Narwhal and Tusk: A DAG-based Mempool and Efficient BFT Consensus

George Danezis, Eleftherios Kokoris Kogias, Alberto Sonnino, Alexander Spiegelman
Facebook Novi

Abstract

We propose separating the task of transaction dissemination from transaction ordering, to enable high-performance Byzantine fault-tolerant consensus in a permissioned setting. To this end, we design and evaluate a mempool protocol, Narwhal, specializing in high-throughput reliable dissemination and storage of causal histories of transactions. Narwhal tolerates an asynchronous network and maintains its performance despite failures. We demonstrate that composing Narwhal with a partially synchronous consensus protocol (HotStuff) yields significantly better throughput even in the presence of faults. However, loss of liveness during view-changes can result in high latency. To achieve overall good performance when faults occur we propose Tusk, a zero-message overhead asynchronous consensus protocol embedded within Narwhal. We demonstrate its high performance under a variety of configurations and faults. Further, Narwhal is designed to easily scale-out using multiple workers at each validator, and we demonstrate that there is no foreseeable limit to the throughput we can achieve for consensus, with a few seconds latency.

As a summary of results, on a Wide Area Network (WAN), Hotstuff over Narwhal achieves 170,000 tx/sec with a 2.5-sec latency instead of 1,800 tx/sec with 1-sec latency of Hotstuff. Additional workers increase throughput linearly to 600,000 tx/sec without any latency increase. Tusk achieves 140,000 tx/sec with 4 seconds latency or 20x better than the state-of-the-art asynchronous protocol. Under faults, both Narwhal based protocols maintain high throughput, but the HotStuff variant suffers from slightly higher latency.

1 Introduction

Increasing blockchain performance has is studied widely, to improve on Bitcoin's [31] throughput of only 4 tx/sec. Bitcoin-NG [19] splits the consensus protocol from block proposal, achieving a 20x throughput increase, with a latency of 10 minutes (same as Bitcoin). Byzcoin [26] was the first to propose using quorum-based consensus to break the barrier of 1,000 tx/sec with a 1-minute latency. For higher throughput and lower latency quorum-based protocols are required, and Narwhal is such a protocol.

Existing approaches to increasing the performance of distributed ledgers focus on creating lower-cost consensus algorithms with the crown jewel being Hotstuff [35], which achieves linear overhead in the partially synchronous setting. To achieve this, Hotstuff leverages a leader who collects, aggregates, and broadcasts the messages of other validators during their rounds. However, theoretical message complexity can be a misleading optimization target. More specifically:

- Any (partially-synchronous) protocol that minimizes overall message number, but relies on a leader to produce proposals and coordinate consensus, fails to capture the high load this imposes on the leader who inevitably becomes a bottleneck.
- Message complexity counts the number of *metadata* messages (e.g., votes, signatures, hashes) which take minimal bandwidth compared to the dissemination of bulk transaction data (blocks). Since blocks are orders of magnitude larger (10MB) than a typical consensus message (100B), the overall message complexity is irrelevant in practice.

Besides minimizing misleading cost metrics, consensus protocols have conflated many functions into a monolithic protocol. In a typical distributed ledger, such as LibraBFT[1] [10], clients send transactions to a validator that shares them using a Mempool protocol. Then a subset of these transactions are periodically re-shared and committed as part of the consensus protocol. Most research so far aims to increase the throughput of the consensus layer.

This paper formulates the following hypothesis: **a better Mempool, that reliably distributes transactions, is the key enabler of a high-performance ledger. It should be separated from the consensus protocol altogether, leaving consensus only the job of ordering small fixed-size references. This leads to an overall system throughput being largely unaffected by consensus.**

This work confirms the hypothesis; monolithic protocols place transaction dissemination in the critical path of consensus, impacting performance more severely than consensus itself. With Narwhal, we show that we can off-load *reliable* transaction dissemination to the Mempool protocol, and only rely on consensus to sequence a very small amount of metadata, increasing performance significantly. Therefore, there is a clear gap between what in theory is optimal for an isolated consensus protocol and what offers good performance in a real distributed ledger.

We adapt Hotstuff to separate block dissemination into a separate Mempool layer and call the resulting system *batched-HotStuff* (Batched-HS). In Batched-HS, validators broadcast blocks of transactions in a Mempool and the leader

[1] LibraBFT recently rebranded to DiemBFT.

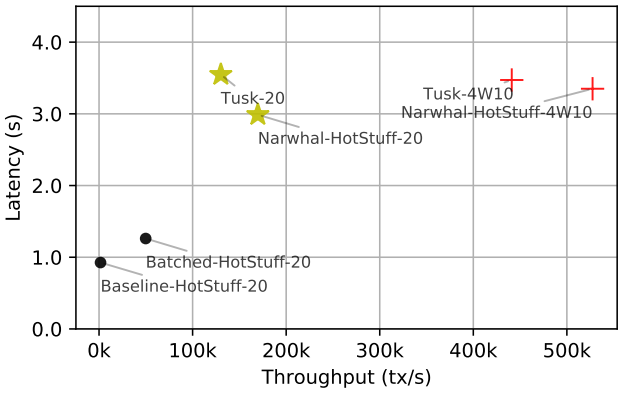

Figure 1. Summary of WAN performance results, for consensus systems with traditional mempool (circle), Narwhal mempool (star), and many workers (cross). Transactions are 512B.

proposes block hashes during consensus, instead of broadcasting transactions in the critical path. This design, however, only performs well under ideal network conditions where the proposal of the leader is available to the majority of the validators promptly.

To make a robust Mempool we design Narwhal, a DAG-based, structured Mempool which implements causal order reliable broadcast of transaction blocks, exploiting the available resources of validators in full. Combining the Narwhal Mempool with HotStuff (Narwhal-HS) provides good throughput even under faults or asynchronous network conditions (but at an inevitable high latency). To reduce latency under faults and asynchrony, we note that Narwhal is close to an asynchronous consensus protocol. We, therefore, enhance Narwhal with the capability of producing common randomness, and design Tusk, a fully-asynchronous, wait-free consensus where each party decides the agreed values by examining its local DAG *without sending any messages*. Tusk removes the need to rely on an external consensus layer in deployments where slightly higher latency is acceptable.

Contributions. We make the following contributions:

- We design Batched-Hotstuff with a simple Mempool that produces significant (x50) improvement under good network conditions and demonstrate HotStuff's lack of performance under even a few failures.
- We build Narwhal, an advanced Mempool protocol that guarantees optimal throughput (based on network speed) even under asynchrony and combines it with our Hotstuff implementation to see increased throughput at a modest expense of latency.
- We leverage the structure of Narwhal and enhance it with randomness to get Tusk, a high-throughput, DDoS resilient, and zero overhead consensus protocol. We demonstrate experimentally its high performance in a WAN, even when failures occur.

Figure 1 summarizes the relative WAN performance of the Narwhal-based systems (star markers), compared with HotStuff (circle marker), when no faults occur, for different numbers of validators and workers (cross marker, number of workers after 'W'). Throughput (x-axis) is increased with a single Narwhal worker and vastly increased when leveraging the parallelizable nature of Narwhal to a throughput of over 500,000 tx/sec, for a latency (y-axis) lower than 3.5 seconds.

2 Overview

This paper presents the design and implementation of Narwhal, a DAG-based *Mempool abstraction*. Narwhal ensures efficient wide availability and integrity of user-submitted transactions under a fully asynchronous network. In Section 3.2, we show how to leverage Narwhal to significantly improve the throughput of existing consensus protocols such as HotStuff [35]. Additionally, when augmented with a simple shared-randomness mechanism, Narwhal can be used to implement Tusk, a zero-message-overhead asynchronous consensus algorithm. In this section we define the problem it addresses, the system and security model in which Narwhal operates, as well as a high-level overview of the system and the main engineering challenges.

2.1 System model, goals and assumptions

We assume a message-passing system with a set of n parties and a computationally bounded adversary that controls the network and can corrupt up to $f < n/3$ parties. We say that parties corrupted by the adversary are *Byzantine* or *faulty* and the rest are *honest* or *correct*. To capture real-world networks we assume asynchronous *eventually reliable* communication links among honest parties. That is, there is no bound on message delays and there is a finite but unknown number of messages that can be lost.

Informally the Narwhal Mempool exposes to all participants, a *key-value* block store abstraction that can be used to read and write blocks of transactions and extract partial orders on these blocks. Nodes maintaining the Mempool are able to use the short key to refer to values stored in the shared store and convince others that these values will be available upon request by anyone. The Narwhal Mempool uses a round-based DAG structure that we describe in detail in the next sections. We first provide a formal definition of the Narwhal Mempool.

A *block b* contains a list of transactions and a list of references to previous blocks. The unique (cryptographic) digest of its contents, d, is used as its identifier to reference the block. Including in a block, a reference to a previous block encodes a 'happened-before' [29] relation between the blocks (which we denote $b \rightarrow b'$). The ordering of transactions and references within the block also explicitly encodes their order, and by convention, we consider all referenced blocks happened before all transactions in the block.

A Mempool abstraction needs to support a number of operations: A *write(d,b)* operation stores a block b associated with its digest (key) d. The returned value *c(d)* represents an

unforgeable *certificate of availability* on the digest d and we say that the write *succeeds* when *c(d)* is formed. A *valid(d, c(d))* operation returns true if the certificate is valid, and false if it is not. A *read(d)* operation returns a block b if a write(d,b) has succeeded. A *read_causal(d)* returns a set of blocks B such that $\forall b' \in B \quad b' \rightarrow \ldots \rightarrow \text{read}(d)$, i.e., for every $b' \in B$, there is a transitive happened before relationship with b.

The Narwhal Mempool abstraction satisfies the following:

- **Integrity:** For any certified digest d every two invocations of *read(d)* by honest parties that return a value, return the same value.
- **Block-Availability:** If a read operation *read(d)* is invoked by an honest party after $write(d, b)$ succeeds for an honest party, the *read(d)* eventually completes and returns b.
- **Containment:** Let B be the set returned by a *read_causal(d)* operation, then for every $b' \in B$, the set B' returned by *read_causal(d')*, $B' \subseteq B$.
- $2/3$-**Causality:** A successful *read_causal(d)* returns a set B that contains at least $2/3$ of the blocks written successfully before $write(d, b)$ was invokes.
- $1/2$-**Chain Quality** At least $1/2$ of the blocks in the returned set B of a successful *read_causal(d)* invocation were written by honest parties.

The Integrity and Block-Availability properties of Narwhal allow us to clearly separate data dissemination from consensus. That is, with Narwhal, the consensus layer only needs to order block digests, which requires low communication cost. Moreover, the Causality and Containment properties guarantee that any consensus protocol that leverages Narwhal will be able to achieve perfect throughput. This is because once we agree on a block digest, we can safely totally order all its causally ordered blocks. Therefore, with Narwhal, different Byzantine consensus protocols differ only in the latency they achieve under different network conditions. We provide supporting evaluation and discuss trade-offs in detail in Section 6. The Chain-Quality [22] property allows Narwhal to be used for Blockchain systems that care not only about committing operation but also providing censorship resistance to the clients.

Last but not least, our main goal in this paper is to provide a mempool abstraction that can scale out and support the increasing demand in consensus services. Therefore, our goal in this paper is how to achieve the above theoretical properties and at the time satisfy the following:

- **Scale out:** Narwhal's peak throughput increases linearly with the number of resources each validator has while the latency does not suffer.

2.2 Intuitions behind the Narwhal design

Established cryptocurrencies and permissioned systems [10, 31] implement a best-effort gossip Mempool. A transaction submitted to one validator is disseminated to all others and included in the consensus before the originating node acts as a leader. This leads to fine-grained double transmissions: most transactions are shared first by the Mempool, and then the miner/leader creates a block that re-shares them. This section describes how this basic Mempool design is augmented to provide the properties defined above, toward Narwhal to (1) reduce the need for double transmission when leaders propose blocks, and (2) enable scaling out when more resources are available.

A first step to removing the double transmission is to broadcast blocks instead of transactions and let the leader propose a hash of a block, relying on the Mempool layer to provide its **integrity-protected** content. However, integrity without availability is difficult to ensure, and timeliness required by partially synchronous protocols is not guaranteed, leading to poor performance at the slightest instability.

To ensure **availability**, as a second step, we consistently broadcast [14] the block, yielding a certificate that the block is persisted and available for download. Now a leader may propose such a certificate, to prove that the block will be available on time. Since block size can be arbitrarily large compared to the certificate this reduces the throughput required from consensus. However, it requires a leader to include one certificate per Mempool block and if the consensus temporarily loses liveness then the number of certificates to be committed may grow infinitely.

In a third step, we introduce the **causality** property to enable a leader to propose a *single availability certificate for multiple Mempool blocks*: Mempool blocks are extended to contain certificates of past Mempool blocks, including from other validators. As a result, a certificate refers, in addition to the block, to its full causal history and enables perfect throughput even at the presence of an unstable network, effectively decoupling the throughput (which is a Mempool property now) from the latency (which remains a consensus. property). A leader proposing such a fixed-size certificate, therefore, proposes an extension to the sequence containing blocks from the full history of the block. This design is extremely economical of the leader's bandwidth, and ensures that delays in reaching consensus impact latency but not average throughput–as mempool blocks continue to be produced and are eventually committed. Nevertheless, it suffers from two issues: (i) A very fast validator may force others to perform large downloads by generating blocks at a high speed; (ii) it may be the case that honest validators do not get enough bandwidth to share their blocks with others – leading to potential censorship.

To alleviate these issues, a fourth step provides **Chain Quality** by imposing restrictions on when blocks may be created by validators and at what rate. Validators annotate Mempool blocks with rounds and require a new block to contain at least a quorum of unique certificates from the preceding round. As a result, a fraction of honest validators' blocks is included in the causal history of all proposals. Additionally, this means that validators are not permitted to

advance a Mempool round before at least some honest validators have finished the previous round, preventing flooding.

The final fifth design step is that of enabling **scale-out** through scale-out within a single trust domain. Instead of having a single machine creating Mempool blocks, multiple worker machines per validator can share Mempool sub-blocks, called *batches*. One primary integrates references to them in Mempool primary blocks. This enables validators to commit a mass of computational, storage, and networking resources to the task of sharing transactions–allowing for quasi-linear scaling.

Narwhal is the culmination of the above five design steps, evolving the basic Mempool design to a robust and performant data dissemination and availability layer. Narwhal can be used to off-load the critical path of traditional consensus protocols such as HotStuff or leverage the **containment property** to perform fully asynchronous consensus. We next study the Narwhal in more detail.

2.3 Engineering challenges

Narwhal builds a causal history of transaction blocks (a DAG) provided by validators over time using fully asynchronous reliable broadcast primitives. The first engineering challenge for Narwhal comes once again from the mismatch of theoretical models versus practical systems and is that of perfect-point-to-point links. The theoretical assumption is that messages are eventually delivered. That is, the source will keep sending the message (even if the TCP connection drops!) until the receiver acknowledges reception. This requires infinite memory buffers, something we want to avoid in our implementation. To resolve this issue we leverage the fault-tolerance of the quorum-based system to design a different theoretical abstraction of *Quorum-based reliable broadcast* over imperfect links that relies on both push and pull of messages and works with bounded memory (Section 4.1).

The second engineering challenge relates to the goal of providing extremely high transaction throughput (>200,000 TPS) at an acceptably low latency. To increase throughput we clearly require the use of more than one machine per validator. Existing blockchain systems focus on sharding in the permissionless setting [4, 27] where the adversary controls every machine with equal probability. We however assume that a validator can use multiple machines in a single trust domain (e.g., a data center). This enables us to split the validator logic into a single primary that handles the Narwhal protocol and multiple *workers* that share transactions with remote validators, which enables a simpler "scale-out" property. Additionally, this allows the reduction of latency by letting the workers asynchronously stream transactions instead of respecting the round-by-round structure.

3 Narwhal Core Design

In Section 3.1 we present the core protocol for a mempool and then in Section 3.2 we show how to use it to get both a higher-throughput traditional consensus as well as a new zero overhead asynchronous consensus protocol.

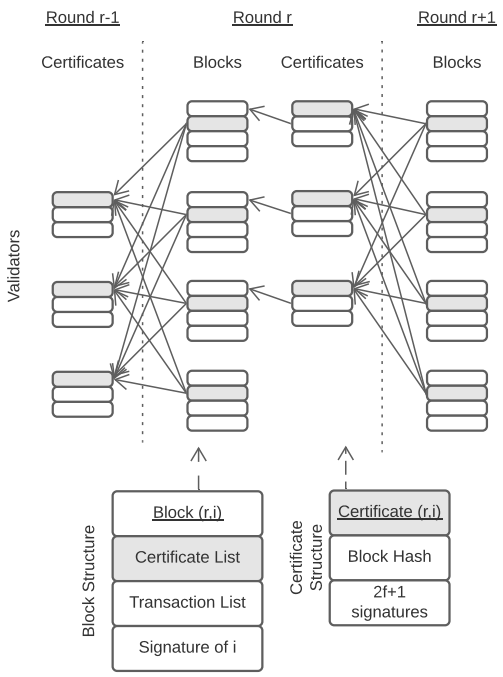

Figure 2. Three rounds of building the mempool DAG. In round $r-1$ there are enough (N-f) certified Blocks, hence validators start building pending Blocks for round r. To do so they include a batch of transactions as well as $N-f$ references to the certified blocks at round $r-1$. Notice that there can be pending blocks from any round, however, the DAG only points to certified blocks from exactly one round before. This enables looking at the DAG as a pacemaker [35] and using it to derive properties for consensus. Once a validator has a ready block it broadcasts it to others to form a certificate and then shares the certificate with all other validators that use them to create blocks at round $r+1$.

3.1 The Narwhal Mempool

The Narwhal Mempool is based on ideas from reliable broadcast [12] and reliable storage [2]. It additionally uses a Byzantine fault-tolerant version of Threshold clocks [21] as a pacemaker to advance rounds. An illustration of Narwhal operation, forming a block DAG, can be seen in Figure 2.

Each validator maintains the current local round r, starting at zero. Validators continuously receive transactions from clients and accumulate them into a transaction list (see Fig. 2). They also receive certificates of availability for blocks at r and accumulate them into a certificate list.

Once certificates for round $r-1$ are accumulated from $2f+1$ distinct validators, a validator moves the local round to r, creates, and broadcasts a block for the new round. Each block includes the identity of its creator, and local round r, the current list of transactions and certificates from $r-1$, and a signature from its creator. Correct validators only create a single block per round.

The validators *reliably broadcast* [12] each block they create to ensure integrity and availability of the block. For practical reasons we do not implement the standard push strategy that requires quadratic communication, but instead use a pull strategy to make sure we do not pay the communication penalty in the common case (we give more details in Section 4.1). In a nutshell, the block creator sends the block to all validators, who check if it is *valid* and then reply with their signatures. A valid block 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.

If a block is valid the other validators store it and acknowledge it by signing its block digest, round number, and creator's identity. We note that condition (2) may lead to blocks with an older logical time being dismissed by some validators. However, blocks with a future round contain $2f + 1$ certificates that ensure a validator advances its round into the future and signs the newer block. Once the creator gets $2f + 1$ distinct acknowledgments for a block, it combines them into a *certificate of block availability* , that includes the block digest, current round, and creator identity. Then, the creator sends the certificate to all other validators so that they can include it in their next block.

The system is initialized through all validators creating and certifying empty blocks for round $r = 0$. These blocks do not contain any transactions and are valid without reference to certificates for past blocks.

Intuitions behind security argument. The existence of the certificate of availability for a block, with $2f+1$ signatures, proves that at least $f + 1$ honest validators have checked and stored the block. Thus, the block is available for retrieval when needed by the execution engine. Further, since honest validators have checked the conditions before signing the certificate, quorum intersection ensures Block-Availability and Integrity (i.e., prevent equivocation) of each block. In addition, since a block contains references to certificates from the previous round, we get by an inductive argument that all blocks in the causal history are certified and thus all history can be later retrieved, satisfying causality. Fuller security proofs for all properties of a mempool are provided in Appendix A.1.

3.2 Using Narwhal for consensus

Figure 3 presents how Narwhal can be combined with an eventually synchronous consensus protocol (top) to enable high-throughput total ordering of transaction, which we elaborate on in Section 3.2.1; Narwhal can also be augmented with a random coin mechanism to yield Tusk (Figure 3, bottom), a high-performance asynchronous consensus protocol we present in Section 3.2.2. Table 1 summarizes the theoretical comparison of vanilla Hotstuff with Narwhal-based systems, which we validate in our evaluation.

Table 1. A comparison between Hotstuff and our protocols. We measure lateny in RTTs (or certificates). Unstable Network is a network that allows for one commit between periods of asynchrony. By optimal we mean the throughput when all messages are delivered at the worse case schedule normalized by the longest message delay.

	HS	Narwhal-HS	Tusk
Average-Case (Latency)	3	4	4
Worse-Case f (Crashes Latency)	$O(n)$	$O(n)$	4
Asynchronous (Latency)	N/A	N/A	7
Unstable Network (Throughput)	1 block	$\frac{2}{3}*$ optimal	$\frac{2}{3}*$ optimal
Asynchronous (Throughput)	0	0	$\frac{2}{3}*$ optimal

3.2.1 Eventually synchronous consensus Established consensus algorithms operating under partial synchrony, such as Hotstuff [35] or LibraBFT [10], can leverage Narwhal to improve their performance. Such systems have a leader who proposes a block of transactions that is certified by other validators. Instead of proposing a block of transactions, a leader can propose one or more certificates of availability provided by Narwhal. Upon commit, the full uncommitted causal history of blocks accessible from the certificates is deterministically ordered and committed. Narwhal guarantees that given a certificate all validators see the same causal history, which is itself a DAG over blocks. As a result, any deterministic rule over this DAG leads to the same total ordering of blocks for all validators, achieving consensus. Additionally, thanks to the availability properties of Narwhal all committed blocks can be later retrieved for execution.

There are many advantages to leaders using Narwhal over sending a block of transactions directly. Even in the absence of failures, the leader broadcasting transactions leads to uneven use of resources: the round leader has to use an enormous amount of bandwidth, while bandwidth at every other validator is underused. In contrast, Narwhal ensures bulk transaction information is efficiently and evenly shared at all times, leading to better network utilization and throughput.

A further advantage of using Narwhal is that the average throughput remains theoretically perfect even when consensus is not live for some period. Eventually-synchronous consensus protocols cannot provide liveness during asynchronous periods or when leaders are Byzantine. Therefore, with a naive mempool implementation, overall consensus throughput goes to zero during such periods. Narwhal, in

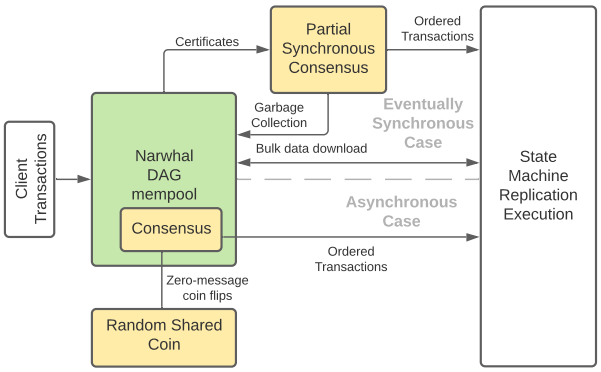

Figure 3. Any consensus protocol can execute over the mempool by occasionally ordering certificates to Narwhal blocks. Narwhal guarantees their availability for the SMR execution. Alternatively, Narwhal structure can be interpreted as an asynchronous consensus protocol with the (zero-message cost) addition of a random-coin.

contrast, continues to share blocks and form certificates of availability even under worst-case asynchronous networks, so blocks are always certified with optimal throughput. Once the consensus protocol manages to commit a digest, validators also commit its causal history, leading to optimal throughput even within periods of asynchrony. Nevertheless, an eventually synchronous protocol still forfeits liveness during periods of asynchrony, leading to increased latency. We show how to overcome this problem with Tusk.

3.2.2 Tusk: Narwhal for asynchronous consensus. In this section, we present the design of Tusk, an asynchronous consensus algorithm for Narwhal that remains live under asynchrony and a large fraction of Byzantine validators.

It is well known that deterministic consensus is impossible under full asynchrony [20]. Hence, our design uses a common secure random coin to achieve consensus. A shared random coin can be constructed from an adaptively secure threshold signature scheme [11] for which key setup can also be performed under full asynchrony [28]. Within Narwhal, each party uses a key-share to compute a partial signature on the round number and includes it into its block for that round. To compute the shared randomness for a round r, validators collect $2f + 1$ shares from the blocks of round r (which are required to advance to round $r+1$), and aggregate them to produce the deterministic threshold signature.

In a nutshell, Narwhal can be interpreted as a round-based DAG of blocks of transactions. Due to the reliable broadcast of each block, all honest validators eventually interpret the same DAG (even in the presence of Byzantine validators). We use the idea from VABA [3] to let the asynchronous adversary commit to a communication schedule and then leverage a perfect shared coin to get a constant probability to agree on a useful work produced therein.

In Tusk, validators interpret every three rounds of the Narwhal DAG as a *consensus instance*. The links in the blocks of the first two rounds are interpreted as all-to-all message exchanges and the third round[2] produces a shared perfect coin that is used to elect a unique block from the first round to be the leader of the instance. The goal is to interpret the DAG in a way that, with a constant probability, safely commit the leader of each instance. Once a leader is committed, its entire causal history in the DAG can be safely totally ordered by any deterministic rule.

The commit rule for Tusk is simple. A validator commits a block leader b of an instance i if its local view of the DAG includes at least $f + 1$ nodes in the second round of instance i with links to b. Figure 4 illustrates two rounds of Tusk (five rounds of Narwhal): in round 3 an insufficient number of blocks support the leader at round 1 (L1); in round 5 sufficient blocks support the leader block at round 3 (L2) and as a result both L1 and L2 are sequenced. A deterministic algorithm on the sub-DAG causally defendant to L1 and then L2 is then applied to sequence all blocks and transactions.

Note that since validators may have different local DAGs at the time they interpret instance i, some may commit b while others do not. However, since any quorum of $2f + 1$ nodes intersect with any quorum of $f + 1$ nodes, we prove in Appendix A.2 that:

Lemma 1. *If an honest validator commits a leader block b in an instance i, then any leader block b' committed by any honest validator v in future instances have a path to b in v's local DAG.*

To guarantee that honest validators commit the same block leaders in the same order, we leverage the above lemma: when the commit rule is satisfied for the leader of some instance, we recursively go back to the last instance in which we committed a leader, and for every instance i in between we check whether there is a path between the leader of the current instance and the leader of i. In case there is such a path we commit the leader of instance i before the leader of the current instance. We use Lemma 1 to prove the following:

Lemma 2. *Any two honest validators commit the same sequence of block leaders.*

Once a validator commits a block leader it deterministically commits all its causal history. By the Containment property of the DAG and the fact the eventually all validators have the same DAG (due to reliable broadcast), it is guaranteed that all honest validators agree on the total order.

As for termination, we use a combinatorial argument to prove in Appendix A.2 that:

Lemma 3. *For every instance i there are at least $f + 1$ blocks in the first round of instance i that satisfy the commit rule.*

Therefore, since the adversary learns who is the leader of an instance only after its first two rounds are fixed and thus only after the $f + 1$ blocks that satisfy the commit rule

[2]To improve latency we combine the third round with the first round of the next instance.

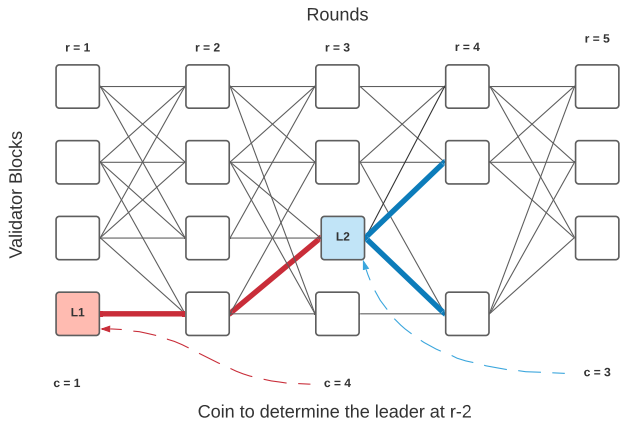

Figure 4. Example of commit rule in Tusk. Every odd round has a coin value that selects a leader of round $r-2$. If the leader has less than $f+1$ support (red) they are ignored, otherwise (blue) the algorithm searches the causal DAG to commit all preceding leaders (including red) and totally orders the rest of the DAG afterward.

are determined, there is a probability of at least $\frac{f+1}{3f+1} > 1/3$ to commit a leader in every instance even if the adversary fully controls the network. In Appendix A.2 we prove the following lemma:

Lemma 4. *In expectation, Tusk commits a block leader every 7 rounds in the DAG under worst-case asynchronous adversary.*

However, if we consider more realistic networks and adversaries we get that Tusk has a much better latency. That is, if the message delays are distributed uniformly at random we prove that

Lemma 5. *In expectation, Tusk commits a block leader every 4 rounds in the DAG in networks with random message delays.*

It is important to note here that neither the generation of a random shared coin in every instance, nor the consensus logic above introduce any additional messages over the Mempool-only Narwhal. Therefore, the asynchronous consensus protocol has zero message overhead, and the same theoretical throughput as the Mempool protocol – but an increased latency.

3.3 Garbage Collection

The final theoretical contribution of Narwhal paired with any consensus algorithm is lifting one of the main roadblocks to the adoption of DAG-based consensus algorithms (e.g., Hashgraph [9]), garbage collection[3]. This challenge stems from the fact that a DAG is a local structure and although it will eventually converge to the same version in all validators there is no guarantee on when this will happen. As a result, validators may have to keep all messages (blocks and certificates) readily accessible to (1) help their peers catch up and (2) be able to process arbitrary old messages.

[3] A bug in our garbage collection led to exhausting 120GB of RAM in minutes compared to 700MB memory footprint of Tusk

This is not a problem in Narwhal, where we impose a strict round-based structure on messages. This restriction allows validators to decide on the validity of a block only from information about the current round (to ensure uniqueness of signed blocks). Any other message, such as certified blocks, carries enough information for validity to be established only with reference to cryptographic verification keys. As a result, validators in Narwhal are not required to store the entire history to verify new blocks. However, note that if two validators garbage collect different rounds then when a new block b is committed, validators might disagree on b's causal history and thus totally order different histories. To this end, Narwhal leverages the properties of a consensus protocol (such as the one we discuss in the previous section) to agree on the garbage collection round. All blocks from earlier rounds can be safely pushed to cold storage and all later messages from previous rounds can be safely ignored.

All in all, validators in Narwhal can operate with a fixed size memory. That is, $O(n)$ in-memory usage on a validator, containing blocks and certificates for the current round, is enough to operate correctly. Since certificates ensure block availability and integrity, storing and servicing requests for blocks from previous rounds can be offloaded to a passive and scalable distributed store or an external provider operating a Content Distribution Network (CDN) such as Cloudflare or S3. Protocols using the DAG content as a mempool for consensus can directly access data from the CDN after sequencing to enable execution of transactions using techniques from deterministic databases [1].

4 Building a Practical System

In this section, we discuss two key practical challenges we had to address in order to enable Narwhal to reach its full theoretical potential.

4.1 Quorum-based reliable broadcast

In real-world reliable channels, like TCP, all state is lost and re-transmission ends if a connection drops. Theoretical reliable broadcast protocols, such as double-echo [14], rely on perfect point-to-point channels that re-transmit the same message forever, or at least until an acknowledgment, requiring unbounded memory to store messages at the application level. Since some validators may be Byzantine, acknowledgments cannot mitigate the denial-of-service risk.

To avoid the need for perfect point-to-point channels we take advantage of the fault tolerance and the replication provided by the quorums we rely on to construct the DAG. As briefly described above, in the Narwhal implementation each validator broadcasts a block for each round r: Subject to conditions specified, if $2f+1$ validators receive a block, they acknowledge it with a signature. $2f+1$ such signatures form a certificate of availability, that is then shared, and potentially included in blocks at round $r+1$. Once a validator advances

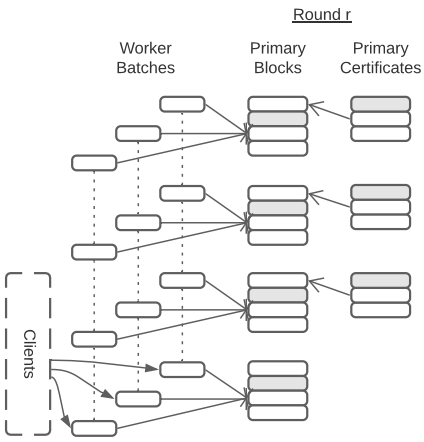

Figure 5. Scale-Out Architecture in Narwhal. Inside a single Trust domain, there is one primary machine that handles the meta-data of building the DAG and multiple worker machines (3 shown in the example) each one streaming Transactions batches to other Trust domains. The hashes of the batches are sent to the primary who adds them in a block as if they are the actual transaction load of the DAG. Clients send transactions to worker machines at all validators.

to round $r + 1$ it *stops re-transmission and drops all pending undelivered messages* for rounds smaller than $r + 1$.

A certificate-of-availability does not guarantee the totality property[4] needed for reliable broadcast: it may be that some honest nodes receive a block but others do not. However, if a block at round $r + 1$ has a certificate-of-availability, the totality property can be ensured for all $2f + 1$ blocks with certificates it contains for round r. Upon, receiving a certificate for a block at round $r + 1$ validators can request all blocks in its causal history from validators that signed the certificates: since at least $f + 1$ honest validators store each block, the probability of receiving a correct response grows exponentially after asking a handful of validators. This *pull* mechanism is DoS resistant and efficient: At any time only $O(1)$ requests for each block are active, and all pending requests can be dropped after receiving a correct response with the sought block. This happens within $O(1)$ requests on average, unless the adversary actively attacks the network links, requiring $O(n)$ requests at most, which match the worst-case theoretical lower bound [17].

The combination of block availability certifications, their inclusion in subsequent blocks, and a 'pull mechanism' to request missing certified blocks leads to a reliable broadcast protocol. Storage for re-transmissions is bounded by the time it takes to advance a round and the time it takes to retrieve a certified block – taking space bounded by $O(n)$ in the size of the quorum (with small constants).

4.2 Scale-Out Validators

In a consensus system all correct validators eventually need to receive all sequenced transactions. Narwhal, like any other mempool, is therefore ultimately limited by the bandwidth, storage, and processing capabilities of a single validator. However, a validator is a unit of authority and trust, and does not have to be limited to employing the resources of a single computer. We, therefore, adapt Narwhal to efficiently utilize resources of many computers per validator, to achieve a scale-out architecture.

Core Scale-Out Design. We adapt Narwhal to follow a simple primary-worker architecture as seen in Figure 5. We split the protocol messages into transaction data and meta-data. Transferring and storing transaction data is an 'embarrassingly parallel' process: A load balancer ensures transactions data are received by all workers at a similar rate; a worker then creates a batch of transactions, and sends it to the worker node of each of the other validators; once an acknowledgment has been received by a quorum of these, the cryptographic hash of the batch is shared with the primary of the validator for inclusion in a block.

The primary runs the Narwhal protocol as specified, but instead of including transactions into a block, it includes cryptographic hashes of its own worker batches. The validation conditions for the reliable broadcast at other validators are also adapted to ensure availability: a primary only signs a block if the batches included have been stored by its own workers. This ensures all data referred to by a certificate of availability can be retrieved.

A pull mechanism has to be implemented by the primary to seek missing batches: upon receiving a block that contains such a batch, the primary instructs its worker to pull the batch directly from the associated worker of the creator of the block. This requires minimal bandwidth at the primary. The pull command only needs to be re-transmitted during the duration of the round of the block that triggered it, ensuring only bounded memory is required.

Streaming. Primary blocks are much smaller when hashes of batches instead of transactions are included. Workers constantly create and share batches in the background. Small batches, in the order of a few hundred to a few thousand transactions (approximately 500KB), ensure transactions do not suffer more than some maximum latency. As a result, most of the batches are available to other validators before primary blocks arrive. This reduces latency since (1) there is less wait time from receiving a primary block to signing it and (2) while waiting to advance the round (since we still need $2f + 1$ primary blocks) workers continue to stream new batches to be included in the next round's block.

Future Bottlenecks. At high transaction rates, a validator may increase capacity through adding more workers, or increasing batch sizes. Yet, eventually, the size of the primary blocks will become the bottleneck, requiring a more efficient accumulator such as a Merkle Tree root batch hashes. This is a largely theoretical bottleneck, and in our evaluation we

[4]The properties of reliable broadcast are Validity, No duplication, Integrity, Consistency, and Totality (see page 112 and 117 of [14]).

never managed to observe the primary being a bottleneck. As an illustration: a sample batch size of 1,000 transactions of 512B each, is 512KB. The batch hash within the primary block is a minuscule 32B and 8B for meta-data (worker ID), i.e. 40B. This is a volume reduction ratio of 1:12, and we would need about 12,000 workers before the primary handles data volumes similar to a worker. So we leave the details of more scalable architectures as distant future work.

5 Implementation

We implement a networked multi-core Narwhal validator in Rust, using Tokio[5] for asynchronous networking, ed25519-dalek[6] for elliptic curve based signatures. Data-structures are persisted using RocksDB[7]. We use TCP to achieve reliable point-to-point channels, necessary to correctly implement the distributed system abstractions. We keep a list of messages to be sent between peers in memory and attempt to send them through persistent TCP channels to other peers. In case TCP channels are drooped we attempt to re-establish them and attempt again to send stored messages. Eventually, the primary or worker logic establishes that a message is no more needed to make progress, and it is removed from memory and not re-sent – this ensures that the number of messages to unavailable peers does not become unbounded and a vector for Denial-of-Service. The implementation is around 4,000 LOC and a further 2,000 LOC of unit tests. We are open sourcing the Rust implementation, Amazon web services orchestration scripts, benchmarking scripts, and measurements data to enable reproducible results[8].

We evaluate both Tusk (Section 3.2.2) and HS-over-Narwhal (Section 3.2.1). Additionally, to have a fair comparison we implement Hotstuff, but unlike the original paper we (i) add persistent storage in the nodes (since we are building a fault-tolerant system), (ii) evaluate it in a WAN, and (iii) implement the pacemaker module that is abstracted away following the LibraBFT specification [10]. We specify two versions of Hot-Stuff (HS). First 'baseline-HS' implements the standard way blockchains (Bitcoin or Libra) disseminate single transactions on the gossip/broadcast network. Second Batched-HS implements the first step of Section 2.2 meaning that validators broadcast blocks instead of transactions out of the critical path and the leader proposes hashes of batches to amortize the cost of the initial broadcast. The goal of this version is to show that a straightforward solution already gives benefits in a stable network but is not robust enough for a real deployment.

[5]https://tokio.rs
[6]https://github.com/dalek-cryptography/ed25519-dalek
[7]https://rocksdb.org
[8]https://github.com/novifinancial/mempool-research

6 Evaluation

We evaluate the throughput and latency of our implementation of Narwhal through experiments on AWS. We particularly aim to demonstrate that (i) Narwhal as a Mempool has advantages over the existing simple Mempool as well as straightforward extensions of it and (ii) that the scale-out is effective, in that it increases throughput linearly as expected. Additionally, we want to show that (iii) Tusk is a highly performing consensus protocol that leverages Narwhal to maintain high throughput when increasing the number of validators (proving our claim that message complexity is not that important), as well as that Narwhal, provides (iv) robustness when some part of the system inevitably fail or suffer attacks. For an accompanying theoretical analysis of these claims see Table 1.

We deploy a testbed on Amazon Web Services, using `m5.8xlarge` instances across 5 different AWS regions: N. Virginia (us-east-1), N. California (us-west-1), Sydney (ap-southeast-2), Stockholm (eu-north-1), and Tokyo (ap-northeast-1). They provide 10Gbps of bandwidth, 32 virtual CPUs (16 physical core) on a 2.5GHz, Intel Xeon Platinum 8175, and 128GB memory and run Linux Ubuntu server 20.04.

In the following sections, each measurement in the graphs is the average of 2 runs, and the error bars represent one standard deviation. Our baseline experiment parameters are: 4 validators each running with a single worker, a block size of 1,000 transactions, a transaction size of 512B, and one benchmark client per worker submitting transactions at a fixed rate for a duration of 5 minutes. We then vary these baseline parameters through our experiments to illustrate their impact on performance. When referring to *latency*, we mean the time elapsed from when the client submits the transaction to when the transaction is committed by the leader that proposed it as part of a block. We measure it by tracking sample transactions throughout the system.

6.1 Narwhal as a Mempool

The results of our first set of experiments can be seen in Figure 6. First, we analyze the baseline and batched HS:

Baseline HS: The performance of the baseline HS protocol (see Figure 6, Baseline-HS lines, in the left bottom corner), with a naive mempool as originally proposed, is quite low. With either 10 or 20 validators throughput never exceeds 1,800 tx/s, although latency at such low throughput is very good at around 1 second. Such surprisingly low numbers are comparable to other works [5], who find HS performance to be 3,500 tx/s on LAN without modifications such as only transmitting hashes [34]. Performance evaluations [37] of LibraBFT [10] that uses Baseline HS, report throughput of around 500 tx/s.

Batched HS: We summarize in Figure 6 the performance of Batched HS without faults (see Batched HS lines): maximum throughput we observe is 80,000 tx/s for a committee of

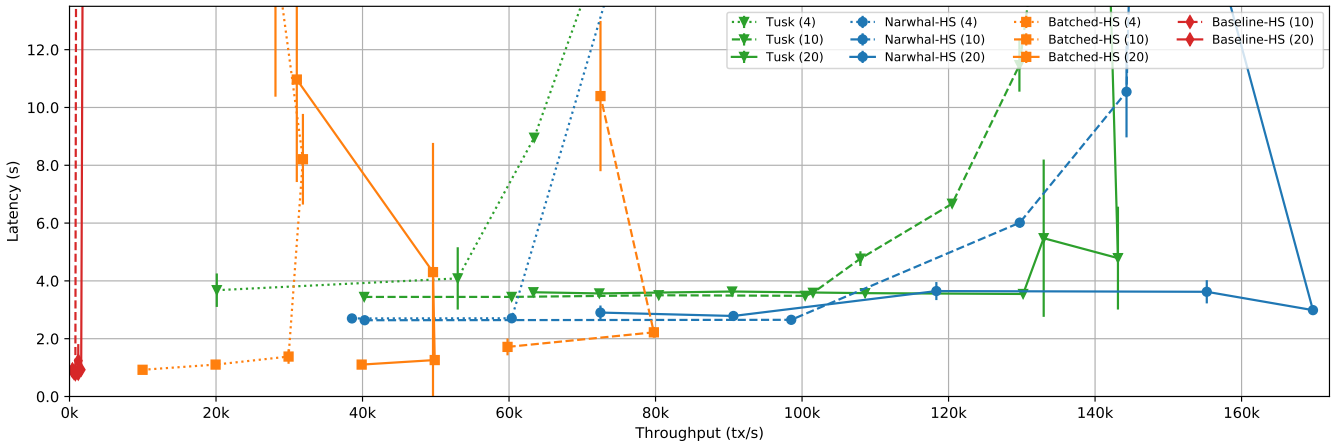

Figure 6. Comparative throughput-latency performance for the novel Narwhal-HotStuff, Tusk, batched-HotStuff and the baseline HotStuff. WAN measurements with 10 or 20 validators, using 1 worker collocated with the primary. No validator faults, 500KB max. block size and 512B transaction size.

10 nodes, and lower (up to 50,000 tx/s) for a larger committee of 20. Latency before saturation is consistently below or at 2 seconds. This almost 20x performance increase compared to baseline HS is the first proof that decoupling transaction dissemination from the critical path of consensus is the key to blockchain scalability.

Narwhal HS: The advantage of Narwhal and causality is illustrated by the Narwhal-HS lines. They show the throughput-latency characteristics of using the Narwhal mempool with HS, for different committee sizes, and 1 worker collocated with each validator. We observe that it achieves 170,000 tx/sec at a latency of around 2.5 seconds similar to the Batched HS latency. This validates the benefit of separating mempool from consensus, with mempool largely affecting the throughput and consensus affecting latency. The counter-intuitive fact that throughput increases with the committee size is due to the worker's implementation not using all resources (network, disk, CPU) optimally. Therefore, more validators and workers lead to increased multiplexing of resource use and higher performance for Narwhal.

Tusk: Finally, the Tusk lines illustrate the latency-throughput relation of Tusk for various committee sizes, and 1 worker collocated with the primary per validator. We observe a stable latency at around 3.5 secs for all committee sizes as well as a peak throughput at 140,000 tx/sec for 20 validators. This is by far the most performant fully asynchronous consensus protocol (see Section 7) and only slightly worse than Narwhal HS. The difference in reported latency compared to theory is that (i) we not only put transactions in the blocks of the first round of a consensus instance but in all rounds, which increases the expected latency by 0.5 rounds and (ii) synchronization is conservative, to not flood validators, hence it takes slightly longer for a validator to collect the full causal graph and start the total ordering.

6.2 Scale-Out using many workers.

To fully evaluate the scalability of Narwhal, we implement each worker and primary on a dedicated instance. For example, a validator running with 4 workers is implemented on 5 machines (4 workers + the primary). The workers are in the same data center as their primary, and validators are distributed over the 5 data centers over a WAN.

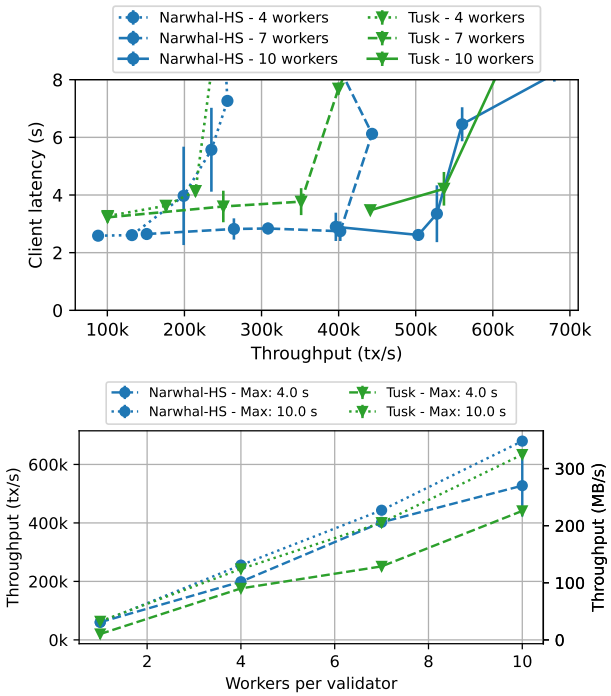

Figure 7. Tusk and HS with Narwhal latency-throughput graph for 4 validators and different number of workers. The transaction and batch sizes are respectively set to 512B and 1,000 transactions.

The top figure at Figure 7 illustrates the latency-throughput graph of Narwhal HS and Tusk for a various number of workers per authority whereas the bottom figure shows the maximum achievable throughput under various service level

objectives. As expected, the deployments with a large number of workers saturate later while they all maintain the same latency, proving our claim that the primary is far from being saturated even with 10 workers concurrently serving hashes of batches. Additionally, on the SLO graph, we can see the linear scaling as the throughput is close to:

$$(\#\text{workers}) * (\text{throughput for one worker})$$

6.3 Performance under Faults.

Figure 8 depicts the performance of all systems when a committee of 10 validators suffers 1 or 3 crash-faults (the maximum that can be tolerated). Both baseline and batched HotStuff suffer a massive degradation in throughput as well as a dramatic increase in latency. For three faults, baseline HotStuff throughput drops 5x (from a very low throughput of 800 tx/s to start with) and latency increases 40x compared to no faults; batched Hotstuff throughput drops 30x (from 80k tx/sec to 2.5k tx/sec) and latency increases 15x.

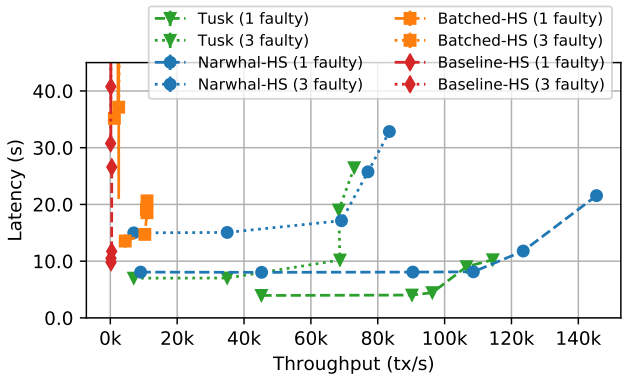

Figure 8. Comparative throughput-latency under faults. WAN measurements with 10 validators, using 1 worker collocated with the primary. One and three faults, 500KB max. block size and 512B transaction size.

In contrast, Tusk and Narwhal-HotStuff maintain a good level of throughput: the underlying Mempool design continues collecting and disseminating transactions despite the faults, and is not overly affected by the faulty validators. The reduction in throughput is in great part due to losing the capacity of faulty validators. As predicted by theory, Tusk's latency is the least affected by faulty nodes, committing in less than 5 sec under 1 fault, and less than 8 sec under 3 faults. The increase in latency is due to the elected block being absent more often. Narwhal-HotStuff exhibits a higher latency, but surprisingly lower than the baseline or batched variants, at less than 10 sec for 1 fault, and around 15 sec for 3 faults (compared to 35-40 sec for baseline or batched). We conjecture this is due to the very low throughput requirements placed on the protocol when combined with Narwhal, as well as the fact that the first successful commit commits a large fraction of recent transactions thanks to the 2/3-Causality.

7 Related work & Limitations

7.1 Performance

Widely-used blockchain systems such as Tendermint, provide 5k tx/sec [13]. However, performance under attack is much contested. Early work suggests that specially crafted attacks can degrade the performance of PBFT consensus systems massively [6], to such a lower performance point that liveness guarantees are meaningless. Recent work targeting PBFT in Hyperledger Fabric [7] corroborates these results [32] and shows latency grows from a few seconds to a few hundred seconds, just through blocks being delayed. Han et al. [24] report similar dramatic performance degradation in cases of simple node crash failure for a number of quorum based systems namely Hyperledger Fabric, Ripple and Corda. In contrast, we demonstrate that Narwhal combined with a traditional consensus mechanism, such as HotStuff maintains its throughput under attack, with increased latency.

Mir-BFT [34], is the most performant variant of PBFT available. For transaction sizes of about 500B (similar to our benchmarks), the peak performance achieved on a WAN for 20 validators is around 80,000 tx/sec under 2 seconds – a performance comparable to our baseline HotStuff with a batched mempool. Impressively, this throughput decreases only slowly for large committees up to 100 nodes (at 60,000 tx/sec). Faults lead to throughput dropping to zero for up to 50 seconds, and then operation resuming after a reconfiguration to exclude faulty nodes. Tusk's single worker configuration for larger committees offers slightly higher performance (a bit less than 2x), but at double the latency. However, Tusk with multiple workers allows 9x better throughput, and is much less sensitive to faults.

We note that careful engineering of the mempool, and efficient transaction dissemination, seems crucial to achieving high-throughput consensus protocols. Recent work [5], benchmarks crash-fault and Byzantine protocols on a LAN, yet observes orders of magnitude lower throughput than this work or Mir-BFT on WAN: 3,500 tx/sec for HotStuff and 500 tx/sec for PBFT. These results are comparable with the poor baseline we achieved when operating HotStuff with a naive best-effort broadcast mempool (see fig. 6).

In the blockchain world, scale-out has come to mean sharding, but this only focuses in the case that every machine distrusts every other machine. However, in classic data-center setting there is a simpler scale-out solution before sharding, that of specializing machines, which we also use. In this paper we do not deal with sharding since our consensus algorithm can obviously interface with any sharded blockchain [4, 8, 27, 36].

7.2 Asynchronous Consensus & Tusk

In the last 5 years the search of practical asynchronous consensus has captured the community [18, 23, 30, 33], because

of the high robustness guarantees it promises. The most performant one was Dumbo2 [23] , which achieves a throughput of 5,000 tx/sec for an SLO of 5 seconds in a setting of 8 nodes in WAN with transaction size of 250B. However, despite the significant improvement, the reported numbers do not realize the hope for a system that can support hundreds thousands of tx/sec We believe that by showing a speedup of 20X over Dumbo2, Tusk finally proves that asynchronous Byzantine consensus can be highly efficient and thus a strong candidate for future scalable deployed Blockchains.

The most closely related work to Tusk is DAG-Rider [25], which is a theoretical concurrent work on the Byzantine Atomic Broadcast problem. They use a DAG structure that resembles ours and have the same asymptotic worse-case analysis. Moreover, their theoretical security argument supports our safety claims. As for liveness, Tusk has a lower number of rounds in the common case due to an optimistic commit rule, but DAG-Rider has better worse-case round complexity (if counted in block of the DAG). In addition, DAG-Rider uses the notion of weak-links to achieve the eventual fairness property required by atomic broadcast, i.e., that any block broadcast by an honest party will be eventually committed. In contrast, we do not guarantee eventual fairness since in practice it requires supporting infinite buffers (caused by asynchrony and faults), which can lead to serious memory issues. All in all, we believe that DAG-Rider further validates the design of Narwhal, since it would take less than 200 LOC to implement DAG-Rider over Narwhal.

7.3 DAG-based Communication & Narwhal

The directed acyclic graph (DAG) data-structure as a substrate for capturing the communication of secure distributed systems in the context of Blockchains has been proposed multiple times. The layered structure of our DAG has also been proposed in the crash fault setting by Ford [21], it is however, embedded in the consensus protocol and does not leverage it for batching and pipelining. Hashgraph [9] embeds an asynchronous consensus mechanism onto a DAG of degree two, without a layered Threshold Clock structure. As a result the logic for when older blocks are no more needed is complex and unclear, and garbage collecting them difficult – leading to potentially unbounded state to decide future blocks. In addition, they use local coins for randomness, which can potentially lead to exponential latency. Blockmania [16] embeds a single-decision variant of PBFT [15] into a non layered DAG leading to a partially synchronous system, with challenges when it comes to garbage collection – as any past blocks may be required for long range decisions in the future. Neither of these protocols use a clear decomposition between lower level sharded availability, and a higher level consensus protocol as Narwhal offers, and thus do not scale out or offer clear ways to garbage collect old blocks.

7.4 Limitations

A limitation of any reactive asynchronous protocol, including Narwhal and Tusk, is that slow authorities are indistinguishable from faulty ones, and as a result the protocol proceeds without them. This creates issues around fairness and incentives, since perfectly correct, but geographically distant authorities may never be able to commit transactions submitted to them. This is a generic limitation of such protocols, and we leave the definition and implementation of fairness mechanisms to future work. Nevertheless, we note that we get 1/2-Chain Quality or at least 50% of all blocks are made by honest parties, which to the best of our knowledge the highest number any existing proposal achieves.

Further, Narwhal relies on clients to re-submit a transaction if it is not sequenced in time, due to the leader being faulty. An expensive alternative is to require clients to submit a transaction to $f + 1$ authorities, but this would divide the bandwidth of authorities by $O(n)$. This is too high a price to pay, since a client submitting a transaction to a fixed k number of authorities has a probability of including a correct one that grows very quickly in k, at the cost of only $O(1)$ overhead. Notably, Mir-BFT uses an interesting transaction de-duplication technique based on hashing which we believe is directly applicable to Narwhal in case such a feature is needed. Ultimately, we relegate this choice to system designers using Narwhal.

8 Conclusion

We experimentally demonstrated the power of Narwhal and Tusk. Narwhal is an advanced mempool enabling Hotstuff to achieve throughput of 170, 000 tx/sec with under 2.5 seconds latency, in a deployment of 20 geographically distributed single-machine validators. Additionally, Narwhal enables any quorum-based blockchain protocol to maintain perfect throughput under periods of asynchrony or faults, as long as the consensus layer is eventually live. Tusk leverages the structure of Narwhal to achieve a throughput of 140, 000 TPS with under 4 seconds latency. The scale-out design allows this throughput to increase to hundreds of thousands TPS without impact on latency.

In one sentence, Narwhal and Tusk irrefutably prove that the main cost of large-scale blockchain protocols *is not* consensus but the reliable transaction dissemination. Yet, dissemination alone, without global sequencing, is an embarrassingly parallelizable function, as we show with the scale-out design of Narwhal.

Our work supports a rethinking in how distributed ledgers and SMR systems are architected, towards pairing a mempool, like Narwhal, to ensure high-throughput even under faults and asynchrony, with a consensus mechanism to achieve low-latency for fixed-size messages. Tusk demonstrates that there exists a zero-message overhead consensus for Narwhal, secure under full asynchrony. As a result,

quorum-based blockchains can scale to potentially millions of transactions per second through scale-out for payments or to build generic reliable systems through state machine replication and smart contracts.

Acknowledgments

This work is funded by Novi, a Facebook subsidiary. We also thank the Novi Research and Engineering teams for valuable feedback, and in particular Mathieu Baudet, Dahlia Malkhi, Andrey Chursin and Zekun Li for early discussions that shaped this work.

References

[1] Daniel J Abadi and Jose M Faleiro. An overview of deterministic database systems. *Communications of the ACM*, 61(9):78–88, 2018.

[2] Michael Abd-El-Malek, Gregory R. Ganger, Garth R. Goodson, Michael K. Reiter, and Jay J. Wylie. Fault-scalable byzantine fault-tolerant services. In Andrew Herbert and Kenneth P. Birman, editors, *Proceedings of the 20th ACM Symposium on Operating Systems Principles 2005, SOSP 2005, Brighton, UK, October 23-26, 2005*, pages 59–74. ACM, 2005.

[3] Ittai Abraham, Dahlia Malkhi, and Alexander Spiegelman. Asymptotically optimal validated asynchronous byzantine agreement. In Peter Robinson and Faith Ellen, editors, *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 - August 2, 2019*, pages 337–346. ACM, 2019.

[4] Mustafa Al-Bassam, Alberto Sonnino, Shehar Bano, Dave Hrycyszyn, and George Danezis. Chainspace: A sharded smart contracts platform. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society, 2018.

[5] Salem Alqahtani and Murat Demirbas. Bottlenecks in blockchain consensus protocols. *CoRR*, abs/2103.04234, 2021.

[6] Yair Amir, Brian Coan, Jonathan Kirsch, and John Lane. Prime: Byzantine replication under attack. *IEEE transactions on dependable and secure computing*, 8(4):564–577, 2010.

[7] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolic, Sharon Weed Cocco, and Jason Yellick. Hyperledger fabric: a distributed operating system for permissioned blockchains. In Rui Oliveira, Pascal Felber, and Y. Charlie Hu, editors, *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*, pages 30:1–30:15. ACM, 2018.

[8] Georgia Avarikioti, Eleftherios Kokoris-Kogias, and Roger Wattenhofer. Divide and scale: Formalization of distributed ledger sharding protocols, 2021.

[9] Leemon Baird. The swirlds hashgraph consensus algorithm: Fair, fast, byzantine fault tolerance. *Swirlds Tech Reports SWIRLDS-TR-2016-01, Tech. Rep*, 2016.

[10] Mathieu Baudet, Avery Ching, Andrey Chursin, George Danezis, François Garillot, Zekun Li, Dahlia Malkhi, Oded Naor, Dmitri Perelman, and Alberto Sonnino. State machine replication in the libra blockchain. *The Libra Assn., Tech. Rep*, 2019.

[11] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. *J. Cryptol.*, 17(4):297–319, 2004.

[12] Gabriel Bracha and Sam Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM (JACM)*, 32(4):824–840, 1985.

[13] Ethan Buchman. Tendermint: Byzantine fault tolerance in the age of blockchains, 2016.

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

[15] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, 2002.

[16] George Danezis and David Hrycyszyn. Blockmania: from block dags to consensus. *arXiv preprint arXiv:1809.01620*, 2018.

[17] Danny Dolev and Rudiger Reischuk. Bounds on information exchange for byzantine agreement. *JACM*, 1985.

[18] Sisi Duan, Michael K. Reiter, and Haibin Zhang. BEAT: asynchronous BFT made practical. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 2028–2041. ACM, 2018.

[19] Ittay Eyal, Adem Efe Gencer, Emin Gün Sirer, and Robbert van Renesse. Bitcoin-ng: A scalable blockchain protocol. In Katerina J. Argyraki and Rebecca Isaacs, editors, *13th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2016, Santa Clara, CA, USA, March 16-18, 2016*, pages 45–59. USENIX Association, 2016.

[20] Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

[21] Bryan Ford. Threshold logical clocks for asynchronous distributed coordination and consensus. *arXiv preprint arXiv:1907.07010*, 2019.

[22] Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part II*, volume 9057 of *Lecture Notes in Computer Science*, pages 281–310. Springer, 2015.

[23] Bingyong Guo, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. Dumbo: Faster asynchronous BFT protocols. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, pages 803–818. ACM, 2020.

[24] Runchao Han, Gary Shapiro, Vincent Gramoli, and Xiwei Xu. On the performance of distributed ledgers for internet of things. *Internet of Things*, 10:100087, 2020.

[25] Idit Keidar, Eleftherios Kokoris-Kogias, Oded Naor, and Alexander Spiegelman. All you need is DAG. In *Proceedings of the 40th Symposium on Principles of Distributed Computing*, PODC '21, New York, NY, USA, 2021. Association for Computing Machinery.

[26] Eleftherios Kokoris Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. Enhancing bitcoin security and performance with strong consistency via collective signing. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 279–296, Austin, TX, August 2016. USENIX Association.

[27] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. Omniledger: A secure, scale-out, decentralized ledger via sharding. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*, pages 583–598. IEEE Computer Society, 2018.

[28] Eleftherios Kokoris-Kogias, Dahlia Malkhi, and Alexander Spiegelman. Asynchronous distributed key generation for computationally-secure randomness, consensus, and threshold signatures. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, pages 1751–1767. ACM, 2020.

[29] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[30] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of bft protocols. In *Proceedings of the 2016 ACM SIGSAC*

Conference on Computer and Communications Security, pages 31–42, 2016.

[31] Satoshi Nakamoto. Bitcoin whitepaper, 2008.

[32] Thanh Son Lam Nguyen, Guillaume Jourjon, Maria Potop-Butucaru, and Kim Loan Thai. Impact of network delays on hyperledger fabric. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 222–227. IEEE, 2019.

[33] Alexander Spiegelman, Arik Rinberg, and Dahlia Malkhi. Ace: Abstract consensus encapsulation for liveness boosting of state machine replication. In *OPODIS*, 2020.

[34] Chrysoula Stathakopoulou, Tudor David, and Marko Vukolic. Mir-bft: High-throughput BFT for blockchains. *CoRR*, abs/1906.05552, 2019.

[35] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan-Gueta, and Ittai Abraham. Hotstuff: BFT consensus with linearity and responsiveness. In Peter Robinson and Faith Ellen, editors, *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 - August 2, 2019*, pages 347–356. ACM, 2019.

[36] Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova. Rapidchain: Scaling blockchain via full sharding. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 931–948. ACM, 2018.

[37] Jiashuo Zhang, Jianbo Gao, Zhenhao Wu, Wentian Yan, Qize Wu, Qingshan Li, and Zhong Chen. Performance analysis of the libra blockchain: An experimental study. *CoRR*, abs/1912.05241, 2019.

A Security Analysis

A.1 DAG

Lemma A.1. *The DAG protocol satisfies Integrity.*

Proof. The lemma follows from the assumption on no hash collusion. That is, it is impossible to find two blocks that are associated with the same digest.

□

Lemma A.2. *The DAG protocol satisfies Block-Availability.*

Proof. Validators locally store every block they sign. A $write(d, b)$ operation completer b is certified. Since $2f + 1$ signatures are required for a block to be certified, we get that at least $f + 1$ honest validators store the block b associated with the digest d. A $read(d)$ operation query all validators and whit for $n - f$ to reply. Therefore, any $write(b, d)$ operation invoked after $write(d, b)$ completes will get a reply from at least 1 honest party that stores b and the lemma follows.

□

Lemma A.3. *The DAG protocol satisfies* $1/2$*-Censorship-Resistance.*

Proof. Since block refers to at least $2f + 1$ from the previous round, B contains at least $2f + 1$ blocks in each round. The lemma follows since at most f of them were written by byzantine validators.

□

Lemma A.4. *The DAG protocol satisfies* $1/3$*-Causality.*

Proof. There are at most $3f + 1$ written blocks associated with each round. Let r be the round in which b was certified, the lemma follows immediately from the fact that B contains at most $2f + 1$ blocks from rounds smaller than r.

□

Lemma A.5. *The DAG protocol satisfies Containment.*

Proof. Every block contains its author and round number and honest validators do not sign two different blocks in the same round from the same author. Therefore, since $2f + 1$ signatures are required to certify a block, two blocks in the same round from the same author can never be certified. Thus, for every certified block that honest validators locally store, they always agree on the set of digest in the block (references to blocks from the previous round). The lemma follows by recursively applying the above argument starting from the block b'.

□

A.2 Asynchronous consensus

Safety.

Lemma 1. *If an honest validator commits a leader block b in an instance i, then any leader block b' committed by any honest validator v in future instances have a path to b in v's local DAG.*

Proof. An honest validator commits a block b in a instance i only if there are at least $f + 1$ nodes in the second round of the instance with links to b. Since every block has at least $2f + 1$ links to blocks in the previous round, we get by quorum intersection that every block in the first round of instance $i + 1$ has a path to b. Therefore, with a simple inductive argument we can show that that every block in every round in instances higher than i have paths to b. The lemma follows.

□

Lemma A.6. *Let b and b' be the block leaders of instances i and i', respectively. If an honest validator v commits b before b', then no honest validator commits b' without first committing b.*

Proof. Since v commits b before b', then there is no path between b to b' in the DAG. Assume by a way of contradiction that some honest validator v' commits b' before b. Thus, there is no path between b to b' in the DAG. However, by Lemma 1, one of the paths must exists. A contradiction.

□

Lemma A.6 immediately implies the following:

Lemma 2. *Any two honest validators commit the same sequence of block leaders.*

Liveness.

Lemma 3. *For every instance i there are at least $f + 1$ blocks in the first round of instance i that satisfy the commit rule.*

Proof. Consider any set S of $2f+1$ blocks in the second round of instance i. The total number of links they have to the first round is $(2f+1)(2f+1) = 4f^2+4f+1$. The number of possible blocks in the first round of the instance is $3f+1$. Therefore, even if every block in the first round has f links from blocks in S, there are still $4f^2+4f+1-f(3f+1) = f^2+3f+1$ links. The maximum number of links from blocks in S to each block in the first round is $2f+1$. Thus, there are at least $\frac{f^2+3f+1}{2f+1-f} \geq f+1$ blocks in the first round such that each one of them has at least $f+1$ links from block in B.

□

Lemma 4. *In expectation, Tusk commits a block leader every 7 rounds in the DAG under worst-case asynchronous adversary.*

Proof. Consider any instance i. By Lemma 3, there are at least $f+1$ block in instance i that satisfy the commit rule. Since the adversary do not know the outcome of the coin before these leaders are determined and since the coin is uniformly distributed, we get that the probability to elect a leader that satisfies the commit rule in instance i is at least $1/3$. Therefore, in expectation, a block leader is committed every 3 instances. Every instance consists of 3 rounds, but since we combine the last round of an instance with the first of the next one, we get that, in expectation, Tusk commits a block leader every 7 rounds in the DAG.

□

Lemma 5. *In expectation, Tusk commits a block leader every 4 rounds in the DAG in networks with random message delays.*

Proof. Consider an instance i, let b_i be the leader of instance i and let S be a set of $2f+1$ blocks in the second round of instance i. Messages delays are distributed uniformly at random, and each block in S includes references to blocks in the first round of instance i independently of other blocks in S. Therefore, each block in S includes a reference to b_i with probability of at least $\frac{2f+1}{3f+1} \geq 2/3$. We next show that the probability that at least $f+1$ nodes in S include b_i is 0.74. To this end, we compute the probability for $f=1$ as for larger f the probability is higher. The probability that at least 2 nodes out of the 3 nodes in S include b_i is $\frac{8}{27}+\frac{12}{27} = 0.74$. Thus, for every instance, the probability to elect a leader that satisfies the commit rule is at least 0.74. Thus, Tusk commits, in expectation, a block leader every 4 rounds (or less for $f>1$) in the DAG.

□