node. For inclusion in the DAG, it is enough to send the transaction to the assigned nodes, but to obtain the lowest latency the correct behavior is assumed.

Keeping the Narwhal worker-primary architecture, workers still only operate on batches of transactions, while the primary operations are done on blocks. When a primary needs to process a batch to extract the transactions for ordering and execution, it requests it from the worker that stored it. This can also be achieved by having workers preemptively send batches to the primary when they are ready for inclusion.

Workers now have an extra responsibility: collecting timestamps for transactions, while primaries assign these timestamps when requested from their workers. To reduce communications, every time a worker assigns a Timestamp Pair to a transaction, it stores this association for future retrieval without having to go through their primary. Workers also need to respond to timestamp requests from other workers, which might require them to ask for a new Timestamp Pair from their primary. The last new task of

workers is to keep track of transactions that they have received, but not included in a batch. They will be inserted into a future batch once a specific timeout has passed.

Timestamp collection occurs **before** reliable storage of batches and a batch **cannot** be considered valid unless it has enough signed Timestamp Pairs for each transaction. Once enough values have been collected, the batch can be broadcast for reliable storage, and after enough workers have acknowledged its reception, it can be sent to the primary for inclusion in a block.

These changes are shown in Figure 4.3.

Primaries always trust their workers regarding the contents of batches, but always need to make sure that foreign ones adhere to the same properties. As explained in section 4.3, this can be enforced when approving blocks proposed by other primaries. We add to the already necessary conditions that all included batches include enough signed timestamps for all transactions.

These changes are shown in Figure 4.4. As visible in Figure 4.4, another extra component is necessary, as consensus on blocks in the DAG does not imply immediate execution of transactions. This new **Analyzer** component is responsible for ordering transactions, and only executing them up until a specific threshold. This is necessary to reduce the chances of having to discard transactions if it is committed too late from the DAG.

In this component, nodes maintain the following data structures:

- `sequencedSet`: the set of fully sequenced and executed transactions;
- `threshold`: The current threshold up until it is safe to execute
- `queue`: A priority queue of transactions that are waiting to be executed, sorted by lowest median timestamp first;
- `logical_table`: a Logical Table implementation that stores timestamp pairs for each replica, with the requirements from section 4.5;

The first is used to de-duplicate already executed transactions. Each transaction could appear multiple times, as we need to avoid censorship.

The queue is polled every time the threshold changes, transactions are popped if their timestamp is lower than the maximum threshold and executed. After execution, they are added to the set of sequenced transactions and the maximum executed timestamp is updated.

The logical table and the threshold work together and require some care. Every time a new wave of blocks is committed, all the Timestamp Pairs for all transactions are added to the logical table. The logical table stores a map of **Logical Vectors**, each indexed by the name (Public key) of other replicas.

A Logical Vector is a data structure that stores Timestamp Pairs sorted by their logical clock. Whenever a set of new pairs is inserted, it removes the ones that are contiguously present from the beginning. For example, if logical clocks $0, 1, 2, 3, 5, 6, 8$ are inserted, the logical vector can safely remove the first 3 values and only store $3, 5, 6, 8$. This is necessary to implement the logic that checks which timestamps were not seen from each replica, to later compute the threshold.

With an updated table of Logical Vectors, after a whole wave of blocks has been added to the map, nodes can query the smallest UNIX timestamp for each replica, compute the median of the smallest $2f+1$ and update the threshold to this value.

4.7. Enforced Properties and Complexity Analysis

Tilikum maintains the security guarantees from Narwhal and is able to uphold the same properties in expectation, in addition to the new ones related to fair ordering.

4.7.1. Maintained Properties

This subsection shows the definition of the properties originally guaranteed by Narwhal and Tusk. We also prove that these properties are enforced in Tilikum. The first part talks about DAG (Narwhal) specific properties, while the second one analyzes the consensus (Tusk) ones. These mostly follow from the original ones from Danezis et al. [23].

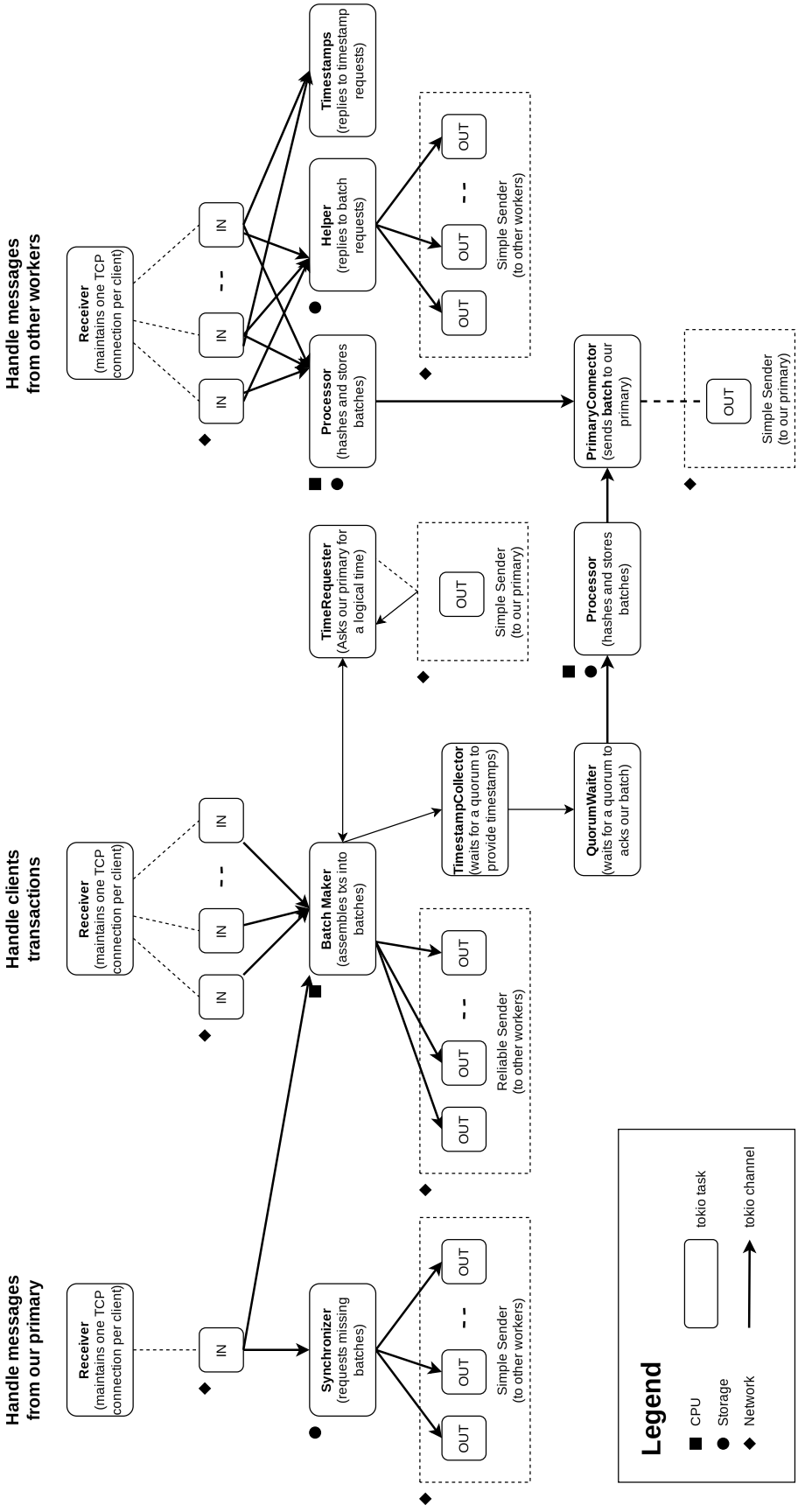


Figure 4.3: The new architecture of a Worker

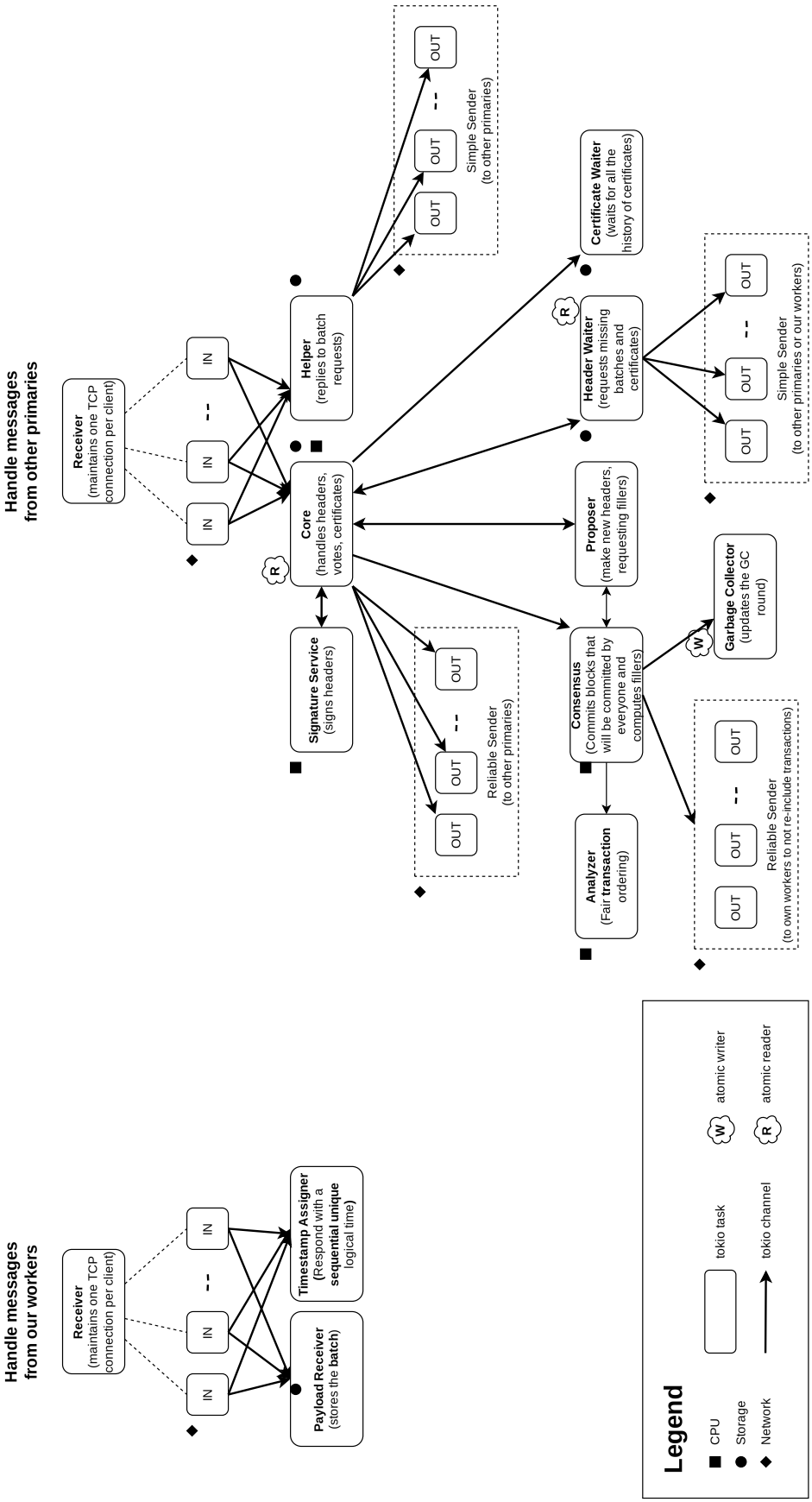


Figure 4.4: The new architecture of a Primary

DAG-Related Properties

The DAG structure in unchanged in Tilikum, and so the following properties are fully respected:

Integrity The $read(d)$ operation returns the same block for all replicas. This is a consequence of collision-resistant hashing. No two blocks will have the same digest and so if $read$ succeeds, it will return the correct block.

Block-Availability If $write(d, b)$ succeeded for an honest party, then $read(d)$ returns b for every honest party. A block is certified if at least $2f+1$ parties sign its storage, which means that at least $f+1$ parties stored the block. When a read operation is performed, the caller waits for $2f+1$ responses, which means that at least one of these will come from an honest party that stored b.

Containment If a block b' is part of the causal history of block b, then all the causal history of b' is also in the causal history of b. This property comes as a consequence of honest nodes not signing more than one block per round per author. As $2f+1$ signatures are required, it is impossible to have multiple blocks from the same author in a round. Every certified block stored in a correct replica will have the same set of references to blocks in the previous rounds. We can recursively apply this reasoning and show the property.

2/3**-Causality** For each block b, its causal history ($read\_causal(d, b)$) will contain at least $\frac{2}{3}$ of the blocks proposed before $write(d, b)$ was called. In each round at most $N = 3f+1$ blocks are correctly written. Each block in the consecutive round has to refer to at least $2f+1$ blocks from the previous one, which is more than $\frac{2}{3}$ of the total.

1/2**-Chain Quality** At least half of the blocks in the set returned by $read\_causal(d)$, were written by honest parties. As each block refers to $2f+1$ blocks from a previous round, at least $f+1$ (half) will be from correct nodes.

4.7.2. Asynchronous Consensus Properties

Properties of the consensus layer (Tusk) are shown and proved here. These mostly follow from the original ones from Danezis et al. [23].

Safety

Any two honest validators will commit the same sequence of blocks during the consensus step.

Lemma 4.7.1. *If an honest validator commits block b as leader in wave i, then any leader block b' committed by honest validators in waves after i has a path to b.*

Proof. Honest validators commit block b in wave i only if it has $f+1$ nodes in the wave with paths to b. All blocks in the first round of wave $i+1$ have $2f+1$ paths to previous blocks. By Quorum intersection, at least one of the paths of the leader block of the wave starting from this round will go to b. By induction, we can show that every block in every round after wave i has a path to b. □

Lemma 4.7.2. *If b is the leader block of wave i and b' is the leader block of wave i', if an honest validator commits b before b', then no other honest validator commits b' without committing first b.*

Proof. If an honest validator committed b before b', then there is no path from b to b'. Assume by absurd that a different validator committed b' before b there would also be no path between b' and b. But by Lemma 4.7.1 at least one of these paths must exist, making a contradiction. □

The safety property comes as a direct consequence of Lemma 4.7.2.

Liveness
The liveness property informally says that something is always happening, so that the system is always progressing towards new correct events. In Tusk, we want to show that we expect new blocks to be committed in a finite amount of time. This liveness property is enforced **only for the consensus step** and not for the ordering phase. That concept of liveness is defined and proven later.
Lemma 4.7.3. *In every wave w there are at least $f+1$ blocks in the first round that can be committed.*

Proof. Let's consider any set S of blocks in the **second** round of wave w. There are then at least $(2f+1)^2$ connections to blocks in the first round. There also at most $3f+1$ blocks in the first round. For each of these blocks to be not allowed for commit, they need to have at most f connections to blocks in the next round. Assuming the worst case where every block in the first round has exactly f connections to blocks in S, the remaining number of links is still $(2f+1)^2 - f(3f+1) = f^2 + 3f + 1$. Each block in S has at most $2f+1$ connections to block in the first round. This means there are at least $\frac{f^2+3f+1}{2f+1-f}$ blocks with more than $f+1$ connections to the next one. □

Lemma 4.7.4. *In expectation, a leader is committed every 7 rounds in a network with asynchronous adversary.*

Proof. By Lemma 4.7.3, there are at least $f+1$ committable blocks for each new instance of consensus. As the random coin is unpredictable, there is a $\frac{1}{3}$ chance of committing a valid block. As consensus is run every 3 rounds, this gives an expected value of 9 rounds. However, the last round of each consensus wave is the same as the first round of the next wave, giving us an expectation of 7. □

4.7.3. Fairness Properties
This subsection shows which fairness properties are fully respected by Tilikum. These are the ordering property, **Ordering linearizability**, and the concepts of **Execution Safety** and **Execution Liveness**.

Fair Ordering through Linearizability
As Tilikum uses the same ordering property and mechanism as Pompē [85], their proofs look relatively similar.
Lemma 4.7.5. *The median of $2f+1$ values, of which up to f can be malicious, is always both upper- and lower-bound by correct values.*

Proof. For a set of $2f+1$ values, we can see the median as the values sandwiched between f smaller (or equal) values and f higher (or equal) values. The f malicious values can be spread in three ways: all in the smaller set, all in the higher set, or spread between them. In the first 2 cases, we have that the median itself is a correct value and that the other set is fully composed of correct values, bounding the median between honest numbers. In the last case, we have that each set contains at least one correct element, which is smaller or equal for the left-side and higher or equal on the right-side, again bounding the median between these two correct entries. □

Let us recall the definition of **Ordering linearizability** in the context of asynchronous consensus over transactions:
Definition 4.7.1. Let t_1 and t_2 be proposed transactions. If the highest timestamp provided by a correct node for t_1 is lower than the smallest timestamp provided by a correct node for t_2, then all correct nodes will order (execute) t_1 before t_2.
Lemma 4.7.6. *Transactions ordered by the median of $2f+1$ collected timestamps respect ordering linearizability.*

Proof. Let us take two transactions tx_1 and tx_2 with the respective median timestamps t_1 and t_2. If the maximum timestamp from a correct node for tx_1 is tm_1 and the minimum timestamp from a correct node for tx_2 is tm_2, we know that $t_1 <= tm_1$ and $t_2 >= tm_2$. Hence, if $tm_1 < tm_2$, the first condition of Definition 4.7.1, we know that $t_1 \leq tm_1 < tm_2 \leq t_2$ and so $t_1 < t_2$, which will make correct nodes order, and so execute, tx_1 before tx_2. □

Execution Safety
We will now prove that the threshold computed through the Logical Table is always safe, which means that the threshold will always be lower than the assigned timestamp of any transaction that may still possibly be committed.
Lemma 4.7.7. *The assigned timestamp t of a transaction tx is not greater than the threshold value that would be calculated if tx is the first hole for every replica.*

Proof. The assigned timestamp t is the median of the timestamp assigned to tx by $2f+1$ replicas. The threshold computed by the Logical Table in case this transaction is the smallest hole for all replicas is the median of the smallest $2f+1$ timestamps assigned by replicas. This value cannot then be bigger than t. □

Theorem 4.7.8. *Taking an arbitrary uncommitted transaction tx that will be inserted in the DAG with assigned timestamp t, the threshold tr cannot be larger than t.*

Proof. An uncommitted transaction will always result in a hole for every entry in the Logical Table. According to Lemma 4.7.7, the threshold resulting from the situation where this is the smallest hole for each replica, results in the threshold being valid for future execution of this transaction. In case this transaction is not the first hole for some or all replicas, we know that the threshold will then be committed over smaller holes, which always have smaller or equal timestamps, as replicas increase their timestamps monotonically. □

Theorem 4.7.3 shows that it is impossible for the threshold to be larger than the timestamp of an uncommitted transaction, showing Execution Safety.

Execution Liveness
We now introduce the concept of **Execution Liveness**. Remember that transactions are first committed through a consensus mechanism (Tusk) and execution is delayed until the `threshold` value is high enough for safe execution.

This section shows that on average and with random network latencies/delays, the threshold is always increasing and so some transaction is always eventually executed.
Lemma 4.7.9. *Assuming random network latencies and that every (correct) replica proposes one block per round, each proposed block has a $\frac{2}{3}$ probability of getting approved in the current round.*

Proof. Having random network delays implies that the order a replica receives blocks is random. If $3f+1$ blocks are submitted each round and each replica waits for $2f+1$ certificates before progressing, then from the point of view of each replica $\frac{2f+1}{3f+1} > \frac{2}{3}$ blocks will be included in the current round. □

Lemma 4.7.10. *The expected value of committed blocks per replica and per wave of consensus is 3.*

Proof. From Lemma 4.7.4 we know that on average consensus happens every 7 rounds. From Lemma 4.7.9 we get that $7 \cdot \frac{2}{3}$ blocks per author are in a wave. Moreover, by 2/3-Causality $\frac{2}{3}$ of these blocks will be committed, giving us an expectation of $7 \cdot (\frac{2}{3})^2 \approx 3.11$. □

Lemma 4.7.11. *Every time a leader block is committed, logical timestamps from at least $f+1$ correct replicas are updated.*

Proof. If a leader block is successfully committed, then from 1/2-Chain Quality we know that at least of the blocks in the causal history are from correct authors. These blocks will include timestamps from $2f+1$ unique sources and so at least $f+1$ will be correct. □

Lemma 4.7.12. *In expectation, each round of consensus will update the local logical clock of each correct replica.*

Proof. As in expectation 3 blocks per replica will be committed and these blocks will definitely contain at least 1 timestamp pair from its author, then their entry in the logical table will be updated. □

Lemma 4.7.12 and Lemma 4.7.11 gives us that on average, the logical table of all correct processes, with a minimum of $f+1$ per wave, will be updated.

These two together can prove the final theorem that guarantees execution liveness:
Theorem 4.7.13. *In expectation, the threshold computed locally by each replica is always increasing, and new transactions are always being executed.*

Proof. By the safety property, each correct replica will commit the same sequence of blocks and so have the same internal status for the Logical Table. The threshold is updated by taking the median value of the $2f+1$ smallest timestamps for which we miss the successive value in the logical table. From lemma 4.7.5, this value is both lower- and upper-bound by correct values and if all correct values are getting updated, then the threshold also gets updated. As timestamps cannot decrease, the threshold is always increasing. □

This guarantees Execution Liveness in the average case. We can also show that it holds in the general case, but in unbounded time, by exploiting the fact that every block in the DAG will eventually be committed [23]. If every block will eventually be committed and every correct replica will try to include in their blocks every transaction they have been assigned together with hole fillers, then every value in the Logical Table will eventually be seen, and the threshold will always increase.

This second way of looking at Execution Liveness gives us the guarantee that the threshold will eventually catch up, but does not allow us to perform any theoretical analysis on the additional latency.

4.8. Design Decisions & Limitations

This section discusses the rationale behind some design decisions and their limitations. In particular, we focus on the decision not to use weak edges to enable garbage collection (like in Narwhal and Tusk [23]) and to include $2f+1$ timestamps for each transaction in a block.

4.8.1. Weak Edges

DAG-Rider [40], the theoretical basis for Narwhal and Tusk [23] and FairDAG [39], builds the DAG using two types of edges: weak and strong. Strong edges have the same properties as edges in Narwhal and Tilikum, they connect blocks in round r to blocks in round $r-1$ and at least $N-f = 2f+1$ need to be present for each block.

Weak edges allow a block at round r to refer to blocks to any round $r' < r-1$, with the restriction that at most f such edges are present in each block. These extra links give the theoretical guarantee that every proposed block will eventually be committed, as older blocks received by correct replicas will be referred to by weak links. However, there are some major limitations coming from the usage of weak edges.

As described by Danezis et al. [23], the existence of weak edges forbids garbage collection. Every block ever received needs to be stored indefinitely as it could be pointed by a weak link, making this implementation impossible with finite storage and memory.

Moreover, Mahe et al. [49] pointed out a mistake in the original algorithm specification for DAG-Rider, which may result in the misuse of weak edges. The creation of a vertex does not depend on the reliable delivery of the previous one, and so it is possible for a node to create vertices with no references with one another, so only referring to blocks from different replicas. When all these disconnected vertices are delivered by other replicas in future rounds they will all be connected with weak edges, possibly violating the constraints of maximum f of such links.

Not including weak edges in the design of Tilikum allows us to maintain garbage collection, which is a requirement for real deployed systems. However, this comes at the cost of possibly not committing some correctly submitted batches, including their bundled timestamps, and hole fillers.

Danezis et al. [23] argue that retransmission of the content of garbage collected blocks (batches and hole fillers) will give in expectation the same guarantees as weak edges. We noticed that this logic was not

implemented in the research implementation of Narwhal and Tusk, [2] but exists in the production-ready Sui code[3]. This feature was backported into Tilikum.

This solution seems to perform well in practice when considering a system like Sui, but in situations like ours where we need to see information from (almost) all replicas to progress it might not be enough. As shown in the Evaluation chapter (Chapter 5) there seem to be still be a consistent amount of undelivered timestamps and hole fillers which slow down the execution threshold from growing. We believe this is future work as it does not interest the main protocol structure, but some practical design decision for the implementation.

4.8.2. Timestamp Inclusion

Inside Tilikum's batches, $2f+1$ timestamp pairs for each transaction are included. This allows every transaction that is committed to be eventually executed, as no extra information is required for this to happen. However, this has two main downsides.

First, the size of batches increases. A batch now needs to include the timestamp pairs and signatures from other nodes to verify their origin. Second, only $2f+1$ pairs can be included, as up to f faulty nodes may not respond to the request. This means that the timestamp assigned by up to f replicas may remain unknown. This is mitigated by using hole fillers.

The inclusion of hole fillers also suffers from the same issues described in Subsection 4.8.1. Blocks including hole fillers may be received by enough nodes, and so its author will build a valid certificate, but not be referred by any node in its next round, effectively disconnecting it from consensus.

[2] https://github.com/asonnino/narwhal/
[3] https://github.com/MystenLabs/sui/blob/narwhal-votes/narwhal

5

Evaluation

In this chapter we evaluate the performance of Tilikum. We are interested in several metrics, divided into two main categories: performance metrics (i.e., throughput and latency) and security metrics such as reordering attack success rate.

The first section explains the experimental setup, including parameters, hardware and measured metrics. We then run some basic experiments to choose which of the censorship prevention methods highlighted in Section 4.2 is better. We compare throughput and latency against Pompē [85] and FairDAG [39]. We measure the attack success rate when using the attacks from the work of Zhang et al. [83] on Narwhal and Tusk and on Tilikum.

FairDAG's code[1] was released towards the end of the evaluation of this work. We adapted its code for a fair comparison against Tilikum. Some changes were necessary and are described at the end of the chapter together with a theoretical comparison of Tilikum and FairDAG [39], focusing in particular on maximum theoretical throughput, real-life feasibility and design decisions.

We compute the message complexity of Tilikum and compare it against Narwhal and Tusk and FairDAG.

The last section of the chapter compares Tilikum thoroughly against FairDAG [39], the current state-of-the-art for order fairness on DAGs.

5.1. Experimental Setup

Tilikum was implemented[2] starting from the Rust research implementation of Narwhal and Tusk[3]. A total of 3500 lines of code were modified in the main implementation, with 2500 more for the experimental setup. Some features of the production implementation of Narwhal previously used in Sui[4] were backported. In particular, re-transmission of uncommitted certificates and the order in which batches are added to block.

Experiments Machines: All experiments were run on the DAS5 distributed cluster [9]. Each individual run lasted 60 seconds and an experiment set consists of at least 5 runs. Clients were always assigned their own machine, while workers and primaries (where applicable) would share a machine if the total number of nodes would surpass the amount of schedulable machines.

Censorship Prevention Implementation: Given the results shown in Section 5.2, which we will detail there, the version of Tilikum in which transactions are assigned and included by (at most) $f+1$ machines was deployed and is used in all experiments. The other scenario where transactions are assigned to a

[1] https://github.com/apache/incubator-resilientdb/commit/9e6c46f6f1d56ed88aae1034699c12d2097b1313
[2] The code contains code snippets from Jianting Zhang and so cannot be publicly released yet.
[3] https://github.com/asonnino/narwhal/
[4] https://github.com/MystenLabs/sui/blob/narwhal-votes/narwhal

single machine, with correct nodes jumping in to include it if it is not committed, was also implemented, but it showed inferior reliability in measured metrics and overall throughput.

Experiments Parameters: Transactions have a fixed size of 128 bytes. Pompē's implementation is based on `libhotstuff` which does not exchange transactions, It instead reaches consensus on their 32 bytes cryptographic hash digest, assuming that the actual transaction is sent for execution when confirmed. FairDAG's implementation based on ResilientDB [34] operates similarly. Having the transaction size fixed to 128 bytes, which is similar in size to a cryptographic hash value, makes the comparison as fair as possible.

In Narwhal and Tilikum, workers build batches of 4 000 bytes (around 32 transactions) or after a timeout of 1 second. Primaries build blocks containing 512 bytes of payload, which is equivalent to 16 batches (each batch is represented by a 32 bytes hash). A block can also be built after a timeout of 2 seconds.

FairDAG [39] is built on top of ResilientDB [34] and has a different concept of batches. A batch is a collection of transactions from the same client and is assigned a single timestamp. To fairly compare against Tilikum and Pompē [85], the batch size was set to 1. The rationale for this decision is explained in Section 5.6.2. The released code for FairDAG does not support the simulation of faulty nodes, so this scenario was not tested. The implementation also used a modified version of Narwhal and Tusk [23] with weak edges as its DAG layer.

The clients were set up to send a fixed rate of 50 000 transactions per second on vanilla Narwhal and Tusk and 13 000 transactions per second on Tilikum. The value of 50 000 was chosen as the experimental cap found by the original authors and re-confirmed in our experiments. The discrepancy with the rate selected for Tilikum is a consequence of the difference in cost for handling transactions in the two systems. In Tilikum, transactions are broadcast by clients to all nodes and when a worker receives a transaction an extra communication step to the primary is required to assign a Timestamp Pair. We kept the Narwhal and Tusk experiment code in which clients send transactions to only one node so that the maximum achievable throughput can be showcased.

ResilientDB [34] does not have a built-in system to cap the transaction input rate, which means that it is effectively capped by the speed and the amount of clients.

The throughput and latencies experiments were run with varying values of nodes N and faulty (silent) nodes f. The number of worker nodes was kept constant at 4. Changing the number of worker nodes was not possible as the number of machines in the cluster was not high enough to allow for higher values.

Metrics Explanation: We distinguish between Consensus and Execution throughput and latency when discussing the metrics for Tilikum and FairDAG, as the protocols first perform consensus over a series of blocks and then unwrap blocks to execute transactions in a fair order. The execution throughput is effectively an upper-bound for possible transaction execution, as Tilikum does not implement a full execution layer. We can consider the consensus value to be an upper-bound (regarding throughput) or a lower-bound (for latency) of the respective execution metric.

The security and fairness evaluation was performed with fixed parameters, clients sent 13 000 transactions per second, 10 replicas were deployed with 4 workers each. Block and batch sizes were left unchanged. Two sets of experiments were performed, one with no faulty (silent) nodes and varying number of arbitragers from 1 to 5 and another with a fixed amount of 3 arbitragers and varying the number of silent replicas from 0 to 3.

It is important to note that while the Sluggish and Speculative attack do not imply that the node deviates from the protocol, the Fissure attack might lead to that happening. The experiments executing the Fissure attack were carried out with the total number of malicious nodes (arbitragers + silent) below 1/3 of the total.

5.2. Evaluating Redundancy

To evaluate which censorship-prevention method is best, we run some experiments with 10 nodes, no faulty nodes, 4 workers per nodes and a variable amount of assigned nodes that will try including the

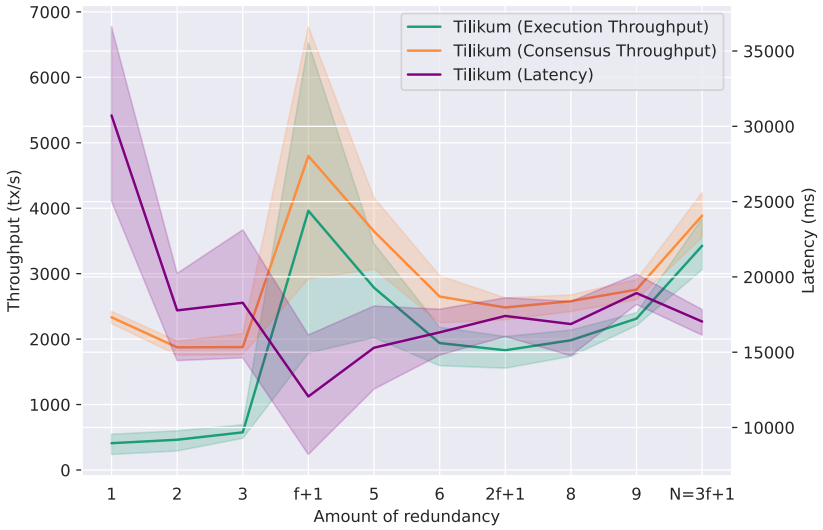


Figure 5.1: Throughput and latency when increasing the number of nodes including a transaction ($n = 10$ replicas).

transaction. This amounts grows from 1 to 10.

When the redundancy parameter r is less than $3 = f+1$, nodes implement a transaction re-injection mechanism. A starting timeout value of 8 seconds is set, after which nodes that did not include a transaction in a batch, and that did not see this transaction being committed, will include it in the next batch. When the re-injection is triggered, the timeout value is slightly increased.

In Figure 5.1 we can see the values of latency, consensus and execution throughput of Tilikum when changing the number of nodes that are assigned to include a transaction. In correspondence of the value $4 = f + 1$, we can see how throughput is at its highest while latency is at its lowest.

When re-injection is necessary, in particular in the case with no redundancy, latency is heavily affected. We also see a bigger gap between consensus and execution throughput, which happens due to possibly missing timestamp values also being delayed.

We also notice an increase in throughput as the value increases until it reaches its maximum. This happens as the chances that a specific transaction is delayed by slower or crashing nodes are lower when more nodes are assigned to it. Moreover, if a transaction appears more times we also can see more timestamps related to it, filling up the logical table faster.

5.3. Transaction Handling Performance Evaluation

Tilikum is implemented on top of Narwhal and Tusk [23] and implements the ordering linearizability property of Pompē [85]. FairDAG-AB [39] also implements ordering linearizibility. In this section we compare performance metrics related to transaction handling between these three systems.

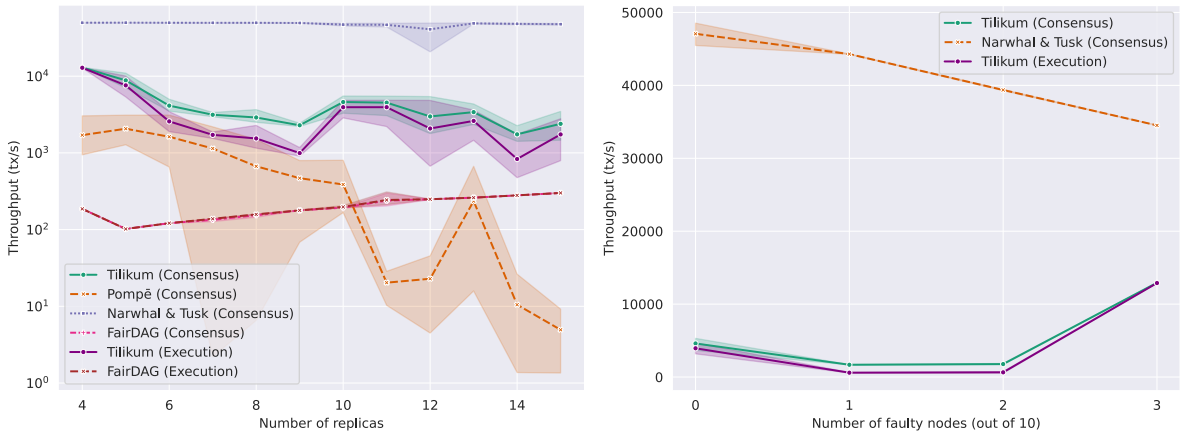
5.3.1. Consensus and Execution Throughput

In Figure 5.2a we can see how throughput changes when increasing the number of total replicas.

We can notice that the overhead incurred by Tilikum, as more data needs to be exchanged more times, is reduced by at least a factor of 4 and on average a factor of 10 compared to Narwhal and Tusk [23].

Tilikum remains faster than Pompē [85], with the throughput being around 5 times higher on average. Pompē's throughput is very noisy, and the values vary greatly between different runs. We think it might be related to the time slotted consensus mechanism, resulting in long periods of stalling in specific scenarios.

FairDAG's consensus and execution throughput are always identical, showing how the combination of redundancy and weak edges allows transactions to be executed quickly. This is also showcased by the



(a) Throughput when increasing the number of replicas

(b) Throughput when increasing the number of faulty replicas

Figure 5.2: Throughput comparison of Tilikum, Pompē, Narwhal and Tusk and FairDAG ($n = 15$ replicas)

low latency of the system.

The consensus throughput of Tilikum is, on average, around 22 times higher than FairDAG's, although this might just be a consequence of the underlying consensus protocol, the fact that the Tusk [23] implementation in ResilientDB [34] does not employ a worker-primary architecture, transaction redundancy and weak edges. The execution throughput of Tilikum is 17 times higher than FairDAG's on average.

It also seems that FairDAG's implementation on top of ResilientDB scales better with a high number of nodes. This is surprising as having more nodes also means that each transaction needs to appear more times for it to be executed. We can explain this as a consequence of having more transactions in the system. We were unable to set a cap on the transaction input rate in ResilientDB's benchmark set-up, unlike Narwhal and Tusk and Tilikum. This means that as we add more nodes in the system and so more clients, we also increase the amount of transactions that are inserted in the DAG.

For $N = 4$ and $N = 5$, execution performance almost reaches its theoretical maximum value, capped by consensus performance. The other case when this happens is when the number of silent nodes reaches $f = 3$, shown in Figure 5.2b.

From analyzing execution logs we noticed that the limitations discussed in Section 4.8 may stop the threshold from growing until retransmission is successful. When enough nodes are silent or the number of nodes that might be excluded from proposing at a specific round is small, this effect is limited and we obtain results that show the system's full potential.

5.3.2. Consensus and Execution Latency

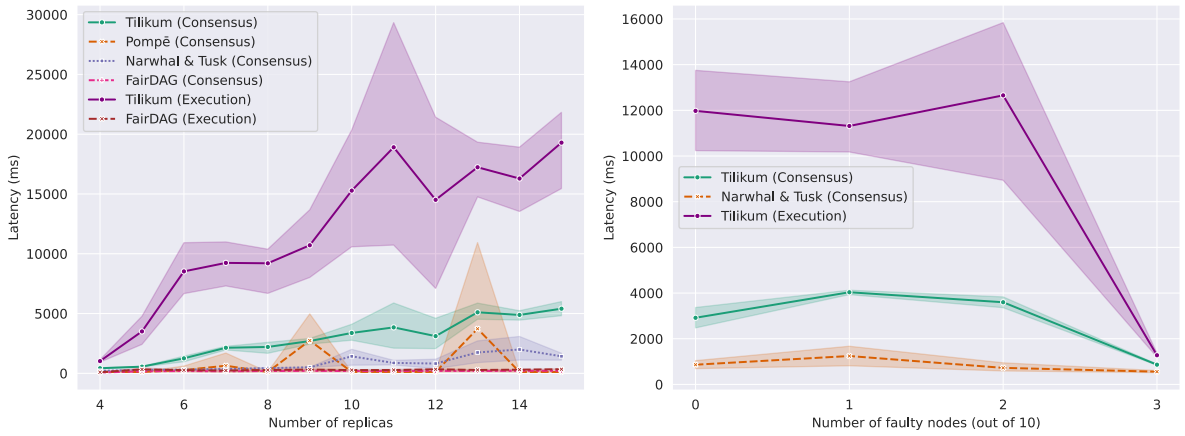
In Figure 5.3a we can see how latency changes when increasing the number of total replicas.

Latency is heavily affected by the extra communication rounds required to collect timestamps, which is noticeable in the Consensus latency. Execution latency is also heavily impacted, as the threshold needs to reach specific values to allow committed transactions to be executed.

Similarly to throughput, we notice that in the case of $f = 3$, shown in Figure 5.3b, when all correct nodes are able to successfully insert a block in every round there is a sudden decrease in execution latency.

The measurements for Tilikum are noticeably noisy, especially for the execution metric. This is a consequence of the strict requirements for execution. Tilikum needs the holes to be filled, which might require retransmission of blocks and metadata. The need for retransmission is network dependant and diferent runs may be more lucky than others.

FairDAG's [39] usage of weak edges allows for the execution threshold to grow at a more consistent and reliable pace. This is visible in the low and consistent latency values. We believe that implementing



(a) Latency when increasing the number of replicas

(b) Latency when increasing the number of faulty replicas

Figure 5.3: Latency comparison of Tilikum, Pompē and Narwhal and Tusk

something similar to weak edges in Tilikum will decrease its latency too, but the extra communication required for timestamp collection will still influence it heavily. The time between receiving a transaction and it being included in a batch is now much higher as Tilikum requires two rounds of reliable broadcast. This results in consensus latency about 4 times higher than Pompē [85] and 15 times higher than FairDAG [39] on average. Execution latency is 40 times higher than FairDAG and 17 times higher than Pompē on average.

5.4. Security Evaluation

This section showcases how Tilikum successfully mitigates MEV extraction and re-ordering attacks. The attacks from the work of Zhang et al. [83] are implemented and run against Tilikum's implementation and Narwhal and Tusk [23].

5.4.1. Attacks Description

In 2024 Zhang et al. [83] evaluated some novel MEV extraction strategies on Narwhal and Tusk [23] and Bullshark [70]. These attacks proved to be highly effective with ASRs (Attack Success Rates) up to 87% on Tusk and 97% on Bullshark.

They developed three attack strategies, that were kindly provided to me for testing on Tilikum. These strategies are: the fissure attack, where attackers "disconnect" victim's blocks; speculative attacks, where blocks with higher priority are built speculatively and; the sluggish attack, where attackers delay the proposal of a block from previous rounds with higher priority (i.e., lower digest).

This subsection first defines some terms and concepts used in evaluating the attacks, then defines the threat model and finally describes all attack strategies.

Terminology

To evaluate whether an attack is successful or not, we need to define some terminology and a threat model. Zhang et al. [83] mainly focused on front-running and so the following definitions assume that is the main goal of an attacker.

Definition 5.4.1 (Block committing order)**.** Given two blocks B_i and B_j, we say that $B_i \prec_c B_j$ if B_i is ordered before B_j, once they are both committed.

We can define a similar relationship for transactions as follows:

Definition 5.4.2 (Transaction execution order)**.** Given two transactions T_i and T_j, we say that $T_i \prec_c T_j$ if T_i is executed before T_j or if T_i is executed and T_j is not.

Note how now we care about transaction **execution**, as our fairly-ordered system separates block commitment from transaction execution.

The cases where at least one of the transactions is discarded are tricky. As defined in Section 3.1,

front-running might make the victim transaction fail. In some cases that is considered the intended result, but in cases such as sandwich attacks, that might actually be undesirable.

In this work, we consider the attack successful if at least one of the two scenarios is realized, so either the attacker's transaction is executed before the victim's one or the victim's transaction fails to execute.

We can now define a security game, where an attacker a builds blocks B_{ai} and a victim v builds blocks B_{vi}.

The security game is then:
Definition 5.4.3 (Frontrunning Security Game)**.** If an attacker a wants to front-run a victim v's transaction T_v with transaction T_a, the attacker wins the security game when $T_a \prec_c T_v$.

In the base Narwhal system, where transaction ordering fully depends on the block committing order, this results in two winning conditions:
Definition 5.4.4 (Committing Winning Rule 1 (CWR1))**.** If B_a and B_v both belong to the same round r, then $B_a \prec_c B_v$ if and only if B_a is positioned to the left in the casual history tree spawned by an elected leader block.
Definition 5.4.5 (Committing Winning Rule 2 (CWR2))**.** If B_a and B_v both belong to the different rounds r_i and r_j respectively, then $B_a \prec_c B_v$ if and only if $r_i < r_j$.

Definition 5.4.5 does not consider the case where the two blocks are committed during different leader elections (i.e., they are not in the same casual history tree for some elected leader). Zhang et al. [85] do not consider this situation in much detail as it makes the attacks more inconsistent and can just be treated as a case of attack failure. In any case, this does not invalidate the following definition where we consider transaction execution order instead of block committing order.
Definition 5.4.6 (Execution Winning Rule (EWR))**.** Consider an attacker a that wants to front-run a victim v's transaction T_v with transaction T_a. These transactions are assigned timestamps ts_v and ts_a respectively. If $ts_a < ts_v$, then $T_a \prec_c T_b$ and the attacker wins.

The attack strategies described below try to build an attacking block to front-run a specific victim block. When dealing with individual transactions, we consider the attack successful if at least one transaction of the attacking block runs before at least one transaction in the target block. This definition of success is very lax and strongly favors the attacker. Normally, an attacker would profit only if the specific front-running transaction executes closely before the victim's. We believe that the use of this definition better showcases the strength and robustness of Tilikum.

Threat Model

In a normal asynchronous network, the number of tolerated Byzantine faults in the network for both Narwhal and Tusk [23] and Pompē [85] is $f < N/3$. These nodes are called **Byzantine** and in Narwhal and Tusk their objective is to disrupt the network to undermine liveness, latency and safety. In Pompē the assumption is that Byzantine nodes try to disrupt fair ordering, by submitting timestamps that benefit them.

This second definition is more similar to the definition of a **frontrunner** attacker in the work from Zhang et al. [83], who instead consider a scenario where at most $N - 1$ parties operate under these rules.

In this work, we also consider up to $N - 1$ frontrunners (i.e., nodes that execute the following attacks which do not violate the protocol), but only up to $f < N/3$ of these nodes are capable of lying about their timestamps. If we allow more than f nodes to submit fake timestamps, ordering linearizability cannot be achieved, as proved in Lemma 4.7.6. One of the following attacks, the Fissure attack, may violate the protocol specifications and so the sum of the number of faulty (fully Byzantine) nodes and frontrunner (building the fissure) cannot exceed $f < N/3$.

Fissure Attack

Consider CWR1, where an attacker wants their block to be positioned further to the left of the DAG than the victim. Consider two blocks in round r, the chances that one is ordered before another (i.e., is on the left in the tree with the anchor as root) are directly correlated to how many paths there are between each block and the elected leader in round $r + 2$.

Given this observation, an attacker could deliberately not connect their blocks to the victim's one, effectively reducing the number of paths to them. The structure of blocks that the attacker built is called a **fissure** Note that performing this attack is a violation of the protocol.

A node should create a block for round r when they received $2f+1$ certificates from round $r-1$, as it is not possible to wait for more. There are no guarantees that more than $2f+1$ certificates will be received for a specific round. Not only may the f Byzantine nodes be silent and never propose one, but the protocol specifically says that nodes have to stop approving blocks belonging to previous rounds, with round progression occurring after processing $2f+1$ valid certificates.

A round could then have only $2f+1$ certificates circulate and if one of them was assembled by the victim, all fissure building nodes will never create any new block, effectively becoming silent. We noticed that this case was not handled in the code provided by Zhang [83], but we fixed it in our implementation. Attackers can try building a fissure for some time, but should not wait indefinitely.

Speculative Attack

This attack tries to exploit the ordering rule for blocks that are in the same round and are considered to be tied in the causal history of the elected leader block, to then win according to CWR2. A common way to randomly break ties is to use digests. If a node needs to choose between two blocks that are tied for their ordering, they can compute the hash of both blocks and commit one before the other according to the hash values. Narwhal and Tusk's implementation executes first the block (certificate) with the **highest** value, as described by Zhang et al. [83].

An attacker who knows this information can then select which batches of transactions to include in their proposed headers, until a block with a particularly high has value is created. This attack does not violate any requirements from the protocol, as there is no restriction on when transactions should be included in a block, just that every node that is assigned a specific one should do it.

Sluggish Attack

According to CWR2, if an attacker can submit a valid block for round r', while the victim already proposed a block for round $r > r'$ and both these blocks will eventually be committed, than the attacker would successfully frontrun the victim. The sluggish attack attempts to achieve this by delaying the proposal of blocks for specific rounds until an interesting victim block appears.

The success rate of this attack greatly depends on network conditions and the number of active nodes in the system, as it requires enough nodes to both approve the attacker's block at round r' and also enough nodes that will approve the victim's block at round $r > r'$. However, it seems that the implementation of Narwhal and Tusk does not actually fully implement the correct conditions to approve a block. The paper reports that "a valid block must be at the local round r of the validator checking it", but this is not reflected in the code, effectively amplifying the effects of this attack.

5.4.2. Attack Success Evaluation

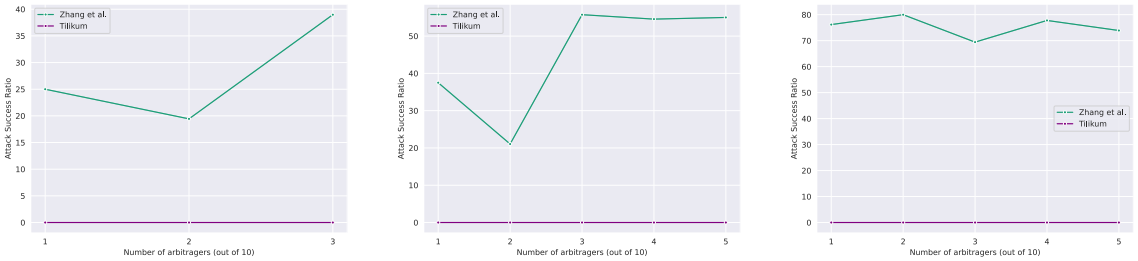
Thanks to Jianting Zhang who provided the code used in the paper describing the attacks [83], we were able to evaluate Tilikum against the original implementation. The small bugs described before were fixed, and the attacks were run with the parameters shown in Section 5.1.

First we show the attack success rate (ASR) when varying the number of frontrunning nodes, then we present the results for the cases where the number of attacking nodes is constant, but other faulty nodes may become silent.

Increasing the Number of Attackers

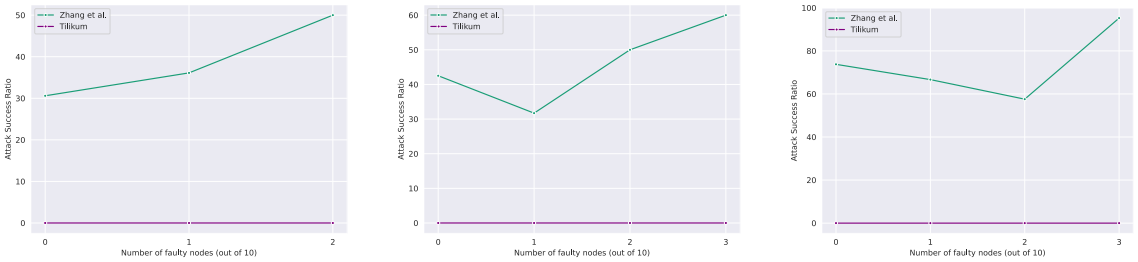
In Figure 5.4 we can see the ASR of the three attack strategies for different numbers of attackers. Tilikum completely stops the attacks from working. This should not come as a surprise, as the attacks work on the assumption that **block ordering** and **transaction ordering** are completely correlated, while Tilikum unwraps blocks and executes transactions based on timestamps.

We can still draw some conclusions about the effectiveness of the attacks. First, examining execution logs, we noticed that overall the number of times attackers successfully **built** an attacking block was reduced. If we account for the extra communication necessary to build valid blocks in Tilikum, the



(a) Attack success rate of the Fissure Attack **(b)** Attack success rate of the Sluggish Attack **(c)** Attack success rate of the Speculative Attack

Figure 5.4: Attack effectiveness with varying number of attackers



(a) Attack success rate of the Fissure Attack **(b)** Attack success rate of the Sluggish Attack **(c)** Attack success rate of the Speculative Attack

Figure 5.5: Attack effectiveness with varying number of silent nodes

window of opportunity for the attack is reduced. The time between the moment the attacker first sees the victim's block and when all the nodes are still building blocks that could refer to the attacker's block is reduced.

Attacks with Silent Nodes

The presence of other faulty silent nodes can increase the effectiveness of attacks. Intuitively, if fewer nodes propose blocks in each round r, the chance that our block will be connected to more blocks in round $r+1$ increases.

This directly benefits attackers running both the Fissure attack and the Speculative attack. Regarding the Sluggish attack, when we maximize the number of silent nodes it is actually impossible for other nodes to progress forward without receiving the attacker's block. This is because nodes require $2f+1$ blocks of round r to start building the block of round $r+1$ and if our number of active nodes is exactly $2f+1$, all nodes need to propose to progress.

The reason we believe the ASR still increases is simply related to the fact that all blocks coming from the attacker will be referred to in successive rounds, effectively guaranteeing they are committed.

Given the separation of consensus and execution, Tilikum remains unfazed by the attacks, as clearly visible in Figure 5.5.

5.5. Message Complexity

In this section, we analyze the number of messages necessary for a transaction to be included by different protocols.

As a disclaimer, Danezis et al. [23] claim that analyzing message complexity for systems such as Narwhal and Tusk is not useful as it does not make a distinction between metadata and block contents and that exchanging lots of small messages is heavily penalized. We still compare the message complexity of Tilikum with the base implementation of Narwhal and Tusk to give a general idea of their overhead, and with FairDAG to try to explain the difference in observed performance.

All calculations assume a block size of B batches and batch size of t transactions. This means that 1 message at the batch level is actually a tth of a message compared to the ones on the transaction level.

At the block level one message is a Bth of a batch message and a tBth of transaction messages. For FairDAG, as no batching is performed, we consider blocks to include tB transactions. We ignore small messages such as acknowledgments.

In Narwhal and Tusk each client sends their transactions to a single replica. Replicas will then include them in a batch, which is then reliably broadcast. From here on, blocks will refer to batches instead of including transactions. With a committee of size N, Narwhal requires $1 + 2\frac{(N-1)}{t}$ message to disseminate and store the transaction, $\frac{N-1}{tB}$ messages to disseminate the block, $\frac{(N-1)}{tB}$ messages with acknowledgments of block reception and another $\frac{N-1}{tB}$ messages to disseminate the certificate. Note that messages containing batches and blocks are used by multiple transactions, so the complexity is amortized by the relevant factor. The message complexity for a single transaction is then $\mathbf{1 + 2\frac{(N-1)}{t} + 3\frac{N-1}{tB}}$, with the largest messages (blocks) shared by tB transactions.

In Tilikum, clients send transactions to all nodes in the committee, so that a timestamp can be assigned as quickly as possible. The nodes or a subset of them will then try to include the transaction in a block. Assuming that a nodes are assigned the transaction and will try to include it, the transaction dissemination step will then require N messages from the client and $2a\frac{(N-1)}{t}$ messages from the validators to the other validators. We also need to collect timestamps for the broadcast batch, which adds another $2a\frac{(N-1)}{t}$ messages to the count. The block dissemination and certification phase is identical, but we multiply by a factor of a, which results in $3a\frac{N-1}{tB}$ In the average case $a = f + 1 < \frac{N}{3}$, which means the largest term is actually quadratic, as it is $\frac{N^2}{3t}$. The largest term in the message complexity is then $\mathbf{\frac{N(N-1)}{3t}}$, with a total complexity of $\mathbf{N + 2\frac{N(N-1)}{3t} + \frac{N(N-1)}{tB}}$ and an asymptotic complexity of $\mathbf{O(\frac{N^2}{t})}$.

In FairDAG, clients first send transactions to all nodes, which amounts to N messages. We assume an implementation of Narwhal and Tusk analogous to the one used by Tilikum is the underlying DAG layer. All validators will then include this transaction in a batch, which is broadcast to everyone with a minimum of $\frac{N(N-1)}{t}$ messages. Blocks are then built and broadcast, which amounts to $N\frac{N-1}{tB}$ messages, plus $N\frac{N-1}{tB}$ messages containing signatures to build the certificate. The certificate is built and broadcast for an additional $\frac{N(N-1)}{tB}$ messages. The total message complexity is then $\mathbf{N + 2\frac{N(N-1)}{t} + 3\frac{N(N-1)}{tB}}$.

The asymptotic message complexity of FairDAG is then $\mathbf{O(\frac{N^2}{t})}$, which is identical to the one of Tilikum. If we look at the exact values, FairDAG is higher because a factor of N is present instead of $a = f + 1 < \frac{N}{3}$.

5.6. Detailed Comparison with FairDAG

In this section we compare against FairDAG [39], the current state-of-the-art for order fairness on DAGs. We first summarize the protocol and what properties it upholds. We then comment on some design decisions, their issues, possible mitigation, and how Tilikum does not suffer from them.

5.6.1. Protocol Description

FairDAG [39] assumes a working DAG-based system, such as Narwhal and Tusk [23], DAG-Rider [40] or Bullshark [70], with some modifications and assumed properties. They call this abstraction the **DAG Layer**, which is described in the first part of this section. They then define a second part of the protocol to implement order fairness called the **Fairness Layer**, described in the second part.

The network is assumed to be asynchronous, but non-adversarial. The protocol has "no assumption that clients always behave correctly; they may exhibit arbitrary or malicious behavior" [39], but also states that clients will send transactions to every replica. We consider this to be an inconsistency.

DAG Layer

The DAG layer should guarantee **agreement**, every correct replica commits the same blocks, **total order**, every correct replica commits blocks in the same order, and **validity**, every block proposed by a correct replica is eventually committed. Note that Narwhal and Tusk do not guarantee validity [23], they only assume it is respected in expectation due to retransmissions.

To guarantee all these properties, the underlying DAG protocol is modified so that **weak edges** are introduced. As explained in Section 4.8.1, weak edges are connections between blocks of round r and

Protocol	Garbage Collection	Low Redundancy	Robust against Malicious Clients	Transaction Insertion Order	Rebroadcast
FairDAG [39]	✗	✗	✗(✓with perf. degradation)	In-order	None
FairDAG (no weak edges)	✓	✗	✗(✓with perf. degradation)	Out-of-order	Full Payload
Tilikum (this work)	✓	✓	✓	Out-of-order	Hole fillers + Digests

Table 5.1: Differences between Tilikum and FairDAG

blocks of round $r' < r$, and a maximum of f such edges can be included in a valid block.

With a working DAG layer, FairDAG then requires replicas to add extra information to their vertices. When a replica first receives a transaction, they will assign it a monotonically increasing **Ordering Indicator**. A replica in FairDAG will try to include every transaction it receives from a client. When a transaction is included in a block, its ordering indicator is added too.

The DAG layers can then execute normally, and the ordering indicators are later used by the **Fairness Layer** to extract fair ordering. FairDAG supports two definition of fairness: γ-batch-order-fairness (FairDAG-RL) and Ordering Linerazibility (FairDAG-AB). As Tilikum also implements Ordering linearizability, we will focus on FairDAG-AB.

Fairness Layer
The Fairness Layer is responsible for executing transactions in the correct fair order. For the Ordering linearizability version, there are two main steps: computing the execution threshold and computing the **Assigned Ordering Indicator** (AOI) of transactions. The AOI will be the timestamp used to determine the final ordering, while the execution threshold has the same role as the threshold we compute in Tilikum (described in Section 4.5).

The execution threshold is defined as the minimum **Lowest Possible Assigned Ordering Indicator** (LPAOI) of the transactions that a replica has **seen**. A transaction is considered to be seen when it is added to the DAG through any vertex. For all seen transactions, a replica stores all the seen ordering indicators. The transaction's LPAOI will be the $(f+1)$th lowest seen value. In case less than $f+1$ ordering indicators have been seen, for each replica that has not included this transaction in a block yet, their currently highest proposed ordering indicator is used.

The AOI of a transaction t is computed when at least $N - f = 2f+1$ ordering indicators of t have been **committed**. The AOI will be equal to the $(f+1)$th lowest value in this list, which is equal to the median in the case of exactly $2f+1$ values.

The paper claims that this respects Ordering linearizability, which we agree with. The AOI are always bound by correct values like in Tilikum. There is an assumption that blocks authored by any correct replica n for round r will not contain ordering indicators larger than the ones in blocks $r + i$ with $i > 0$. While this is respected and can be enforced if not, it can still raise some issues as the order of received messages is not guaranteed by the asynchronous network. To mitigate issues where future rounds blocks are received before older ones, the authors require that all blocks from replica n in round r refer to the block from replica n in round $r - 1$. This system is not functional if weak edges are not present. This is explained in detail in Section 5.6.2.

Moreover, under the assumption of malicious clients, it is possible that the execution threshold is blocked indefinitely, unless some attention is put in the implementation. These attack strategies are showed later in Section 5.6.2.

5.6.2. Design Issues & Mitigations

In this subsection we discuss what we think are design issues in FairDAG [39], how they could be mitigated and why Tilikum does not suffer from them. The first part discusses the disadvantages of using weak edges and having redundant transactions in the DAG. Finally, we propose three attack scenarios where a single malicious client can break liveness for the system, and how to prevent this.

Table 5.1 summarizes the main difference between Tilikum and FairDAG.

Weak Edges & Redundancy

FairDAG [39] relies on weak edges for its implementation. As a transaction needs to appear $2f+1$ times in the DAG to be executed, losing the strong theoretical guarantees given by the Validity property is not permissible.

As already explained by Danezis et al. [23], maintaining weak edges comes at the cost of losing garbage collection. Without garbage collection, the system is unusable in practice, which we believe is a big shortcoming in the design. The solution adopted by Narwhal and Tusk to achieve validity in the absence of weak edges is to retransmit the payload (i.e., transactions) of blocks that were not committed [23] together with the payload of the block for the round where this is detected. This approach cannot be easily adapted to FairDAG. The execution threshold relies on the assumption that the timestamps in the block created in round r by a replica are always smaller than the ones in future rounds (i.e., $r + i$ for $i > 0$).

If we allow the retransmission of the uncommitted payload of a block created in round r during $r + i$, the blocks of rounds $r < r' < r + i$ will effectively include ordering indicators that are larger than the ones in the block for $r + i$, breaking the assumption. Removing the need for weak edges is then not trivial and reduces the applicability of FairDAG in a realistic scenario. To remove weak edges, replicas should rely on a way to detect the actual exact order of block's payloads and detect missing ones that might be retransmitted. A mechanism similar to Tilikum's logical clocks could be employed.

Apart from disallowing garbage collection, the way in which FairDAG computes the execution threshold and its reliance on weak edges **does not allow** for transactions to be inserted into the DAG in a different order than the one shown by their timestamps. This assumption is relatively strong and can be problematic when paired with an asynchronous network.

Tilikum does not require transactions to appear more than once in the DAG for it to be executed, as all necessary information is bundled in the batch's metadata. As now the DAG will have redundant information, if we assume that no other factor is bottle-necking the system, its throughput should be at most the one achieved by the underlying DAG layer divided by the amount of (correct) nodes. This does not seem to be the case in the plots provided by the authors [39]. The author re-implemented Narwhal and Tusk [23] on the same framework they used for FairDAG [39], ResilientDB [34]. We think that the reason for this discrepancy is found in the way transactions are treated.

ResilientDB [34] allows client to batch transactions. These batches are then treated by the underlying protocol (in this case FairDAG on top of Narwhal and Tusk) as a single transaction. This results in a batch of transactions being assigned a single timestamp, removing fairness between them. Moreover, the throughput values were effectively multiplied by the batch size. The default value available in the code was of 400, which means that each 400 transactions were assigned a single timestamp.

We also noticed that the script extracting the throughput and latency values from the execution logs was intentionally excluding values lower than 1000. While we do not know the rationale behind this decision, we think it is not a correct behavior. Both Tilikum and FairDAG's execution speeds depend on a threshold which might not grow or grow particularly slowly in periods of network asynchrony or other types of faults. We think that measuring the system's behavior under these circumstances is more correct, and we did so in our experiments.

Attacks on Liveness with Malicious Clients

FairDAG assumes that "clients generate and submit transactions to replicas, then await execution results in response", and that "there is no assumption that clients always behave correctly; they may exhibit arbitrary or malicious behavior" [39]. This however clashes with the later assumption that "clients broadcast their transactions to all replicas" [39].

If we actually assume arbitrary client behavior, the system suffers from some of its design choices. We now propose three scenarios in which malicious clients can halt the system and break liveness. The first two scenarios are slight variations of each other, but they achieve different results. The first one leads to an overall slowdown in throughput, while the second can potentially stop the execution threshold from advancing, completely blocking the system. We then suggest two mitigation strategies which might have consequences on the scalability of the system. The last attack strategy is more inherent to the system's assumptions.

The main way clients can misbehave is by not actually broadcasting their transactions, but instead send them to a subset of replicas. This has negative outcomes for the client, as also pointed out by the authors, as malicious replicas could censor or delay the inclusion of transactions in their blocks. However, the system makes no assumption that correct replicas will try to include transactions that have not been sent directly by a client.

A transaction received only by a subset of nodes will then be included only by that same subset of replicas. We will now see how different sizes of this subset impact the system, with two main types of attacks.

Let's start with the simple case where a client sends a transaction to $S < f+1$ nodes. In this case, those nodes will use some of their block space to include this transaction, thus reducing throughput. The LPAOI of this transaction will however always increase. The entry in `lp_ois` for the S nodes that received it will be constant, but for all other nodes it will keep increasing as the DAG expands. Given that the LPAOI is the $(f+1)$th lowest value in the array and there are less than $f+1$ constant values, then it will always be changing. The only consequence of this attack is reduced throughput, as block space is taken by this unexecutable transaction, and higher resource usage for the nodes that need to keep track of the Ordering Indicators for this transaction.

The second possibility for clients is more impactful. If a malicious client sends a transaction t to $f+1 < S < 2f+1$ nodes, the LPAOI will be eventually computed to a constant value, but the transaction will never be executed. As the rest of the DAG grows, the LPAOI of t will eventually be the smallest and become the execution threshold for the system. The transaction t will never receive enough ordering indicators for execution and so it will always remain in the set of transactions without AOI, and always be the transaction determining the execution threshold. The system will then fully come to a halt, with the DAG continuing to expand, but no new transaction being executed.

There are two ways in which this issue can be solved, and they both have some impact on the overall throughput of the system. The simpler way requires correct replicas to re-broadcast all transactions they receive to all other replicas, similarly to the third mitigation proposed by Vafadar et al. [73]. The second attack scenario is fully stopped by this mitigation. At least one correct replica will receive the transaction and broadcast it to all other replicas, making it eventually executable. The first attack scenario is still possible, but its impact was not as severe as the second one to begin with.

If we want to fully stop the first attack as well, we can introduce a way for replicas to include transactions in their blocks even if they were not received by a client (or by another replica during the re-broadcasting step described above). A correct replica that receives a valid block including a transaction digest d that was not received by any client, can consider the receipt of this block as seeing the transaction for the first time, thus assigning its own ordering indicator and adding d to its next block. This is similar to what Tilikum does. A request for timestamps for a specific batch may include unseen transactions, which are then assigned a Timestamp Pair on the spot.

The last way malicious clients can influence the system has to do with how FairDAG stores transactions in the DAG. Transactions are actually **not** stored in the ledger, but a cryptographically secure digest is used instead to reduce network and storage usage. This works correctly assuming that all replicas have access to all transactions, and have a fast way to match the committed digest to the actual transaction content. A malicious client partially sending a transaction, enough times to allow execution (i.e., $2f+1$), can stop nodes that did not receive it from knowing its contents, stopping their ability to maintain the correct state.

This attack is fully mitigated by the first strategy described above. If at least one correct replica sees the transaction, they will broadcast it allowing every other correct node to store it.

Note that Tilikum does not suffer from any of the above issues, all information required for executing a transaction is available in any block including the transaction and the execution threshold always grows, at least at the same speed as the slowest correct node. We believe that one of the above mitigations should be implemented in FairDAG [39], so that their impact on the performance metrics can be measured.

6

Possible Extensions and Future Work

This chapter describes how Tilikum can be extended to support other definitions of fairness in sections 1 and 2. The third section lists which parts of Tilikum require further improvements, in particular the improvement of throughput and latency by better handling of the execution threshold. The last part talks about more evaluation experiments that might be interesting to better understand how to optimize Tilikum.

6.1. Batch Order Fairness

Tilikum focuses on Ordering linearizability, but that is not the only definition of fairness. Another common definition is γ-batch-order-fairness (also explained in Section 3.4). To compute a fair ordering under γ-batch-order-fairness as defined in Themis [43], we require that for each pair of transactions at least $2f+1$ replicas propose their preferred ordering.

The current block structure in Tilikum contains such information for transactions in the same batch, as each batch has $2f+1$ Timestamp Pairs that can be used to extract which transaction was seen earlier. When we want to order transactions from different batches, the situation becomes more complicated. We have no guarantees that the two sets of $2f+1$ nodes that participated in the timestamp collection for the two batches fully intersect. It can then happen that for a specific pair of transactions tx_1, tx_2 we do not have enough ordering preferences.

Nodes should then have a way to submit orderings when necessary, similarly to how hole fillers currently work in Tilikum. We leave this as future work due to time constraints and scope.

6.2. Fairness Definitions Combinations

Both Ordering linearizability and γ-batch-order-fairness try to order transactions as much as possible, that is until there is enough evidence for a transaction to be executed before another. These unsolvable scenarios can be exploited by malicious actors to exclude specific transactions from being fairly ordered, such as what was shown by Vafadar et al. [73].

Combining different fairness definitions to try to fairly order these unsolvable cases was never explored. It could be possible to first order transaction using γ-batch-order-fairness, similarly to what Themis [43] and FairDAG [39] do, and then order transactions inside the same loop using Ordering linearizability. The other way around is also possible, first transactions are ordered through linearizability, and if the timestamp ranges of two transactions overlap, ordering preferences over these two specific transactions can be collected.

To the best of our knowledge, this approach has never been proposed or evaluated. An extensive theoretical and practical analysis could reveal its efficacy.

6.3. Improving the Threshold Growing Speed

In the Evaluation chapter (Chapter 5) we noticed that Tilikum's performance is comparable to, if not better than, the current state of the art. What slows down Tilikum is the computation of the execution threshold. The threshold can get stuck if the network conditions delay the commitment of specific blocks that contain transactions currently not seen in any block. There are many ways in which this can be fixed, here we will list some preliminary ideas that can be further explored in future work.

The simplest way to drastically reduce the amount of blocks that might not be committed is to uphold Validity by re-introducing weak edges. While this would fix this specific issue, and make for some nice plots that showcase the high throughput of Tilikum, it would make the system impractical, which we think is worse than the current situation.

A compromise could be to introduce partially weak edges, that is weak edges that are limited in how far backward they can go. As consensus should happen every 3 rounds, we can allow weak edges for blocks in round r to refer t blocks in rounds $r-2$ and $r-3$. This would allow us to maintain garbage collection, while also decreasing the chances of blocks receiving a valid certificate despite staying uncommitted and allowing every replica to propose a block in every round. This approach has also be seen existing in Sui[1] A theoretical and practical analysis of this solution should be performed in future work.

Tilikum currently employs retransmission of the uncommitted block of round r after committing own blocks with rounds higher than r. While this gives some better guarantees in expectation, in practice we notice how delaying the inclusion of transactions has significant impact on throughput and latency. Correct nodes could employ a mechanism to allow faster recovery for slower nodes. When new entries in the logical table for specific replicas keep coming, but older holes are not getting filled, correct nodes that are far ahead could allow these nodes to submit a block in their place instead. While this is a very rough idea, if implemented correctly and with Byzantine Fault Tolerant strategies, it could be a valid alternative to weak edges.

Finally, the main structure of Tilikum leverages the block structure and implementation detail of Narwhal and Tusk [23]. It is still possible to adapt the necessary changes to a different DAG-based ledger, such as Bullshark [70] and Mysticeti [7]. Mysticeti in particular implements a faster consensus mechanism which allows (almost) direct commitment of blocks. This could be of great benefit to Tilikum, as it might reduce the chances of uncommitted blocks stalling the execution threshold.

6.4. Larger Scale Evaluation

Given the current hardware limitations of the cluster we used for the experiments, it was not possible to perform large scale experiments. Currently, Sui has more than 100 nodes operating as validators. [2] These nodes are also geo-distributed, introducing extra latency in communications. Our experiments evaluated Tilikum with at most 15 validators, in the same local network. To better understand how Tilikum performs in the real-world, larger experiments must be performed to confirm the results obtained.

[1] We can see that an age limit was implemented. (github.com)
[2] https://suiscan.xyz/mainnet/home

7

Conclusion

Blockchain systems revolutionized the way we view online payment systems, with Bitcoin being the prime example of how a decentralized currency can be used [56]. While Bitcoin enabled parties to exchange one type of token, Ethereum [78] enabled running arbitrary code through the use of smart contracts [72].

Smart contracts enforce the rules they are coded with in the same way a legal contract can be enforced through law. As Smart Contracts effectively enable arbitrary code and rules, the natural evolution of cryptocurrency was to use smart contracts to develop alternative decentralized financial systems, birthing DeFi. If traditional finance is highly regulated and transactions can happen only through authorized brokers and markets, in DeFi the system runs without a central authority that can enforce or change rules. While this can be seen as an advantage, the only enforced rules are the ones written in code, it enables some malicious behaviors.

In traditional finance, brokers are not allowed to use information obtained through their clients to obtain a profit. Such behavior is considered fraudulent and a type of market manipulation. While brokers do not exist in DeFi, miners and validators have similar capabilities.

Validators see the state of the transaction queue (mempool) before these transactions are executed, which allows them to craft ad-hoc transactions to extract profits. The value that validators can extract by manipulating the content of blocks is called Miner Extractable Value (MEV). This behavior is (currently) not illegal or possible to enforce effectively, as it is intrinsically enabled by how decentralized ledgers work.

Many proposals are trying to minimize the negative impact of MEV extraction, but the approaches are very diverse and define the mitigation as successful in incompatible ways [80]. A promising approach is to order transaction fairly, based on the order in which different validators see transactions. This method is called Order Fairness, and even between methods that spouse this definition, many differences exist.

DAG-based blockchain protocols achieve higher throughput and lower latency through the separation of block dissemination and consensus [23, 75]. Such protocols have been implemented in practice too, with systems such as Sui [11], IOTA [63], Spectre [68], GHOST [69], and Prism [8] gathering a large amount of users. As the definition of DAG-based protocol is quite lax, many design philosophies and graph structures can fall under this term.

The amount of systems combining DAGs with Order Fairness methods is low, with only FairDAG [39] being released recently. While FairDAG's design has some strong points, some of its design decisions are not compatible with realistic systems.

In this work we proposed Tilikum, a Fairly Ordered DAG-based ledger based on Narwhal and Tusk [23]. Tilikum implements Ordering linearizability as defined by Pompē [85], while maintaining the desirable properties of using a DAG-based mempool. Our design surpasses the shortcomings of FairDAG at the cost of slightly less strong theoretical guarantees and lower performance. Tilikum does not require

weak edges, enabling garbage collection, and allows transaction execution even when the transaction has been inserted in the DAG only by a single replica.

The experimental evaluation showed the potential of Tilikum, although it does not surpass the current state of the art. We suggested some improvements that can be implemented as future work, and some extensions to enable Tilikum to support multiple definitions of fairness.

We believe that the ideas that build Tilikum can become the foundation of future Fairly Ordered DAG-based ledgers, as Tilikum maintains desirable properties that enable the system to be implemented in practice.

The research questions highlighted in Chapter 1 have been answered as follows:

- **RQ1:** Can a DAG-based protocol implement order fairness? **Yes**, both Tilikum and FairDAG [39] showed that it is possible to integrate fair ordering in a DAG ledger;
- **RQ2:** What is the performance overhead of order fairness on this protocol? The overhead is **considerable**, but better than existing solutions. On average, Tilikum is 10 times slower than Narwhal and Tusk, but 22 times faster than FairDAG;
- **RQ3:** Is this protocol able to defend against DAG-specific reordering attacks, such as the ones from Zhang et al. [83]? **Yes**, the attacks are fully stopped as the order of blocks does not influence the order of execution;
- **RQ4:** How does this protocol compare to FairDAG [39], the state-of-the-art fair ordering solution for DAGs? The **shortcomings** of FairDAG, in particular weak edges and vulnerabilities to malicious clients, have been **fully addressed** in Tilikum.

References

[1] Hayden Adams et al. "Uniswap v3 core". In: *Tech. rep., Uniswap, Tech. Rep.* (2021).

[2] Orestis Alpos et al. *Eating sandwiches: Modular and lightweight elimination of transaction reordering attacks*. 2023. arXiv: 2307.02954 [cs.DC]. URL: https://arxiv.org/abs/2307.02954.

[3] Guillermo Angeris and Tarun Chitra. "Improved price oracles: Constant function market makers". In: *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*. 2020, pp. 80–91.

[4] Kenneth J Arrow. *Social choice and individual values*. Vol. 12. Yale university press, 2012.

[5] Balaji Arun et al. "Shoal++: High throughput dag bft can be fast!" In: *arXiv preprint arXiv:2405.20488* (2024).

[6] David Austen-Smith and Jeffrey S Banks. *Positive political theory I: Collective preference*. Vol. 1. University of Michigan Press, 2000.

[7] Kushal Babel et al. "Mysticeti: Low-latency dag consensus with fast commit path". In: *arXiv preprint arXiv:2310.14821* (2023).

[8] Vivek Bagaria et al. "Prism: Deconstructing the blockchain to approach physical limits". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 2019, pp. 585–602.

[9] Henri Bal et al. "A medium-scale distributed system for computer science research: Infrastructure for the long term". In: *Computer* 49.5 (2016), pp. 54–63.

[10] Iddo Bentov et al. "Tesseract: Real-time cryptocurrency exchange using trusted hardware". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 2019, pp. 1521–1538.

[11] Same Blackshear et al. "Sui lutris: A blockchain combining broadcast and consensus". In: *arXiv preprint arXiv:2310.18042* (2023).

[12] bloXroute. *bloXroute BackRunMe*. 2022. URL: https://docs.bloxroute.com/bsc-and-eth/apis/backrunme (visited on 04/20/2025).

[13] Gabriel Bracha. "Asynchronous Byzantine agreement protocols". In: *Information and Computation* 75.2 (1987), pp. 130–143.

[14] Felix Brandt et al. *Handbook of computational social choice*. Cambridge University Press, 2016.

[15] Martin Brennecke et al. "The de-central bank in decentralized finance: A case study of MakerDAO". In: (2022).

[16] Eric Budish, Peter Cramton, and John Shim. "The high-frequency trading arms race: Frequent batch auctions as a market design response". In: *The Quarterly Journal of Economics* 130.4 (2015), pp. 1547–1621.

[17] Christian Cachin et al. "Quick order fairness". In: *International Conference on Financial Cryptography and Data Security*. Springer. 2022, pp. 316–333.

[18] Jo-Mei Chang and Nicholas F. Maxemchuk. "Reliable broadcast protocols". In: *ACM Transactions on Computer Systems (TOCS)* 2.3 (1984), pp. 251–273.

[19] Hao Chung et al. "Rapidash: Foundations of side-contract-resilient fair exchange". In: *Cryptology ePrint Archive* (2022).

[20] Michele Ciampi, Aggelos Kiayias, and Yu Shen. "Universal composable transaction serialization with order fairness". In: *Annual International Cryptology Conference*. Springer. 2024, pp. 147–180.

[21] CowSwap. *CowSwap Exchange*. 2024. URL: https://swap.cow.fi (visited on 06/12/2024).

[22] Philip Daian et al. "Flash boys 2.0: Frontrunning, transaction reordering, and consensus instability in decentralized exchanges". In: *arXiv preprint arXiv:1904.05234* (2019).

[23] George Danezis et al. "Narwhal and tusk: a dag-based mempool and efficient bft consensus". In: *Proceedings of the Seventeenth European Conference on Computer Systems*. 2022, pp. 34–50.

[24] Shayan Eskandari, Seyedehmahsa Moosavi, and Jeremy Clark. "Sok: Transparent dishonesty: front-running attacks on blockchain". In: *Financial Cryptography and Data Security: FC 2019 International Workshops, VOTING and WTSC, St. Kitts, St. Kitts and Nevis, February 18–22, 2019, Revised Selected Papers 23*. Springer. 2020, pp. 170–189.

[25] Ethereum. *Ethereum Proposer-builder separation*. 2024. URL: https://ethereum.org/en/roadmap/pbs (visited on 06/12/2024).

[26] Flashbot. *Flashbot MEV Explore*. 2024. URL: https://explore.flashbots.net/ (visited on 06/12/2024).

[27] Flashbots. *Flashbots Auction*. 2024. URL: https://docs.flashbots.net/flashbots-auction/overview (visited on 06/12/2024).

[28] Flashbots. *Flashbots Protect*. 2024. URL: https://protect.flashbots.net/ (visited on 06/12/2024).

[29] Flashbots. *MEV-Boost*. 2024. URL: https://github.com/flashbots/mev-boost (visited on 06/12/2024).

[30] Flashbots. *MEV-Share*. 2024. URL: https://docs.flashbots.net/flashbots-protect/mev-share (visited on 06/12/2024).

[31] William V Gehrlein. "Condorcet's paradox". In: *Theory and decision* 15.2 (1983), pp. 161–197.

[32] Martin D Gould et al. "Limit order books". In: *Quantitative Finance* 13.11 (2013), pp. 1709–1742.

[33] Vincent Gramoli et al. "AOAB: optimal and fair ordering of financial transactions". In: *2024 54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE. 2024, pp. 377–388.

[34] Suyash Gupta et al. "Resilientdb: Global scale resilient blockchain fabric". In: *arXiv preprint arXiv:2002.00160* (2020).

[35] Lioba Heimbach and Roger Wattenhofer. "Eliminating sandwich attacks with the help of game theory". In: *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*. 2022, pp. 153–167.

[36] Lioba Heimbach and Roger Wattenhofer. "Sok: Preventing transaction reordering manipulations in decentralized finance". In: *Proceedings of the 4th ACM Conference on Advances in Financial Technologies*. 2022, pp. 47–60.

[37] Maurice P Herlihy and Jeannette M Wing. "Linearizability: A correctness condition for concurrent objects". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12.3 (1990), pp. 463–492.

[38] Philipp Jovanovic et al. "Mahi-mahi: Low-latency asynchronous bft dag-based consensus". In: *arXiv preprint arXiv:2410.08670* (2024).

[39] Dakai Kang et al. *FairDAG: Consensus Fairness over Concurrent Causal Design*. 2025. arXiv: 2504.02194v1 [cs.DB]. URL: https://arxiv.org/abs/2504.02194v1.

[40] Idit Keidar et al. "All you need is DAG". In: *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*. 2021, pp. 165–175.

[41] Mahimna Kelkar, Soubhik Deb, and Sreeram Kannan. "Order-fair consensus in the permissionless setting". In: *Proceedings of the 9th ACM on ASIA Public-Key Cryptography Workshop*. 2022, pp. 3–14.

[42] Mahimna Kelkar et al. "Order-fairness for byzantine consensus". In: *Advances in Cryptology–CRYPTO 2020: 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17–21, 2020, Proceedings, Part III 40*. Springer. 2020, pp. 451–480.

[43] Mahimna Kelkar et al. "Themis: Fast, strong order-fairness in byzantine consensus". In: *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 2023, pp. 475–489.

[44] Rami Khalil, Arthur Gervais, and Guillaume Felley. *TEX-A Securely Scalable Trustless Exchange*. Cryptology ePrint Archive, Paper 2019/265. 2019. URL: https://eprint.iacr.org/2019/265.

[45] Klaus Kursawe. "Wendy Grows Up: More Order Fairness". In: *Financial Cryptography and Data Security. FC 2021 International Workshops - CoDecFin, DeFi, VOTING, and WTSC, Virtual Event, March 5, 2021, Revised Selected Papers*. Ed. by Matthew Bernhard et al. Vol. 12676. Lecture Notes in Computer Science. Springer, 2021, pp. 191–196. DOI: 10.1007/978-3-662-63958-0_17. URL: https://doi.org/10.1007/978-3-662-63958-0%5C_17.

[46] Klaus Kursawe. "Wendy, the Good Little Fairness Widget: Achieving Order Fairness for Blockchains". In: *AFT '20: 2nd ACM Conference on Advances in Financial Technologies, New York, NY, USA, October 21-23, 2020*. ACM, 2020, pp. 25–36. DOI: 10.1145/3419614.3423263. URL: https://doi.org/10.1145/3419614.3423263.

[47] Razya Ladelsky and Roy Friedman. "On Quorum Sizes in DAG-Based BFT Protocols". In: *arXiv preprint arXiv:2504.08048* (2025).

[48] Lindsay X Lin et al. "Deconstructing decentralized exchanges". In: *Stanford Journal of Blockchain Law & Policy* 2.1 (2019), pp. 58–77.

[49] Erwan Mahe and Sara Tucci-Piergiovanni. "Order Fairness Evaluation of DAG-based ledgers". In: *arXiv preprint arXiv:2502.17270* (2025).

[50] MakerDAO. *MakerDAO Documentation*. 2024. URL: https://docs.makerdao.com/smart-contract-modules/dog-and-clipper-detailed-documentation (visited on 06/12/2024).

[51] Dahlia Malkhi and Pawel Szalachowski. "Maximal extractable value (mev) protection on a dag". In: *arXiv preprint arXiv:2208.00940* (2022).

[52] Jerry W Markham. "Front-running-insider trading under the commodity exchange act". In: *Cath. UL Rev.* 38 (1988), p. 69.

[53] Ciamac C Moallemi and Utkarsh Patange. "An Analysis of Fixed-Spread Liquidation Lending in DeFi". In: *Proceedings of The 4th Workshop on Decentralized Finance (DeFi) In Association with Financial Cryptography*. 2024.

[54] Imad Moosa. "The regulation of high-frequency trading: A pragmatic view". In: *Journal of Banking Regulation* 16 (2015), pp. 72–88.

[55] Ke Mu et al. "Separation is good: A faster order-fairness Byzantine consensus". In: (2024).

[56] Satoshi Nakamoto. "Bitcoin: A peer-to-peer electronic cash system". In: (2008).

[57] Bulat Nasrulin et al. "LO: An Accountable Mempool for MEV Resistance". In: *Proceedings of the 24th International Middleware Conference*. 2023, pp. 98–110.

[58] Shutter Network. *Shutter Network*. 2024. URL: https://shutter.network (visited on 06/12/2024).

[59] Osmosis. *Osmosis*. 2024. URL: https://osmosis.zone (visited on 06/12/2024).

[60] Skip Protocol Osmosis Labs. *Osmosis Protorev by Skip Protocol*. 2023. URL: https://forum.osmosis.zone/t/osmosis-protorev-by-skip-protocol/604 (visited on 04/20/2025).

[61] Chris Piatt, Jeffrey Quesnelle, and Caleb Sheridan. "Eden network". In: (2021).

[62] Nikita Polyanskii, Sebastian Mueller, and Ilya Vorobyev. "Starfish: A high throughput BFT protocol on uncertified DAG with linear amortized communication complexity". In: *Cryptology ePrint Archive* (2025).

[63] Serguei Popov. "The tangle". In: *White paper* 1.3 (2018), p. 30.

[64] Kaihua Qin et al. "An empirical study of defi liquidations: Incentives, risks, and instabilities". In: *Proceedings of the 21st ACM Internet Measurement Conference*. 2021, pp. 336–350.

[65] Patrick Schueffel. "DeFi: Decentralized finance-an introduction and overview". In: *Journal of Innovation Management* 9.3 (2021), pp. I–XI.

[66] Gregory Scopino. "The (questionable) legality of high-speed pinging and front running in the futures market". In: *Conn. L. Rev.* 47 (2014), p. 607.

[67] Sikka. *Sikka Threshold Encryption*. 2024. URL: https://sikka.tech/projects (visited on 06/12/2024).

[68] Yonatan Sompolinsky, Yoad Lewenberg, and Aviv Zohar. "Spectre: A fast and scalable cryptocurrency protocol". In: *Cryptology ePrint Archive* (2016).

[69] Yonatan Sompolinsky and Aviv Zohar. "Secure high-rate transaction processing in bitcoin". In: *Financial Cryptography and Data Security: 19th International Conference, FC 2015, San Juan, Puerto Rico, January 26-30, 2015, Revised Selected Papers 19*. Springer. 2015, pp. 507–527.

[70] Alexander Spiegelman et al. "Bullshark: Dag bft protocols made practical". In: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 2022, pp. 2705–2718.

[71] Alexander Spiegelman et al. "Shoal: Improving dag-bft latency and robustness". In: *International Conference on Financial Cryptography and Data Security*. Springer. 2024, pp. 92–109.

[72] Nick Szabo. "The idea of smart contracts". In: *Nick Szabo's papers and concise tutorials* 6.1 (1997), p. 199.

[73] Mohammad Amin Vafadar and Majid Khabbazian. "Condorcet Attack Against Fair Transaction Ordering". In: *5th Conference on Advances in Financial Technologies*. 2023.

[74] Sarisht Wadhwa et al. "He-htlc: Revisiting incentives in htlc". In: *Cryptology ePrint Archive* (2022).

[75] Qin Wang et al. "Sok: Dag-based blockchain systems". In: *ACM Computing Surveys* 55.12 (2023), pp. 1–38.

[76] Ye Wang et al. "Cyclic arbitrage in decentralized exchanges". In: *Companion Proceedings of the Web Conference 2022*. 2022, pp. 12–19.

[77] Fredrik Winzer, Benjamin Herd, and Sebastian Faust. "Temporary censorship attacks in the presence of rational miners". In: *2019 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. IEEE. 2019, pp. 357–366.

[78] Gavin Wood et al. "Ethereum: A secure decentralised generalised transaction ledger". In: *Ethereum project yellow paper* 151.2014 (2014), pp. 1–32.

[79] Shaokang Xie et al. *Fides: Scalable Censorship-Resistant DAG Consensus via Trusted Components*. 2025. arXiv: 2501.01062 [cs.DC]. URL: https://arxiv.org/abs/2501.01062.

[80] Sen Yang et al. "Sok: Mev countermeasures: Theory and practice". In: *arXiv preprint arXiv:2212.05111* (2022).

[81] Maofan Yin et al. "HotStuff: BFT consensus with linearity and responsiveness". In: *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*. 2019, pp. 347–356.

[82] Pouriya Zarbafian and Vincent Gramoli. "Lyra: Fast and scalable resilience to reordering attacks in blockchains". In: *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE. 2023, pp. 929–939.

[83] Jianting Zhang and Aniket Kate. *No Fish Is Too Big for Flash Boys! Frontrunning on DAG-based Blockchains*. Cryptology ePrint Archive, Paper 2024/1496. 2024. URL: https://eprint.iacr.org/2024/1496.

[84] Yi Zhang, Xiaohong Chen, and Daejun Park. "Formal specification of constant product (xy= k) market maker model and implementation". In: *White paper* (2018).

[85] Yunhao Zhang et al. "Byzantine ordered consensus without byzantine oligarchy". In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 2020, pp. 633–649.

[86] Zibin Zheng et al. "An overview on smart contracts: Challenges, advances and platforms". In: *Future Generation Computer Systems* 105 (2020), pp. 475–491.