Implementing and Understanding Performance of Fino

Siddhant Sharma
University of California, Berkeley

Chris Liu
University of California, Berkeley

Neil Giridharan
University of California, Berkeley

Abstract

BFT-SMR has traditionally low throughput and high message communication overhead. However, recent research advances DAG-based BFT protocols that decouple reliable message broadcast and transaction ordering to achieve high throughput for additional latency overhead. Fino [13], by Malkhi and Szalachowski, is a novel DAG-BFT protocol that emphasizes simplicity while trying to achieve high performance for throughput and latency. Protocols such as Narwhal-Bullshark [10] explore this tradeoff space as well, achieving impressive performance in the happy case. In the case of network latency or Byzantine leaders, Bullshark can face high latency, as nodes are often blocked awaiting progress. Fino takes an approach of integrating timeouts into its messages to promote view changes, leading to questions about performance between the two protocols in various workloads. In this paper, we implement Fino with the Narwhal broadcast layer. We observe how this affects performance by running benchmarks against other DAG-BFT protocols across various scenarios and workloads. Furthermore, we develop an end-to-end evaluation framework to provide insight on the holistic performance of consensus protocols when applied to applications with transaction execution—we find this reveals more nuanced behavior than would be possible by only observing maximum throughput and minimum latency data.

1 Introduction

BFT-SMR has been a major point of focus in recent blockchain scalability efforts, as the bottleneck for many chains has been the core consensus layer. Consensus protocols have evolved from traditional PBFT [3] to have nuanced properties and design decisions for different use-cases, but the throughput and latency are low (<100 tx/second). Certain blockchains, such as Ethereum [19], try to maintain larger validator sets, leading to poor throughput (7 tx/second) and slow block times (12 seconds per block). This is often due to the complex coupling of message broadcasting and poor pipelining, leading to very "sequential" periods of consensus ordering.

Protocols such as *HotStuff* [1] reduce the message complexity required for leaders to communicate with participants to commit and order new transactions and even offer avenues for pipelining to increase throughput. HotStuff achieves this through the notion of quorum certificates, which are used to guarantee safety. However, HotStuff still does not maximize the network throughput, as non-leader nodes do not produce any transactions.

Narwhal and Tusk [5] introduces a novel decoupling of transaction/block gossiping and transaction ordering, that allow Narwhal-Tusk and Narwhal-HotStuff to achieve over 150k tx/second and 125k tx/second respectively. Narwhal provides an underlying DAG structure that allows all nodes to produce uncommitted blocks that are eventually committed when a leader proposes a block and orders the block's causal history through a topological sort. This allows the network to produce blocks at network speed through zero consensus overhead on top of the message complexity! Further improvements such as Bullshark [10] introduce better asynchrony and latency guarantees through concepts of physical DAGs (formed in Narwhal's messaging layer) and logical DAGs (built on top of the physical DAG) used for consensus.

DAG-BFT protocols such as *Bullshark* and *Tusk* make improvements on the latency guarantees of *HotStuff* (under partial synchrony and eventual synchrony), but can still run into periods of high latency when nodes do not receive messages from leaders to advance rounds or when leaders do not receive enough votes on proposals. In production deployments, this is extremely problematic, as we can lose liveness for extended periods.

Fino [13] introduces an additional message type, timeouts, integrated within the message DAG for BFT protocols. This allows nodes to broadcast messages indicating they have timed out on the current round and are progressing forward in their local DAG given enough support from peers. The network can proceed with bounded waiting periods, without blocking, in cases where leaders are unresponsive/Byzantine or when there is widespread network latency or packet drops. Theoretically, the throughput and latency guarantees are asymptotically the same, but the performance in practice should be significantly better due to nodes not blocking on missing actions from other nodes. Papers such as *MEV Protection on a DAG* [13] introduce this notion of "integrated" timeouts, but do not provide implementations with throughput or latency benchmarks against other consensus protocols. Comparing the throughput and latency performance of timeouts in logical DAGs against other DAG-BFT protocols can shed light on the performance differences between physical and logical DAG-BFT consensus protocols.

2 BFT Definitions

2.1 Model

For simplicity, our system contains n nodes $\Pi = \{p_1, p_2, \cdots, p_n\}$ that participate in state machine replication. Up to $f < \frac{n}{3}$ nodes can be Byzantine. They may act arbitrarily and can implement any adversarial behavior (such as crashing or delaying, duplicating, dropping, and modifying messages) but cannot subvert standard cryptographic methods. To allow nodes to use these cryptographic methods, we assume the existence of public-key infrastructure (PKI) that provides asymmetric encryption, signing, and message integrity functionality. Byzantine nodes are not necessarily independent, and can therefore collude. We say that all nodes that are not Byzantine are *correct*. We assume all nodes are connected and send messages to each other over a partially synchronous network, where the network is asynchronous up to an unknown global stabilization time (GST), after which there is a known bound Δ on message delays between nodes. Furthermore, we assume that all messages between correct nodes are eventually delivered. In essence, we assume the same Byzantine, networking, and cryptographic models as Bullshark and similar protocols to ensure our comparisons are fair.

2.2 Required Infrastructure

Many relevant BFT protocols, including Fino, involve the following known abstractions:

Reliable Broadcast: A node p_k can call reliable broadcast, $r\_bcast_k(m,r)$, to broadcast a message m during round $r \in \mathbb{N}$ of the protocol. Each node p_i also has an output channel $r\_deliver_i(m,r,p_k)$ for a message m, a round number r, and a node p_k which called the corresponding $r\_bcast_k(m,r)$. Reliable broadcast guarantees agreement, integrity, and validity—we will not define them formally, as we inherit them from Narwhal. Agreement allows all correct nodes to eventually output the same $r\_deliver(m,r,p_k)$ values. Integrity guarantees that each correct node will only output $r\_deliver(m,r,p_k)$ at most once each round. Validity ensures that every correct node eventually outputs $r\_deliver(m,r,p_k)$ as a result of a correct node calling $r\_bcast_k(m,r)$. For our implementation, we use Narwhal's [5] reliable broadcast functionality for fair evaluations between Fino and Bullshark.

Global perfect coin: A node $p_i \in \Pi$ can randomly select a node $p_j \in \Pi$ by invoking $choose\_leader_i(w)$ on an instance of a global perfect coin $w \in \mathbb{N}$; the outcome of which can be represented by the random variable X_w. The global perfect coin guarantees agreement, termination, unpredictability, and fairness. We will not define them formally, as we inherit them from Narwhal. Agreement allows two correct nodes calling $choose\_leader(w)$ to receive the same values. Termination guarantees if at least $f+1$ correct nodes call the function, then the function will eventually return. Unpredictability allows for the probability of an adversary guessing the return value of p_j to be equal to indistinguishable from a random value within a negligible range. Fairness allows for the probability of a node being picked to be $\frac{1}{n}$ for a system with n nodes.

2.3 Problem Definition

Fino aims to solve the problem of reaching consensus on a total ordering of all transactions in the face of Byzantine failures for all correct nodes. Specifically, if a correct node orders message m_1 before m_2, then no correct node can order message m_2 before m_1. Fino relies on a DAG transport layer to communicate with peers and maintain a DAG view for each node in the system.

Our goal is to implement the Fino protocol, using Narhwal as the underlying DAG transport protocol, to understand its performance in a real-world system when compared to other BFT protocols. Fino's introduction of timeouts into its DAG structure promises to reduce latency under certain node behavior, but its effects in practice are currently unknown. Therefore, our goal is also to measure the impact of including timeouts on real-world, practical workloads. By using Narwhal as the transport layer, we can leverage a well-tested messaging layer that has been used in research and production environments. Implementing and benchmarking Narwhal-Fino allows us to understand the class of DAG-BFT protocols that use a logical layering on top of the messaging better on latency and throughput metrics.

3 Related Work

3.1 Narwhal

Narwhal [5] is a mempool protocol that introduces a key insight of decoupling message gossiping and message sequencing. It handles the dissemination of transactions, grouped into blocks, introducing a DAG-based structure to manage the casual history of these blocks. Narwhal finds performance gains by only performing consensus on a small amount of metadata; it also avoids wasteful messaging found in many traditional

BFT protocols by guaranteeing that all proposed blocks are eventually committed.

3.1.1 Relevant Properties

Narhwal assumes a system of n nodes tolerant of suffering f Byzantine failures, such that $n \geq 3f+1$. Its provides a performant mempool and a store of uncommitted transactions, revolving around a key-value store of blocks b keyed by their digest d. It provides the following properties:

- **Integrity**: Correct nodes reading from the same digest d will receive the same block b.
- **Block-Availability**: Reading a digest d after a successful key-value write (d,b) on a correct node must eventually return b.
- **Containment** The causal history of a later block always contains that of an earlier block.
- **2/3-Causality** The causal history of a block contains $\geq \frac{2}{3}$ of the previous blocks.
- **1/2-Chain Quality** At least $\frac{1}{2}$ of a block's causal history is written by correct nodes.

3.1.2 Replication Protocol

Narhwal implements a novel reliable broadcast protocol, introducing the notion of a *certificate of availability*. In Narwhal's DAG, blocks comprise of a hash signature, a list of transactions, as well as a collection of certificates of availability for the previous round's block. Certificates of availability are comprised of the hash of its corresponding block alongside $2f+1$ signatures from other nodes during the current round r, verifying the block's successful delivery. The gossip protocol goes as follows:

1. Nodes create a transaction list using incoming client transactions and a certificate list from incoming certificates.
2. Upon receiving $2f+1$ certificates from other nodes within a round $r-1$, a node moves into the next round r, adding its transaction list to a new block and broadcasting it to other nodes.
3. Incoming blocks are validated by the node to ensure they contain a $2f+1$ certificate from the previous round r; the node signs the hash with its own signature if this is the case.
4. Nodes create a certificate of availability once it has received $2f+1$ signatures for a block it broadcasted. Once a certificate of availability has been created, the node broadcasts it to all other nodes and stops broadcasting the original block.

Narhwal and Tusk [5] follows up this message dissemination protocol with its own consensus protocol named Tusk; which is responsible for actually committing blocks.

3.2 Bullshark

Bullshark [10] is a BFT protocol that introduces additional metadata into Narwhal's messaging layer to support consensus. Specifically, each block proposed during round r in Narwhal's DAG additionally includes a vote for blocks proposed during round $r-1$. When a block receives $f+1$ votes, it and its causal history are ordered and committed. Additionally, after $f+1$ blocks are proposed during a round, the block proposed by the leader of that round (known as the *anchor block*) and its causal history are also ordered and committed. When a node times out while waiting for other nodes to send in votes during a round, it "short-circuits" and sends out its current information on other nodes' votes alongside its current transaction block to be committed later.

```
$p \leftarrow$ current node
$f \leftarrow$ # of Byzantine failures
$dag \leftarrow$ the current DAG
function try_commit($v$: Vertex):
    $r \leftarrow$ get_round_number($v$)
    if $r$ is even then
        if $v$.author $= p \wedge p$ is leader for round $r$ then
            $stake \leftarrow 0$
            for message $m \in dag$.get($r-1$) do
                if $m$.parent $= v$ then
                    $stake \leftarrow$
                      $stake +$ get_stake($r$.author)
                end
            end
            if $stake > f+1$ then
                $ls \leftarrow dag$.order_causal_history($v$)
                commit_blocks($ls$)
            end
        end
    end
end
```
Algorithm 1: Bullshark Commit Protocol

Leaders only have the potential to commit blocks during even rounds—odd rounds are used to determine which block will be voted upon. To reach consensus on a block, the leader calculates its *stake*, representing how many nodes have voted in approval of the block being proposed during the current round. If the stake exceeds the validity threshold of $f+1$, the leader orders the DAG vertices in the block's causal history and commits.

3.3 Fino

Fino [13] is described as a BFT protocol introducing the notion of a *logical DAG* on top of Narwhal to achieve high transaction throughput. Each round of the protocol, nodes gossip blocks and collect signatures on these blocks as per the Narwhal protocol. Separate from Narwhal's DAG composed of the blocks being gossiped (otherwise known as the *physical DAG*), the logical DAG introduced by Fino is composed of messages voting on these gossiped blocks. This allows the protocol to advance the physical DAG at network speed, independent of the consensus logic for the logical DAG. This implies the logical layer of the DAG is a subset of the physical DAG, but committing blocks from the logical DAG will eventually commit all blocks on the physical DAG.

Fino relies on *vote*, *propose*, and *complain* (otherwise known as *timeout*) messages to form the logical DAG responsible for managing causal history and committing blocks. Specifically, its classification of timeouts as part of logical messages allows nodes to communicate with each other about moving on from rounds with unresponsive or Byzantine leaders without suffering performance losses. Asymptotically, Fino has the same messaging complexity as Bullshark, but Fino attempts to take a different approach to slow-view changes when leaders or peers are unresponsive.

```
$p \leftarrow$ current node
$f \leftarrow$ # of Byzantine failures
$dag \leftarrow$ the current DAG
function try_commit(v: Vertex):
    $r \leftarrow$ get_round_number($v$)
    if $r$ is even then
        if $v$.author $= p \wedge p$ is leader for round $r$ then
            $stake \leftarrow 0$
            for message $m \in dag$ .get($r-1$) do
                if $m$.parent $= v \wedge m$ is a vote then
                    $stake \leftarrow$
                      $stake +$ get_stake($r$.author)
                end
            end
            if $stake > f+1$ then
                $ls \leftarrow dag$.order_causal_history($v$)
                commit_blocks($ls$)
            end
        end
    end
end
```

Algorithm 2: Fino Commit Protocol. The key difference between Fino and Bullshark's commit is using the stake of only `vote` messages to commit.

The commit protocol for Fino is very similar to that of Bullshark, but with the key difference that it must differentiate between vote and complaint messages within a round when computing stake. Similarly to Bullshark, the leader orders and commits if the number of `votes` exceeds the $f+1$ validity threshold. However, unlike Bullshark's "short-circuit" timeouts, nodes only move on to the next round if the number of complaints within a round exceeds the quorum threshold of $2f+1$.

4 Implementing Fino

4.1 Integration with Narwhal-Bullshark

By itself, Fino is a simple consensus protocol. There are only 2 forms of consensus messages sent, with very simple data structures and data types, such as signed integers, message digests, and asymmetric cryptographic signatures. The protocol's cryptographic properties are guaranteed via our original model and PKI assumptions. However, the Fino protocol requires a data dissemination layer to send its logical DAG metadata within messages. In Malhki's original work on Fino, this is a generalized DAG-transport protocol that contains simple `broadcast` and `deliver` APIs. In our work, we rely on Narwhal to provide reliable broadcast and DAG-transportation. We choose Narwhal due to its testing in real world, production-ready applications such as Sui [14] and Aptos [2], as well as its well-designed architecture that can scale system throughput quasi-linearly with respect to the number of data-dissemination workers per node. To reach consensus on top of Narwhal messaging, we attach Fino's logical messages and implement the Fino consensus protocol.

4.1.1 Modifying Narwhal

Implementing Fino on top of Narwhal requires modifying some of Narwhal's core data structures. To minimize the code changes required, we added a `Decision` enum to represent the logical DAG decision that participants in the network make based on gossip information they are processing. `Votes`, `Complains`, and `Proposes` all have `Rounds` attached to the enum, specifying which logical round number the decision is for. If the header is created just for the physical layer of the DAG (essentially, for Narwhal), the `Decision` holds `Empty`, with no associated logical round number. All other core data structures between Fino and Narwhal remain unchanged. We maintain as small of a code diff between the two implementations to ensure we do not infringe on Narwhal's properties, such as garbage collection, or insert unnecessary metadata or data structures that can affect performance.

Integrating Fino's logical timer alongside Narwhal's system is tricky due to potential blocking and race conditions between physical and logical blocks, as well as other components of the Narwhal system. Narwhal's architecture contains a `Core` module that orchestrates Tokio (Rust asynchronous scheduling) [17] handles and interactions upon receiving votes, cer-

```
Enum Decision options
    Vote(Round);
    Complain(Round);
    Propose(Round);
    // default
    Empty;
end
Struct Header contains
    PublicKey author;
    Decision decision;
    Map<Digest, WorkerId> payload;
    Set<Digest> parents;
    Digest id;
    Signature signature;
end
Struct Certificate contains
    Header header;
    Vec<(PublicKey, Signature)> votes;
end
```

Algorithm 3: Core Data Structures for Fino

tificates, and headers. The `Core` dispatches actions to the `Proposer` module to create new headers and dispatch them to other nodes, among other intermediate steps. Therefore, the Narwhal `Primary`'s `Core` module is the perfect place to integrate the Fino abstraction of a logical timer. We instrument the `Core` module with a timer to keep track of time since the last view change. When a header is received for a given round, we add it to each node's count of `Ok` or `Complain` for the current logical round. Then, if a node has received more than $f+1$ `Ok` or $2f+1$ `Complain` of validator stake for a logical round, the quorum of nodes can view change. When a node receives a `Propose`, it checks for its logical timer expiration. If the timer has not expired, the node responds with a `Ok` to indicate a vote. If the timer has expired, it broadcasts a `Complain` for the given logical round. The value that the `Core` decides to broadcast via the `Proposer` module is sent via a Tokio channel. The `Proposer` is modified to listen to this additional data from the `Core` in its execution loop. Upon receiving the next view's decision from the `Core`, the proposer attaches it to the next set of headers it processes for the given logical view.

By separating the logical timer from the proposer, we discover Fino has potential performance loss due to synchronization issues between the `Core` and `Proposer` modules. As a toy example, consider the case where the `Core` module is processing headers and certificates, but the `Proposer` has just sent out a header. The Fino decision sent by the `Core` would need to wait for the next message to arrive, be processed, and finally be sent out. This can add significant latency per header and lead to some Fino decisions to be discarded as they are stale, as the logical DAG may have progressed to the next view and the header would no longer be needed. A nuance of this is that it interpolates the trade-off space between latency and throughput through such delays. Since the underlying DAG transport protocol running underneath Fino, Narwhal, continues to progress at network speed, we simply add additional latency before we commit. As a result, each consensus commit contains more transactions but incurs extra latency. This can occur in certain tail cases under the correct network delays or message receiving times. It is important to note this is not a correctness issue and will not lead to liveness issues in the partial synchrony model that Fino follows.

4.1.2 Implementing Fino's Consensus Protocol

Implementing Fino's consensus is fairly similar to Bullshark's implementation in Narwhal-Bullshark. The core functional difference is instead of checking for $f+1$ stake from parents in Bullshark, Fino instead checks for $f+1$ stake of `Vote` votes to commit for a given round. Similarly, Fino only commits at the logical layer, but each commit contains the messages and blocks gossiped from the physical layer of Narwhal. All operations in Fino's consensus protocol works over the logical DAG, unlike Bullshark integrating all information into the physical layer via Narwhal.

4.2 Integration with ABCI

Application Blockchain Interface (ABCI) [16] is a framework that provides a standardized interface between the consensus and mempool layer of blockchains to the applications that run on blockchains. ABCI has a strong focus on modularity and interoperability, making it a great candidate for developing a framework of macrobenchmarking different systems by easily swapping out consensus protocols, mempools, and execution environments. ABCI is used in production in various blockchains today, primarily found in the Cosmos [4] ecosystem.

We developed a framework using ABCI that allows for benchmarking by integrating generalized execution environments via RPC, building off the work developed by Konstantopoulos, et al. in their article *Cosmos without Tendermint: Exploring Narwhal and Bullshark* [12]. We allow clients to interact with an application by sending RPC requests to a shim residing at the consensus and data availability layer of ABCI. The shim can route requests directly to the peer-to-peer layer or consensus layer for message routing or ordering (depending on the consensus protocol of choice) or forward the request to the execution layer of the application. Upon ordering of transactions, the ABCI framework can execute newly produced blocks using an execution environment of choice and propagate state changes to clients. The communication between data availability layer and execution layer are also facilitated through RPC.

To integrate the Narwhal family of BFT protocols, we can plug in Narwhal as our data availability layer for gossiping with peers and storing our ledger. This includes the Narwhal

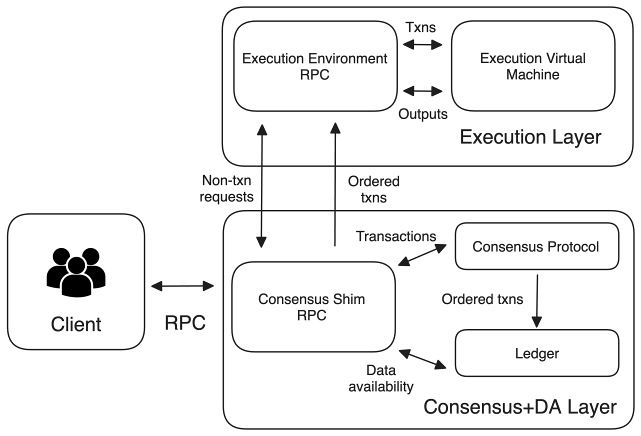

Figure 1: End-to-End Macrobenchmarking System Overview, generalizing the work from [12].

primary and any workers used to scale the block gossiping for the desired end-to-end system load. To integrate the consensus layer, we can spin up our primary with the desired consensus protocol. The primary can be configured with the appropriate networking to its workers and other services such as signature aggregators or databases. The process of configuring the data availability and consensus protocol is designed to be as similar to running other benchmarks with the Narwhal system, for ease of use and feature-parity.

To integrate an execution environment, the execution environment needs to spawn as an RPC server and route requests to its internal services, such as the execution virtual machine, the execution state handlers, and more. For our benchmarking, we integrated the Ethereum Virtual Machine (EVM) [19] with its corresponding Ethereum execution RPC[1] to ABCI. The Ethereum execution environment RPC gets requests routed from the client via the consensus shim to compute client-specific requests or execute new blocks of transactions for state-transition computation. We must route the client requests via the consensus shim, as we need to determine which requests are new transactions for the mempool and which requests are directed for the execution environment. We can extend our work by testing various execution environments that have different programming and execution paradigms, such as Block-STM [9] designed for production-ready implementations of Narwhal-Bullshark called Aptos [2]. To integrate the Ethereum Virutal Machine, we make use of Anvil[2], the local node development environment in the Foundry[3] family of Ethereum development tools. We will use this end-to-end framework to benchmark various consensus protocols, ranging from Bullshark to HotStuff to Fino to understand throughput and latency under various workloads and network configurations.

[1]Ethereum execution RPC spec can be found at https://github.com/ethereum/execution-apis

[2]Anvil can be found at https://github.com/foundry-rs/foundry/tree/master/crates/anvil

[3]Foundry can be found at https://github.com/foundry-rs/foundry/

5 Evaluation

5.1 Evaluation Methodology

We compare Fino against other BFT protocols to understand how it compares against Bullshark and HotStuff. Bullshark is the most similar BFT protocol, as it builds on top of Narwhal and is another zero-message overhead protocol, making it the focus of our evaluations. We use three main methods of benchmarking: simulated latency, cloud benchmarking, and macro-benchmarking by attaching the Ethereum Virtual Machine's execution environment to various consensus protocols. We execute all non-cloud benchmarks on Apple M1 Max machines with 64 GB of memory and 1 TB of storage, allowing for ample machine resources to not bottleneck our testing.

We employ cloud benchmarks to test the performance of Fino against Bullshark in a wide area network (WAN) on Google Cloud Platform. We test 5 runs under the simple configuration of 4 honest nodes to get an idea of the limits of the system under heavy load from clients. We can understand how the systems perform under realistic workloads and geographically distributed nodes and clients.

The simulated latency benchmark simulates network congestion across different committee configurations and average latency. We use a Weibull distribution to model the latency between peers communicating between each other. Weibull distributions have two parameters, scale and shape. We modify the shape parameter to simulate different average latency values. We also benchmark performance using a bimodal distribution of latency to model happy and sad case execution paths for modeling the network and understanding how the consensus throughput and latency are affected in such cases. In such tests, we use an input transaction rate of 50,000 tx/s, with a maximum block size of 500KB and transaction size of 512B.

Finally, we employ our end-to-end system benchmarking methodology described in Section 4.2 to get a better understanding of how end-to-end (as well as consensus) throughput and latency are affected when committing and executing operations within each block. Specifically, we use the Ethereum Virtual Machine (EVM) and its RPC protocol to execute the transactions ordered by each consensus protocol. We use simple transactions, such as simple interactions with other accounts on the EVM, but also have the option to load in additional state, such as ABCI modules or Ethereum smart contracts, to simulate more complex interactions and transactions. For end-to-end testing, we use an input rate of 10,000 tx/s, with a maximum block size of 500KB. It is difficult to estimate the exact transaction size, as there are several variable factors that can increase transaction size, such as gas prices and fees. However, a reasonable lower bound is 100B, given the required fields in each Ethereum transaction and the recursive length prefix encoding algorithm used to serialize data structures.

5.2 Cloud Microbenchmarks

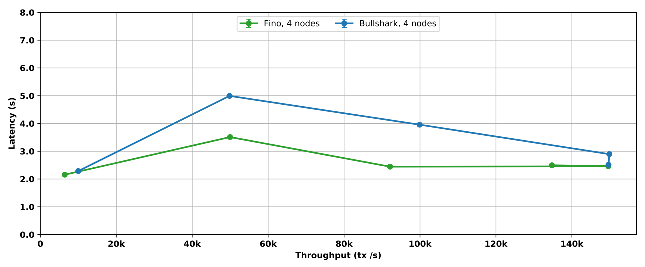

Figure 2: Cloud benchmarks of Fino and Bullshark, running in a GCP Compute Engine using a wide area network

For our cloud benchmarks, we leveraged Google Cloud Platform's Compute Engine, using N2 machines with 4 vCPUs and 16GB of RAM. We use this setup to simulate an average node's hardware requirements to understand real-world performance. We use a wide area network (WAN) to simulate geographically distributed nodes to account for a diverse validator set. We executed the benchmark over 5 runs, using input transaction rates of 15,000, 50,000, 100,000, and two runs of 150,000 transactions per second. From the graphs, we see that Fino and Bullshark have very similar performance, where Fino has a slightly lower throughput in almost every input rate but trades it off for slightly lower latency.

Fino achieves about 11,000 tx/s and 2.1s of latency for 15,000 tx/s input rate, which is slightly less than Bullshark's 14,500 tx/s and 2.3s of latency. Fino achieves 48,500 tx/s and 3.5s of latency, whereas Bullshark achieves 48,000 tx/s and 5.0s of latency. This helps highlight in certain happy paths, Fino can achieve better latency than Bullshark. For input 100,000 tx/s, Fino achieves 93,000 tx/s and 2.2s of latency whereas Bullshark achieves 99,000 tx/s and 4.0s of latency. Once again, we see the logical DAG implemented by Fino trades off latency for throughput. In the 150,000 tx/s case, we can see some hints of the synchronization problem Fino and other logical DAG-BFT protocols can face. In one run, we see that Fino and Bullshark achieve almost the exact same throughput and latency of 149,000 tx/s and 2.5s of latency. However, in another run, Fino achieves only 135,000 tx/s with 2.6s of latency, whereas Bullshark achieves 148,500 tx/s and 2.9s of latency. Here, we see that Fino achieves a significantly worse throughput for slightly better latency. We believe this is due to Fino initially missing commits due to network latency, causing the Fino logical timers to mis-synchronize with Narwhal's timers. Then, Fino's next set of decisions tend to be slightly mis-synchronized with Narwhal, causing commits to happen with less latency but collecting fewer transactions to commit in the same time period. Bullshark shines in this example, as it is guaranteed to operate on the same timers as Narwhal, since it does not maintain its own timer state.

5.3 Simulated Latency Microbenchmarks

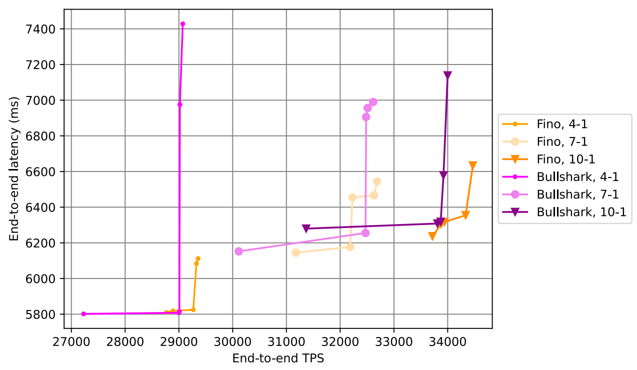

Figure 3: Fino vs. Bullshark, simulated latency with Weibull distribution. The labels indicate number of nodes in committee, and the number following the hyphen is number of faulty nodes.

In Figure 3, we simulate latency for incoming messages into each node using a Weibull distribution. The scale and shape parameters of the distribution are 1.5 and 2.5, and we use this to simulate milliseconds of latency by sleeping nodes during this period. We simulate latency across 25 iterations of the simulation and track minimum, 25th percentile, 50th percentile, 75th percentile, and maximum end-to-end throughput and latency. Our evaluations show Fino and Bullshark fall into the same class of throughput, with minor differences that we attribute to the probabilistic simulation of latency. Fino also has smaller spreads in the latency in comparison to Bullshark: Bullshark's end-to-end latency ranged between roughly 700 - 1600ms, whereas Fino's end-to-end latency ranged between roughly 400 - 1100ms. This helps highlight the strong performance comparison between Fino and Bullshark and the potential benefits of integrating timeouts within the DAG. We see the general trends apply to both Fino and Bullshark across different committee configurations. This microbenchmark also gives us some insights of the performance of Fino with faults against other algorithms. From this, we can see Fino maintains its throughput without incurring high latency penalties in the face of certain models of network behavior and faulty actors. However, we believe a bimodal distribution may better simulate network latency in a uncongested and congested manner, leading to our next simulations.

In Figure 4, we simulate a bimodal distribution of network latency to simulate a fast- and slow-case for network latency

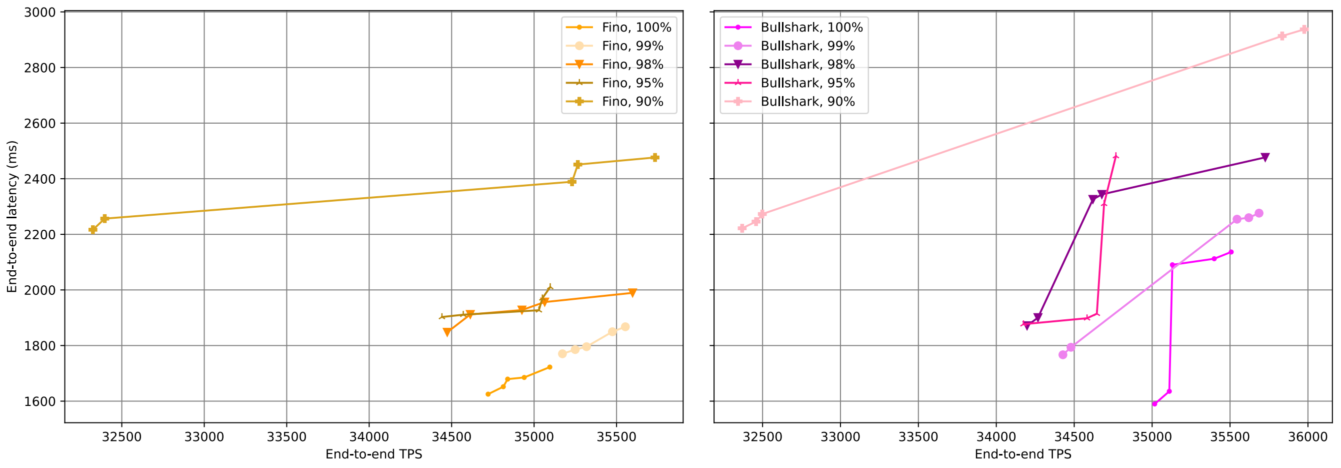

Figure 4: Fino and Bullshark with simulated latency microbenchmark using a bimodal distribution. Each curve represents a different sampling ratio between the two modes of the distribution for network latency.

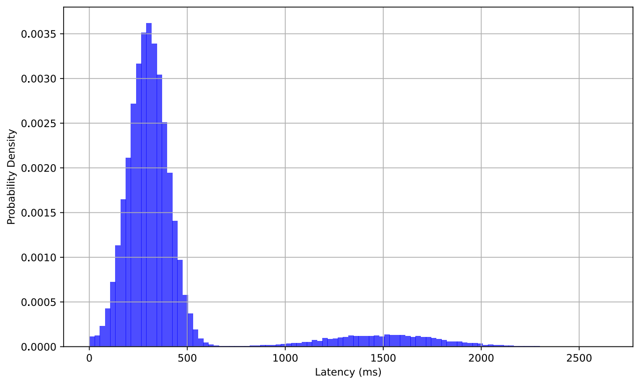

Figure 5: Probability distribution of latencies with 100,000 samples of bimodal distribution sampling 95% from $\mu_1 =$ 300ms, $\sigma_1 = 100$, and 5% from $\mu_2 = 1500$ms, $\sigma_2 = 300$. This is the "slow" distribution we used for our simulations.

and bad actors. We use 2 distributions, dubbed "fast" and "slow" for brevity. The fast distribution has $\mu_1 = 2$ms, $\sigma_1 = 2$, $\mu_2 = 300$ms, $\sigma_2 = 100$. The slow distribution has $\mu_1 = 300$ms, $\sigma_1 = 100$, $\mu_2 = 1500$ms, $\sigma_2 = 300$. We run the simulation across five "quick-ratios" of 100%, 99%, 98%, 95%, and 90% for each distribution, where a "quick-ratio" of 95% means 95% of the network's latency is sampled from the normal distribution $N(\mu_1, \sigma_1)$, whereas the other 5% is sampled from the normal distribution $N(\mu_2, \sigma_2)$. This allows us to explore how the performance of each protocol is affected as the frequency of network congestion changes. Similarly to above, we simulate latency across 25 iterations of the simulation and track minimum, 25th percentile, 50th percentile, 75th percentile, and maximum end-to-end throughput and latency. We primarily focus on the slow distribution for our analysis, as we believe it more closely models real-world network behavior.

Our evaluations show that Fino's throughput performance is slightly worse than that of Bullshark's, but also tends to have less spread. An increase in network reliability modeled by an increasing quick-ratio consistently yielded higher throughput minimums for Bullshark, whereas the throughput maximum tended to fall within the 35,500 tx/s - 36,000 tx/s range (with the exception of 34,7800 tx/s under a 95% quick-ratio). Fino saw its throughput maximums fall within the 35,000 tx/s - 36,000 tx/s range, and its throughput minimum takes a step back from about 35,200 tx/s at a 99% quick-ratio to only 34,700 tx/s at a 100% quick-ratio (near-perfect network stability!); its maximum throughput additionally steps back from 35,600 tx/s to 35,100 tx/s between a 99% and 100% quick-ratio. However, its spread became noticeably tighter than their Bullshark counterparts as the network becomes less stable: Fino yielded a range of 347 tx/s under a 100% quick-ratio (compared to Bullshark's 491 tx/s) and a range of 381 tx/s under a 99% quick-ratio (compared to Bullshark's 1,259 tx/s). On average, Fino is able to outperform Bullshark's throughput as the network latency increases and becomes more unstable. Fino and Bullshark are neck-and-neck for all quick-ratios within the fast distribution, but the performance advantage Fino has on Bullshark's throughput grows as the quick-ratio decreases within the slow distribution, if only marginally: for all quick-ratios within the fast distribution, Fino and Bullshark's throughputs are within 0.5% of each other—under a quick-ratio of 98% on the slow distribution, Fino outperforms Bullshark's throughput by 0.8%; this performance gap further increases to 2.4% under a quick-ratio of 90%.

On the other hand, Fino's latency tends to be more stable than Bullshark's; the protocols' minimum latency values are comparable across all quick-ratios, but Bullshark tends to experience a latency spike between the 25th and 75th percentile—yielding latency ranges from about 509ms to 715ms between quick-ratios—whereas Fino's maximum latency values are comparable to their corresponding minimums within each quick-ratio—yielding latency ranges from about 97ms to 260ms and noticeably outperforming the maximum latency values of Bullshark. This tendency to not stray far from its minimum latency values means that Fino is capable of consistently outperforming Bullshark's average latency by 10% - 15% for most quick-ratios under both slow and fast distributions—the minimum such gap being 6% under the slow distribution with a quick-ratio of 95%.

5.4 End-to-End Macrobenchmarks

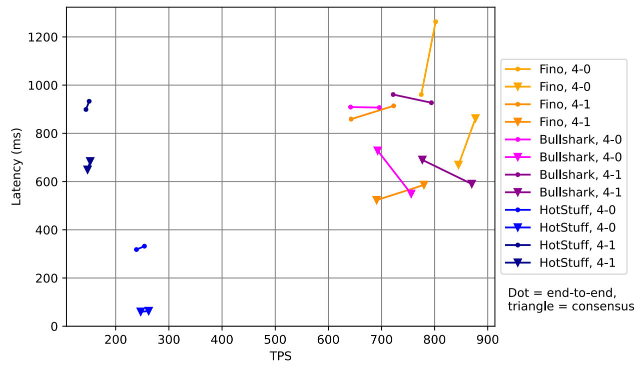

Figure 6: Macrobenchmarks integrating a full consensus and execution environment with Fino and Bullshark.

In Figure 6, we explore the performance of Fino, Bullshark, and HotStuff using the end-to-end testing framework described in Section 4.2, focusing on the end-to-end throughput and latency of the system. As described above, we use the Ethereum Virtual Machine as our execution environment, communicating with it via the Ethereum RPC. In the first benchmark, we want to push the systems to their limits, so we remove all simulated latency and test for maximum throughput and minimum latency. We see that in an end-to-end test, the fast consensus commits HotStuff can achieve do not translate to fast end-to-end latency, despite its expected low latency. Overall, HotStuff tends to achieve less throughput and slightly less latency than Fino and Bullshark. Comparing Fino and Bullshark shows the inconsistent performance of Fino. Fino achieves roughly the same throughput as Bullshark, but tends to have slightly higher latency. We attribute this to a mis-synchronization between the Narwhal messaging layer and the Fino logical layer. It is possible in certain cases, the Fino logical layer sends a decision to the Narwhal message layer just after Narwhal sends a header out, leading to the current header needing to wait a full round to be included. We believe this contributed to the increase in latency for Fino, compared to Bullshark's physical integration with Narwhal's DAG.

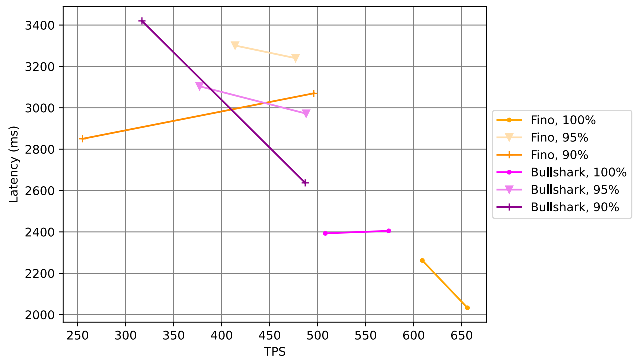

Figure 7: Macrobenchmark of bimodal distribution of simulated latency between Fino and Bullshark. We use a bimodal distribution with $\mu_1 = 300$ms, $\sigma_1 = 100$ and $\mu_2 = 1500$ms, $\sigma_2 = 300$

In Figure 7, we additionally simulate a bimodal distribution of network latency similar to that used in the microbenchmarks. This is an extension of the previous benchmark but with simulated latency to compare how Fino performs in specific network conditions against Bullshark. Like the microbenchmark evaluations, the fast distribution has $\mu_1 = 2$ms, $\sigma_1 = 2$, $\mu_2 = 300$ms, $\sigma_2 = 100$ and the slow distribution has $\mu_1 = 300$ms, $\sigma_1 = 100$, $\mu_2 = 1500$ms, $\sigma_2 = 300$. Again, we primarily focus on the slow distribution for our analysis, as we believe it more closely models real-world network behavior. Comparing Bullshark and Fino shows Fino achieves better performance in the "steady" case of only sampling from the 300ms distribution, but performs worse and worse as we sample from the 1500ms distribution. Noticebaly, Fino has greater variability in latency and throughput than Bullshark, which we attribute to mis-synchronization between Narwhal and Fino timers. The degradation in latency performance of Fino supports this: as we introduce more "out of distribution" latencies, Fino takes longer to commit as timers may get out of sync. The table of data in Section 1 showcases the full data collected, but the general trend we see is Fino achieves higher throughput with worse latency than Bullshark in the end-to-end, simulated latency benchmarks we conducted.

6 Future Work

One interesting area of research is exploring implementations of alternative DAG transport layers besides Narwhal. More specifically, integrating Fino into Narwhal is unfruitful today since a Narwhal header requires $n-f$ certificates in the current round to advance to the next round. This is by design to achieve a quorum of at least 1 honest node between rounds. However, Fino only requires at most $2f+1$ certificates to advance to the next round. Thus, Narwhal imposes a suboptimal latency restriction on Fino. It would be interesting to explore other designs of DAG transport layers that have requirements that fit better with Fino. On the other hand, Narwhal is a well-tested protocol that provides several properties that allow Fino to achieve very high throughput, such as only passing message digests in certificates and effectively separating data availabilty from message transport. Alternative DAG transport layers can still maintain a simple API of broadcasting and delivering transactions without equivocating, but explore other aspects of the design space such as availability or the quasi-linear scalability of Narwhal. Projects such as Aleph [8] and DAG-Rider [11] are alternative designs and further understanding this design space would be helpful for better understanding such BFT protocols.

Another area of future work can be oriented around rethinking the need for a DAG structure in the first place! DAGs provide a strong structured approach to building a blockchain that can be used to build BFT protocols atop, but this is a double-edged sword. A major disadvantage of Narwhal is the notion of equal transaction load between all clients and nodes. In reality, this is often not the case. The distribution of transactions may follow a very long-tailed distribution: a small subset of nodes may get a large proportion of transactions with the remaining nodes in the network getting a smaller proportion. In such cases, nodes with lesser transactions cannot make full use of the Narwhal gossip design. An ideal solution would be to allow nodes in the network to progress at their own speed and not be tied to the network or consensus pace of other nodes, leading to automatic scaling of the consensus layer without depending on the client or transaction load of peers. To do so, the system must be able to guarantee a node's certificates must not depend on other replica's certificates, unlike Narwhal's current design. Exploring this design space would allow for utilization of learnings from DAG-BFT protocols as well as continuing the paradigm of separating consensus from data dissemination.

Additionally, modifying the consensus protocols themselves to be more scalable is another area of interest that could introduce a new axis to optimize along—existing work optimizing distributed consensus protocols like MultiPaxos [18] hint at the possibility of further increasing consensus throughput by the addition of new machines. Through the strategy of decoupling and partitioning nodes (otherwise known as compartmentalization), bottlenecks in consensus protocols can be identified, isolated, and partitioned away to unlock the potential of higher throughput that simply adding more nodes to the system would be unable to achieve—all the while observably identical behavior as the original protocol. These insights can additionally be applied to BFT consensus protocols to achieve higher throughput at scale! The steps required to apply this work to Byzantine protocols would involve reasoning on these transformations under a new failure model; existing work largely only consider how compartmentalization maintains correctness under crash failures—ensuring new nodes being introduced cannot alter protocol behavior and violate correctness through Byzantine failures is crucial to ensuring the transformations are sound.

Integrating and testing additional execution environments can be of further interest. In current literature, DAG-BFT protocols are primarily used in chained settings. Implementing and understanding applicability for traditional replicated state machines such as databases could highlight additional strengths or weaknesses of this class of consensus protocols under different workloads and models. Protocols such as Fino, Bullshark, and Tusk have properties that may perform well or poorly in distributed databases such as RocksDB [6], Postgres [15], or etcd [7]. For example, Fino and Bullshark linearize a committed block's causal history to achieve ordering, but this may not interact well with state machine serialization needed to achieve fast throughput in certain key-value stores. Our end-to-end benchmarks explore such performance for the Ethereum blockchain's state machine and execution layer, but extending this analysis can be fruitful.

7 Conclusion

We explore the implementation of Narwhal-Fino and compare performance by benchmarking Fino against existing state-of-the-art DAG-BFT protocols. We choose Narwhal as the DAG transport layer for Fino, a novel consensus algorithm by Malkhi et al., due to its high throughput and low latency performance, quasi-linear scaling properties with respect to number of workers, and battle-testing in production deployments. Our microbenchmarks highlight the differences between Fino and Bullshark, a similar protocol, and explore the behavior under different network latency to understand how integrating timeouts into DAG messages affects consensus throughput and latency. Our macrobenchmarks add transaction execution using the Ethereum virtual machine, shedding light on performance comparisons in a simulated production system. Our evaluations show Fino is competitive against other DAG-BFT protocols like Bullshark in terms of throughput, and can sometimes offer less latency! However, in certain cases where we find mis-synchronization between the Narwhal messaging layer and Fino logical DAG layer, Fino can incur heavy latency penalties due to messages waiting an additional physical DAG round to be sent.

8 Acknowledgments

We would like to thank Prof. Kubiatowicz for his guidance during the project, especially with regards to robust evalution and benchmarking. We would like to thank Maanav Khaitan for his help in implementing the macrobenchmarking system.

References

[1] ABRAHAM, I., GUETA, G., AND MALKHI, D. Hot-stuff the linear, optimal-resilience, one-message BFT devil. *CoRR abs/1803.05069* (2018).

[2] APTOSLABS. The aptos blockchain: Safe, scalable, and upgradeable web3 infrastructure.

[3] CASTRO, M., AND LISKOV, B. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation* (USA, 1999), OSDI '99, USENIX Association, p. 173–186.

[4] COSMOS. Cosmos: The internet of blockchains.

[5] DANEZIS, G., KOKORIS-KOGIAS, E., SONNINO, A., AND SPIEGELMAN, A. Narwhal and tusk: A dag-based mempool and efficient BFT consensus. *CoRR abs/2105.11827* (2021).

[6] DONG, S., KRYCZKA, A., JIN, Y., AND STUMM, M. Rocksdb: Evolution of development priorities in a key-value store serving large-scale applications. *ACM Transactions on Storage 17*, 4 (Oct. 2021), 1–32.

[7] ETCD-IO. Etcd-io/etcd: Distributed reliable key-value store for the most critical data of a distributed system.

[8] GAGOL, A., LESNIAK, D., STRASZAK, D., AND SWIETEK, M. Aleph: Efficient atomic broadcast in asynchronous networks with byzantine nodes. *CoRR abs/1908.05156* (2019).

[9] GELASHVILI, R., SPIEGELMAN, A., XIANG, Z., DANEZIS, G., LI, Z., MALKHI, D., XIA, Y., AND ZHOU, R. Block-stm: Scaling blockchain execution by turning ordering curse to a performance blessing, 2022.

[10] GIRIDHARAN, N., KOKORIS-KOGIAS, L., SONNINO, A., AND SPIEGELMAN, A. Bullshark: DAG BFT protocols made practical. *CoRR abs/2201.05677* (2022).

[11] KEIDAR, I., KOKORIS-KOGIAS, E., NAOR, O., AND SPIEGELMAN, A. All you need is DAG. *CoRR abs/2102.08325* (2021).

[12] KONSTANTOPOULOS, G., KIRILLOV, A., AND NEU, J. Cosmos without tendermint: Exploring narwhal and bullshark, Jul 2022.

[13] MALKHI, D., AND SZALACHOWSKI, P. Maximal Extractable Value (MEV) Protection on a DAG. *arXiv e-prints* (aug 2022), arXiv:2208.00940.

[14] MYSTENLABS. The sui smart contracts platform.

[15] ROWE, L. A., AND STONEBRAKER, M. The postgres data model. In *Proceedings of the 13th International Conference on Very Large Data Bases* (San Francisco, CA, USA, 1987), VLDB '87, Morgan Kaufmann Publishers Inc., p. 83–96.

[16] TENDERMINT. Application blockchain interface specification.

[17] TOKIO-RS. Tokio-rs/tokio: A runtime for writing reliable asynchronous applications with rust. provides i/o, networking, scheduling, timers, ...

[18] WHITTAKER, M. J., AILIJIANG, A., CHARAPKO, A., DEMIRBAS, M., GIRIDHARAN, N., HELLERSTEIN, J. M., HOWARD, H., STOICA, I., AND SZEKERES, A. Scaling replicated state machines with compartmentalization [technical report]. *CoRR abs/2012.15762* (2020).

[19] WOOD, G., ET AL. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper 151*, 2014 (2014), 1–32.

A Bimodal Distribution Simulated Latency Data

Metric	***Latency Distribution***	***Consensus Protocol***	
		Fino	*Bullshark*
Consensus TPS	*Fast, 100%*	461.5	**611.5**
	Fast, 95%	**789.5**	596.5
	Fast, 90%	**642**	379
	Slow, 100%	**839**	648.5
	Slow, 95%	**539**	526
	Slow, 90%	**448**	445.5
Consensus Latency	*Fast, 100%*	888.5	**532**
	Fast, 95%	712.5	**548.5**
	Fast, 90%	604.5	**539**
	Slow, 100%	**1395.5**	1568.5
	Slow, 95%	2442.5	**2115**
	Slow, 90%	**2183.5**	2336.5
End-to-End TPS	*Fast, 100%*	434	**580**
	Fast, 95%	**724**	561
	Fast, 90%	**594**	363
	Slow, 100%	**632.5**	541
	Slow, 95%	**445.5**	432.5
	Slow, 90%	**375.5**	402
End-to-End Latency	*Fast, 100%*	1257	**773.5**
	Fast, 95%	1234	**834.5**
	Fast, 90%	1062.5	**818**
	Slow, 100%	**2147.5**	2399
	Slow, 95%	3270	**3038**
	Slow, 90%	**2960**	3028.5

Table 1: End-to-end simulated latency with different bimodal distributions, showing the mean of 2 runs

Metric	**Latency Distribution**	***Consensus Protocol***	
		Fino	*Bullshark*
Consensus TPS	*Fast, 100%*	**37030.47**	36963.19
	Fast, 99%	36978.94	**36979.50**
	Fast, 98%	**37015.55**	36990.65
	Fast, 95%	36687.88	**36808.04**
	Fast, 90%	**37013.42**	36814.89
	Slow, 100%	35949.86	**36202.29**
	Slow, 99%	**36373.34**	36178.05
	Slow, 98%	**35892.45**	35669.75
	Slow, 95%	**35908.47**	35614.88
	Slow, 90%	**35344.10**	34448.39
Consensus Latency	*Fast, 100%*	**482.74**	566.61
	Fast, 99%	**508.99**	605.02
	Fast, 98%	**513.78**	596.13
	Fast, 95%	**546.72**	599.77
	Fast, 90%	**580.62**	706.21
	Slow, 100%	**1082.44**	1306.40
	Slow, 99%	**1138.87**	1398.13
	Slow, 98%	**1201.42**	1458.62
	Slow, 95%	**1207.32**	1331.85
	Slow, 90%	**1462.50**	1672.17
End-to-End TPS	*Fast, 100%*	**36763.33**	36750.26
	Fast, 99%	36731.68	**36755.01**
	Fast, 98%	**36759.64**	36755.10
	Fast, 95%	36429.72	**36590.29**
	Fast, 90%	**36761.83**	36599.17
	Slow, 100%	34882.70	**35207.02**
	Slow, 99%	**35355.15**	35113.30
	Slow, 98%	**34871.38**	34629.77
	Slow, 95%	**34876.25**	34592.96
	Slow, 90%	**34243.01**	33435.72
End-to-End Latency	*Fast, 100%*	**631.67**	719.49
	Fast, 99%	**693.10**	787.01
	Fast, 98%	**702.62**	784.85
	Fast, 95%	**755.24**	809.10
	Fast, 90%	**821.65**	945.75
	Slow, 100%	**1673.93**	1897.35
	Slow, 99%	**1813.69**	2071.34
	Slow, 98%	**1932.31**	2186.83
	Slow, 95%	**1938.07**	2058.64
	Slow, 90%	**2357.63**	2565.84

Table 2: Microbenchmarks with simulated latency with different bimodal distributions, showing the mean of 25 runs