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 reliable transaction dissemination from transaction ordering, to enable high-performance Byzantine fault-tolerant quorum-based consensus. 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 high performance despite failures. 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.

Composing Narwhal with a partially synchronous consensus protocol (Narwhal-HotStuff) yields significantly better throughput even in the presence of faults or intermittent loss of liveness due to asynchrony. However, loss of liveness can result in higher latency. To achieve overall good performance when faults occur we design Tusk, a zero-message overhead asynchronous consensus protocol, to work with Narwhal. We demonstrate its high performance under a variety of configurations and faults.

As a summary of results, on a WAN, Narwhal-Hotstuff achieves over 130,000 tx/sec at less than 2-sec latency compared with 1,800 tx/sec at 1-sec latency for Hotstuff. Additional workers increase throughput linearly to 600,000 tx/sec without any latency increase. Tusk achieves 160,000 tx/sec with about 3 seconds latency. Under faults, both protocols maintain high throughput, but Narwhal-HotStuff suffers from increased latency.

1 Introduction

Byzantine consensus protocols [15, 19, 21] and the state machine replication paradigm [13] for building reliable distributed systems have been studied for over 40 years. However, with the rise in popularity of blockchains there has been a renewed interest in engineering high-performance consensus protocols. Specifically, to improve on Bitcoin's [33] throughput of only 4 tx/sec early works [29] suggested committee based consensus protocols. For higher throughput and lower latency committee-based protocols are required, and are now becoming the norm in proof-of-stake designs.

Existing approaches to increasing the performance of distributed ledgers focus on creating lower-cost consensus algorithms with the crown jewel being Hotstuff [38], which achieves linear message complexity in the partially synchronous setting. To achieve this, Hotstuff leverages a leader who collects, aggregates, and broadcasts the messages of other validators. However, theoretical message complexity should not be the only 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 asymptotic message complexity is irrelevant in practice, for fixed mid-size committees (up to $\sim$ 50 nodes).

Additionally, consensus protocols have conflated many functions into a monolithic protocol. In a typical distributed ledger, such as Bitcoin or LibraBFT[1] [12], 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 throughput.**

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 asymptotically optimal for an isolated consensus protocol and what offers good performance in a real distributed ledger.

Prior work into closing this gap both in permissionless [9, 35] and permissioned protocols [19] focuses only on separating the transmission of large messages and metadata and not on guaranteeing reliability. As a result, they work exceptionally well under no faults but suffer severely at the

[1] LibraBFT recently renamed to DiemBFT.

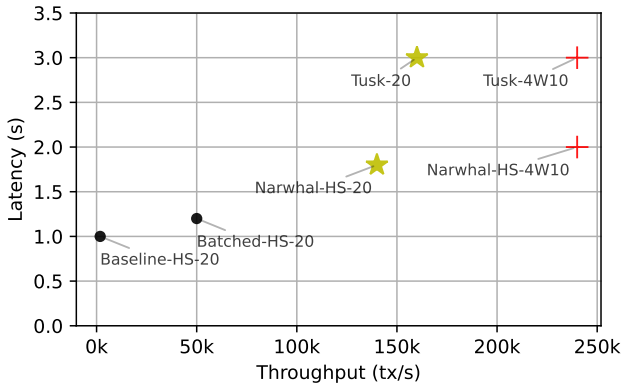

Figure 1. Summary of WAN performance results, for consensus systems with traditional mempool (circle), Narwhal mempool (star), and many workers (cross). Transactions are 512B.

slightest network failure. In order to evaluate this existing approach, we adapt Hotstuff to separate block dissemination into a separate Mempool layer and call the resulting system Batched-HS (Section 6). In Batched-HS, validators broadcast blocks of transactions in a Mempool and the leader proposes block hashes during consensus, instead of broadcasting transactions in the critical path, gaining up to 50x in performance over the existing implementation. 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 unstable network conditions (but at an inevitable higher latency). To reduce latency under faults and asynchrony, we can extend Narwhal with a random coin to provide asynchronous consensus, which we call Tusk. Tusk is a fully-asynchronous, wait-free consensus where each party decides the agreed values by examining its local DAG *without sending any additional messages.*

Contributions. We make the following contributions:

- 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 practical extension of DAG-Rider [28]. Tusk is a high-throughput, DDoS resilient, and zero overhead consensus protocol. We demonstrate experimentally its high performance in a WAN, even when faults 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.

After sharing our initial results with the community two major (top 20 and top 50 by market capitalization) proof of stake blockchains are adopting our design. The full paper with proofs can be found on IPFS with hash: QmegyuTcvDJG94PaLGGErEmH6BmLYhegQ8F7vbbpzHhbTe.

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. It is a structured, persistent, Byzantine fault-tolerant distributed storage that provides availability as well as a partial order on blocks of transactions. In this section we define the problem Narwhal addresses, its system and security model, as well as a high-level overview 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 encodes a causal 'happened-before' relation between the blocks (which we denote $b \longrightarrow b'$). 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.

Our Mempool abstraction supports 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 satisfies the following properties:

- **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 invoked.
- $1/2$-**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 digest certificates, which which have a small size. Moreover, the Causality and Containment properties guarantee that any consensus protocol that leverages Narwhal (even partially synchronous) achieves high-throughput despite periods of asynchrony. This is because once we agree on a block digest, eg. when synchrony is restored, we can safely totally order all its causally ordered blocks created during the periods of asynchrony. Therefore, *with Narwhal, different Byzantine consensus protocols mostly differ in the latency they achieve under different network conditions*, as we examine in Section 7. The Chain-Quality [25] property allows Narwhal to be used by Blockchains providing censorship resistance.

Last but not least, our mempool abstraction can scale out and support the increasing demand in consensus services. Therefore, we aim to achieve the above theoretical properties and at the time satisfy the following:

- **Scale out:** Narwhal's 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 blockchains [12, 33] implement a best-effort gossip Mempool. A transaction submitted to one validator is gossiped to all others. 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. In this section we extend step-by-step this basic design towards Narwhal to (i) reduce the need for double transmission when leaders propose blocks, and (ii) enable scaling out when more resources are available.

A first step 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, validators also need to ensure hashes represent available blocks, requiring them to download them before certifying a block – within the critical path of the consensus algorithm.

To ensure **availability**, as a second step, we consistently broadcast [18] the block, resulting in a certificate that the block will be available for download. A leader proposes a certificate, which is short and proves the block will be available. However, one certificate per Mempool block has to be included, and if the consensus temporarily loses liveness then the number of certificates to be committed may grow indefinitely.

As a third step, we add **causality** to propose a *single certificate for multiple Mempool blocks*: Mempool blocks include certificates of past Mempool blocks, from all validators. As a result, a certificate refers, to a block, and its full causal history. A leader proposing such a fixed-size certificate, therefore, proposes an extension to the sequence containing blocks from its full history. 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, two issues remain: (i) A very fast validator may force others to perform large downloads by generating blocks at a high speed; (ii) honest validators may not get enough bandwidth to share their blocks with others – leading to potential censorship.

A fourth step provides **Chain Quality** by imposing restrictions on block creation rate. Each block from a validator contains a round number, and must include a quorum of certificates from the previous round to be valid. As a result, a fraction of honest validators' blocks are included in any proposal. Additionally, a validator cannot advance to a Mempool round before some honest ones concluded the previous round, preventing flooding.

The final fifth design step is that of enabling **scale-out**. 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

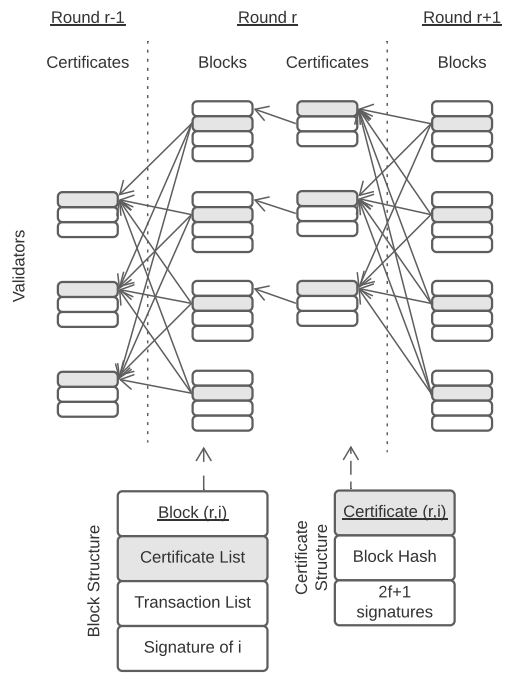

Figure 2. Three rounds of Narwhal. In round $r-1$ there are enough $(N-f)$ certified blocks, and validators start building blocks for round r. Each includes a batch of transactions and $N-f$ certificates for round $r-1$. Blocks for r can only include certificates from the previous round. Once a validator has a ready block it broadcasts it to others to form a certificate and then shares the certificate with all other validators to include at round $r+1$.

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 Narwhal in more detail.

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 high-throughput consensus. Finally, in Section 3.3 we address the main roadblock of previous DAG designs, garbage collection.

3.1 The Narwhal Mempool

The Narwhal Mempool is based on ideas from reliable broadcast [16] and reliable storage [2]. It additionally uses a Byzantine fault-tolerant version of Threshold clocks [24] 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* [16] 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. A certificate of availability includes $2f+1$ signatures, ie. at least $f+1$ honest validators have checked and stored the block. Thus, the block is available for retrieval when needed to sequence transactions. 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. Since a block contains references to certificates previous rounds, we get by an inductive argument that all blocks in the causal history are certified and available, satisfying causality[2].

3.2 Using Narwhal for consensus

Figure 3 illustrates how Narwhal can be combined with an eventually synchronous consensus protocol[3] (top) to enable

[2]More complete security proofs for all properties of a mempool are provided in the full paper.

[3]The network assumption for this protocol to work is stronger than what Narwhal requires.

	HS	Narwhal-HS	Tusk
Average-Case (Latency)	3	4	4.5
Worse-Case f (Crashes Latency)	$O(n)$	$O(n)$	4.5
Asynchronous (Latency)	N/A	N/A	7
Unstable Network Throughput	✗	✓	✓
Asynchronous Throughput	✗	✗	✓

Table 1. A comparison between Hotstuff and our protocols. We measure latency in RTTs (or certificates). Unstable Network is a network that allows for one commit between periods of asynchrony. By ✓we mean the throughput is the same as it would be under synchrony

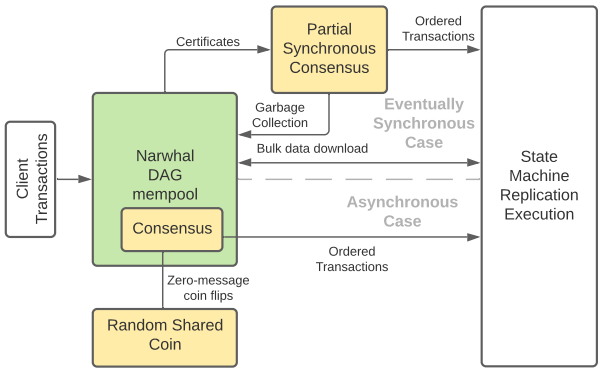

Figure 3. Any consensus protocol can execute over the mempool by occasionally ordering certificates to Narwhal blocks. Narwhal guarantees their availability for the SMR execution. Alternatively, Narwhal structure can be interpreted as an asynchronous consensus protocol with the (zero-message cost) addition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

high-throughput total ordering of transaction, which we elaborate on in Section 3.2; 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 5. Table 1 summarizes the theoretical comparison of vanilla Hotstuff with Narwhal-based systems, which we validate in our evaluation.

Narwhal-Hotstuff. Consensus algorithms operating in partial synchrony, such as Hotstuff [38] or LibraBFT [12], can leverage Narwhal to improve their throughput. 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 created in Narwhal. Upon commit, the full uncommitted causal history of 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 retrieved and transactions sequenced.

There are many advantages to leaders using Narwhal over sending a block of transactions directly. Even in the absence of failures, a 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.

Eventually-synchronous consensus protocols cannot provide liveness during asynchrony periods or when leaders are Byzantine. Therefore, with a naive mempool implementation, overall consensus throughput goes to zero during such periods. Narwhal, in contrast, continues to share blocks and form certificates of availability even under asynchronous networks, so blocks are always certified at maximal throughput. Once the consensus protocol manages to commit a digest, validators also commit its causal history, with no gaps for 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.

3.3 Garbage Collection

Another 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 [10]), garbage collection[4]. 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 blocks and certificates readily accessible to (1) help their peers catch up and (2) be able to process arbitrary old messages.

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 examine 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. Blocks from earlier

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

rounds can be safely be stored off the main validator 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 [18], 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. 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 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[5] 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 matches the worst-case theoretical lower bound [22].

[5]The properties of reliable broadcast are Validity, No duplication, Integrity, Consistency, and Totality (see page 112 and 117 of [18]).

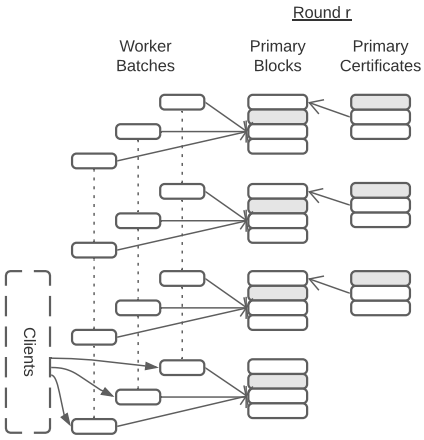

Figure 4. Scale-Out Architecture in Narwhal. Within each validator, there is one primary machine that handles the meta-data of building the DAG and multiple worker machines (3 shown in the example) each one streaming Transactions batches to other validators. The hashes of the batches are sent to the primary that includes them in the next block. Clients send transactions to worker machines at all validators.

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 use 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 4. 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, by induction, that 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 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 Tusk asynchronous consensus

In this section, we present Tusk, an asynchronous consensus algorithm for Narwhal that remains live under asynchrony or DDoS attacks. Tusk's theoretical starting point is DAG-Rider [28], from which it inherent its safety guarantees. Tusk modifies DAG-Rider into an implementable system and improves its latency in the common case.

Tusk validators operate a Narwhal mempool as described in Section 3.1, but also include in each of their blocks information to generate a distributed perfect random coin. Such a coin can be constructed from an adaptively secure threshold signature scheme [14] for which a key setup can also be performed under full asynchrony [31]. The Tusk algorithm gets the causally ordered DAG constructed by Narwhal and totally orders its blocks with zero extra communication. That is, every validator locally interprets its local view of the DAG and use the shared randomness to determine the total order.

To interpret the DAG, validators divide it into *waves*, each of which consists of 3 consecutive rounds. Conceptually, in the first round of a wave each validator proposes its block (and consequently all its causal history); in the second round each validator votes on the proposals by including them in their block; and in the third round validators produce randomness to elect one random leader's block in retrospect. Once the random coin for the wave is revealed, each validator v commits the elected leader block b of a wave w if there are $f+1$ blocks in the second round of w (in v's local view of the DAG) that refer to b. In which case, v then carefully orders b's causal history up to the garbage collection point. However, since validators may obtain different local views of the DAG, some may commit a leader block in a wave while others may not.

To guarantee that all honest validators eventually order the same block leaders, after committing the leader in a wave w, v sets it to be the next *candidate* to be ordered and recursively go back to the last wave w' in which it committed a leader. For every wave i in between w' and w, v checks whether there is a path between the candidate leader and the leader block b of wave i. In case there is such a path v orders b before the current candidate, set the correct candidate to be b and continue the recursion. This mechanism's safety argument is similar to DAG-Rider [28], and we provide here only an intuition.

Our theoretical main difference from DAG-Rider is the termination guarantee. While both protocols commit a block leader every constant number of rounds in expectation, Tusk's constants are better in the common case. By common case we mean networks where message delays are randomly distributed rather then controlled by the adversary. Specifically, DAG-Rider's waves consist of 4 rounds, and thus each block in the DAG is committed in expectation every 5.5 rounds in the common case. In Tusk we improve latency by considering waves that consists of 3 rounds. In addition, as an optimization, we piggyback the the first round of wave with the third round (that produces randomness) of the preceding one. As a result, each block in the DAG is committed in expectation every 4.5 rounds in the common case.

Figure 5 illustrates two waves of Tusk– the first wave is rounds 1-3, and the second is rounds 3-5 – with 4 validators and $f=1$. The elected leaders of waves 1 and 2 are determined at rounds 3 and 5, and we denote them L_1 and L_2,

respectively. There are an insufficient number of blocks in round 2 (less than $f+1$) that refer to ("vote for") L_1 and thus L_1 is not committed when round 3 is interpreted. However $f+1=2$ blocks in round 4 refer to L_2, and as a result L_2 is eventually committed. Since there is a path between L_2 and L_1, L_1 is ordered before L_2. Meaning that the sub-DAG causally dependent on L_1 is ordered first (by some deterministic rule), and then the same rule is applied to the sub-DAG causally dependent on L_2.

It is noteworthy that neither the random shared coin generation nor the consensus logic introduces any additional messages over Narwhal. Therefore, Tusk has zero message overhead, and the same theoretical throughput as Narwhal.

5.1 Safety intuition

Each round in the DAG contains of at least $2f+1$ blocks, and since any quorum of $2f+1$ blocks intersect with any quorum of $f+1$ blocks we get:

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

Given the above lemma, the recursive mechanism that is described above to order block leaders guarantees the following:

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

Since after ordering a block leader each validator orders the leader's causal history by some pre-define deterministic rule, we get that by the Containment property of Narwhal all honest validators agree on the total order of the DAG's blocks.

5.2 Liveness and latency

Inspired from the VABA [3] protocol we let an adversary commit to a communication schedule and then leverage a perfect shared coin to get a constant probability, despite asynchrony, to agree on a useful work produced therein.

As for termination, we use a combinatorial argument to prove in the full paper that:

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

To guarantee liveness against an adaptive asynchronous adversary, we use the randomness produced in the third round of a wave to determine the wave's block leader. Therefore, the adversary learns who is the block leader of a wave only after the first two rounds of the wave are fixed. Thus the $f+1$ blocks that satisfy the commit rule (from the above lemma) are determined before the adversary learns the block leader. Therefore, the probability to commit a block leader in each wave is at least $\frac{f+1}{3f+1} > 1/3$ even if the adversary fully controls the network.

In the full paper we prove the following lemma:

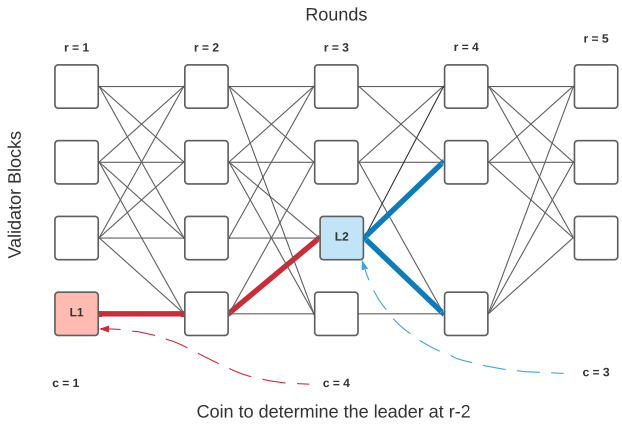

Figure 5. Example of commit rule in Tusk. Every odd round has a coin value that selects a leader of round $r-2$. If the leader has less than $f+1$ support (red) they are ignored, otherwise (blue) the algorithm searches the causal DAG to commit all preceding leaders (including red) and totally orders the rest of the DAG afterward.

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

As for latency in realistic networks in which message delays are distributed uniformly at random, we prove in the full paper the following:

Lemma 5. *In networks with random message delays, in expectation, Tusk commits each block in the DAG in 4.5 rounds.*

6 Implementation

We implement a networked multi-core Narwhal validator in Rust, using Tokio[6] for asynchronous networking, ed25519-dalek[7] for elliptic curve based signatures. Data-structures are persisted using RocksDB[8]. 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 of Narwhal and Tusk[9], HS-over-Narwhal[10] as well as all Amazon Web

[6] https://tokio.rs
[7] https://github.com/dalek-cryptography/ed25519-dalek
[8] https://rocksdb.org
[9] https://github.com/facebookresearch/narwhal/tree/tusk
[10] https://github.com/facebookresearch/narwhal/tree/narwhal-hs

Services orchestration scripts, benchmarking scripts, and measurements data to enable reproducible results[11].

We evaluate both Tusk (Section 5) and HS-over-Narwhal (Section 3.2). 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 [12]. We specify two versions of HotStuff (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 state-of-the-art technique [9, 19, 35] of validators batching transactions and sending them out of the critical path. Specifically, Batched-HS separates the task of data dissemination and consensus in the same way as Prism [9]. It first disseminates batches of transactions (called 'transaction blocks' in Prism), then the leader proposes hashes of batches to amortize the cost of the initial broadcast. These transaction batches perform a similar function to Narwhal's headers. The goal of this version is to show that this solution already gives benefits in a stable network but is not robust enough for a real deployment. We also open source our implementation of Batched-HS[12].

7 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 parts of the system inevitably crash-fail or suffer attacks. For an accompanying theoretical analysis of these claims see Table 1. Note that evaluating BFT protocols in the presence of Byzantine faults is still an open research question [11].

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 batch size of 500KB, a block size of 1KB, 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, under high load.

7.1 Narwhal as a Mempool

Figure 6 illustrates the throughput of Narwhal for varying numbers of validators, as well as baseline and batched HS:

Baseline HS. Baseline HS throughput (see Figure 6, Baseline-HS lines, left bottom corner), with a naive mempool as originally proposed, is quite low. With either 10 and 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 [37]. Performance evaluations [40] of LibraBFT [12] that uses Baseline HS, report throughput of around 500 tx/s.

Batched HS. For Batched HS without faults (see Figure 6, Batched HS lines), the maximum throughput we observe is 70,000 tx/s for a committee of 10 nodes, and lower (up to 50,000 tx/s) for a larger committee of 20. Latency before saturation is around 2 seconds. The almost 20x performance improvement compared to baseline HS is evidence that decoupling transaction dissemination from the critical path of consensus is the key to blockchain scalability. Performance decrease with the committee size similarly to other works [5].

Narwhal HS. The Narwhal-HS lines, 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 140,000 tx/sec at a latency consistently below 2 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 for various committee sizes, and 1 worker collocated with the primary per validator. We observe a stable latency at around 3 secs for all committee sizes as well as a peak

[11] https://github.com/facebookresearch/narwhal/tree/tusk/results/data
[12] https://github.com/asonnino/hotstuff/tree/d771d4868db301bcb5e3deaa915b5017220463f6

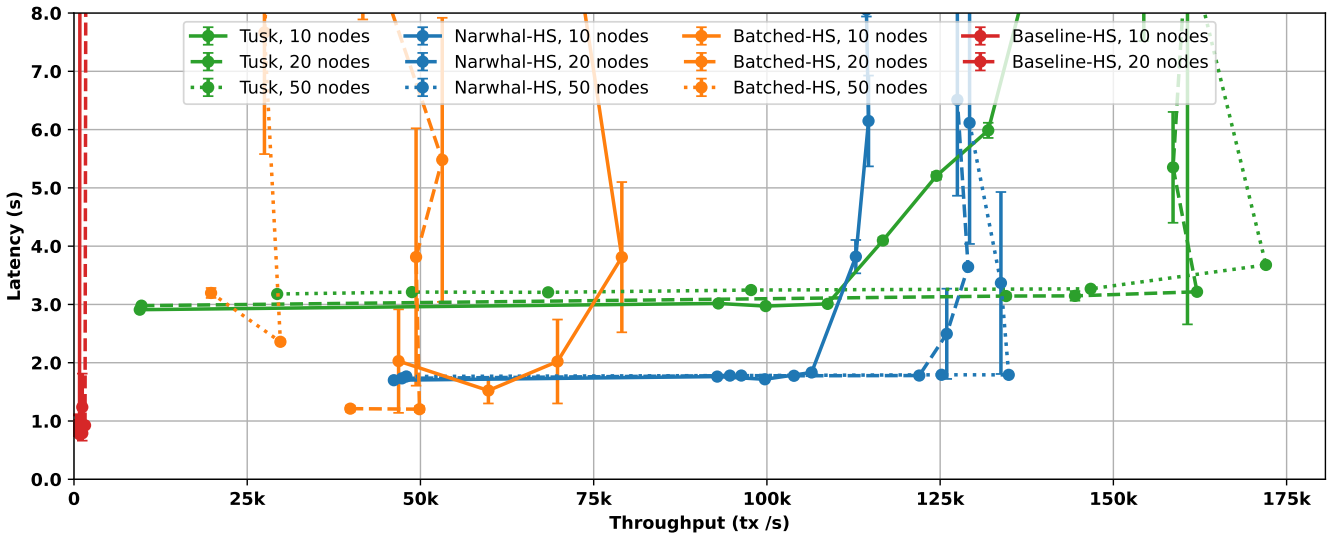

Figure 6. Comparative throughput-latency performance for the novel Narwhal-HotStuff, Tusk, batched-HotStuff and the baseline HotStuff. WAN measurements with 10, 20, and 50 validators, using 1 worker collocated with the primary. No validator faults, 500KB max. block size and 512B transaction size.

throughput at 170,000 tx/sec for 50 validators. This is by far the most performant fully asynchronous consensus protocol (see Section 8) 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.

7.2 Scale-Out using many workers

To evaluate the scalability of Narwhal, we measure throughput with 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.

The top Figure 7 illustrates the latency-throughput graph of Narwhal HS and Tusk for a various number of workers per authority whereas the bottom Figure 7 shows the maximum achievable throughput under various service level objectives (SLO). 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:

(#workers) ∗ (throughput for one worker)

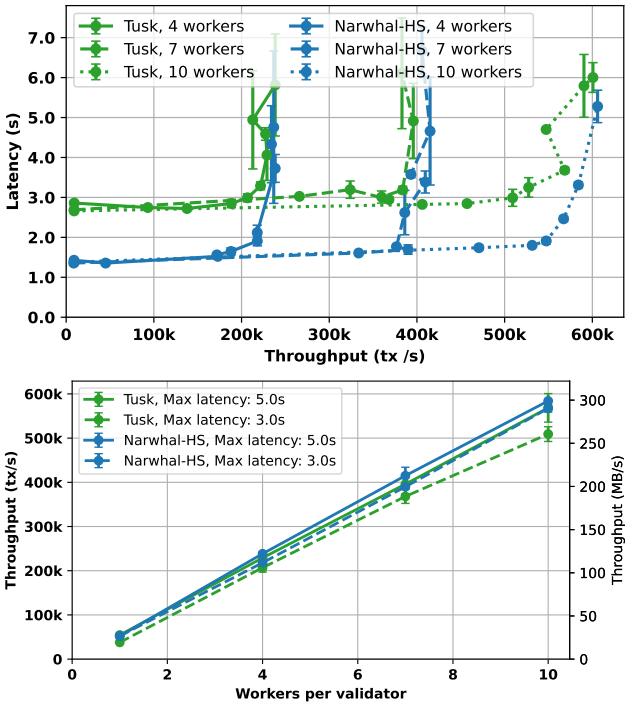

Figure 7. Tusk and HS with Narwhal latency-throughput graph for 4 validators and different number of workers. The transaction and batch sizes are respectively set to 512B and 1,000 transactions.

7.3 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 70k tx/sec to 2.5k tx/sec) and latency increases 10x.

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

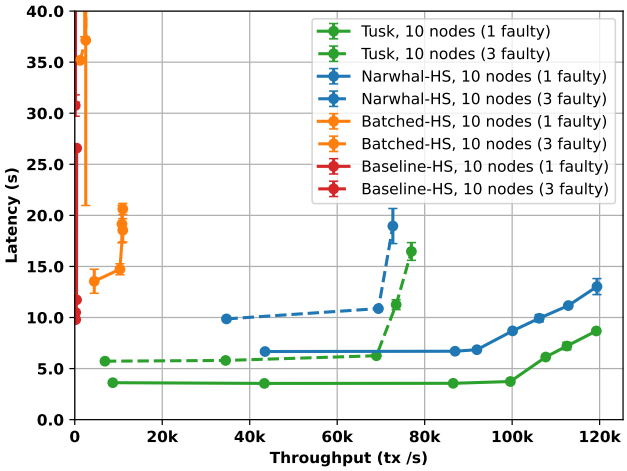

Figure 8. Comparative throughput-latency under faults. WAN measurements with 10 validators, using 1 worker collocated with the primary. One and three faults, 500KB max. block size and 512B transaction size.

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 4 sec under 1 fault, and less than 6 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 7 sec for 1 fault, and around 10 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.

8 Related work & Limitations

8.1 Performance

Widely-used blockchain systems such as Tendermint, provide 5k tx/sec [17]. 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 [34] and shows latency grows from a few seconds to a few hundred seconds, just through blocks being delayed. Han et al. [27] 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 [37], 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 Figure 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, 30, 39].

8.2 Asynchronous Consensus & Tusk

In the last 5 years the search of practical asynchronous consensus has captured the community [23, 26, 32, 36], because of the high robustness guarantees it promises. The most performant one was Dumbo2 [26] , 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 [28]. For the core algorithmic part we extend DAG-Rider: (i) we replace the classic reliable broadcast with our version described in Section 4.1; (ii) change the commit rule to have a better common-case latency; and (iii) remove weak links to allow garbage collection.

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. This make garbage collection impossible, as every block ever received has to be stored in-case it is pointed by a weak link. Thus, the strong notion of fairness of DAG-Rider is unimplementable within finite storage and purely theoretical. In Tusk we forbid weak links. We

instead re-inject transactions of uncommitted garbage collected blocks into new blocks in subsequent rounds. Thus, Instead of block-level fairness this achieves the more relevant metric of transaction-level fairness.

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.

8.3 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 [24], it is however, embedded in the consensus protocol and does not leverage it for batching and pipelining. Hashgraph [10] 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 [20] embeds a single-decision variant of PBFT [19] 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.

8.4 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.

9 Conclusion

We experimentally demonstrated the power of Narwhal and Tusk. Narwhal is an advanced mempool enabling Hotstuff to achieve throughput of 130, 000 tx/sec with under 2 seconds latency, in a deployment of 50 geographically distributed single-machine validators. Additionally, Narwhal enables any quorum-based blockchain protocol to maintain its throughput within 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 160, 000 TPS with about 3 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 supported by the Novi team at Facebook. We also thank the Novi Research and Engineering teams for valuable feedback, and in particular Mathieu Baudet, Andrey Chursin, Zekun Li, and Dahlia Malkhi 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] Vivek Bagaria, Sreeram Kannan, David Tse, Giulia Fanti, and Pramod Viswanath. Prism: Deconstructing the blockchain to approach physical limits. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 585–602, 2019.

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

[11] Shehar Bano, Alberto Sonnino, Andrey Chursin, Dmitri Perelman, and Dahlia Malkhi. Twins: White-glove approach for bft testing. *arXiv preprint arXiv:2004.10617*, 2020.

[12] 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.

[13] Alysson Bessani, Joao Sousa, and Eduardo EP Alchieri. State machine replication for the masses with bft-smart. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 355–362. IEEE, 2014.

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

[15] Gabriel Bracha. Asynchronous byzantine agreement protocols. *Information and Computation*, 75(2):130–143, 1987.

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

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

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

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

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

[21] Danny Dolev, Michael J Fischer, Rob Fowler, Nancy A Lynch, and H Raymond Strong. An efficient algorithm for byzantine agreement without authentication. *Information and Control*, 52(3):257–274, 1982.

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

[23] 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.

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

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] Satoshi Nakamoto. Bitcoin whitepaper, 2008.

[34] 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.

[35] A Pinar Ozisik, Gavin Andresen, Brian N Levine, Darren Tapp, George Bissias, and Sunny Katkuri. Graphene: efficient interactive set reconciliation applied to blockchain propagation. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 303–317, 2019.

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

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

[38] 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.

[39] 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.

[40] 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 Artifact Appendix

A.1 Abstract

We open-source the Rust implementation of Narwhal and Tusk, HS-over-Narwhal as well as all Amazon Web Services orchestration scripts, benchmarking scripts, and measurements data to enable reproducible results.

A.2 Description & Requirements

A.2.1 How to access Our implementation of Narwhal and Tusk is hosted on the following public GitHub repository:

`https://github.com/asonnino/narwhal`

The graphs in this paper are generated using the following commit:

`70dc862db090260dc77acdaa807b4584475bafc2`

Our implementation of HS-over-Narwhal is hosted on the following public GitHub repository:

`https://github.com/asonnino/narwhal/tree/narwhal-hs`

The graphs in this paper are generated using the following commit:

`add64984c62b7cf24dced6d40df52ae57032cfa3`

A.2.2 Hardware dependencies Narwhal does not require any particular hardware dependency. We however run all benchmarks on Amazon Web Services (AWS), using `m5.8xlarge` instances. They provide 10Gbps of bandwidth, 32 virtual CPUs (16 physical core) on a 2.5GHz, Intel Xeon Platinum 8175, and 128GB memory. For persistent storage, we equip every instance with an Ebs drive *gp2* of 200GB.

A.2.3 Software dependencies Every instance runs a fresh install of Ubuntu Server 20.04. We used the following AMI image: 'Canonical, Ubuntu, 20.04 LTS, amd64 focal image build on 2020-10-26'. Our implementation of Narwhal requires Rust 1.51+ and the following dependencies (installed through `apt-get`):

- `build-essential`
- `cmake`
- `clang`

A.2.4 Benchmarks The benchmarks described in this paper require no input data. We however open-source the raw data used to produce the graphs of Section 7: https://github.com/asonnino/narwhal/tree/master/benchmark/data/paper-data.

A.3 Set-up

To facilitate the experimental set-up, we open-source our AWS orchestration scripts. The scripts are written in Python 3.8 and can be installed as follows:

```
git clone https://github.com/asonnino/narwhal.git
cd narwhal/benchmark
pip install -r requirements.txt
```

A detailed tutorial to used the scripts is availlabble on GitHub: https://github.com/asonnino/narwhal/tree/master/benchmark. In particular, steps 1-4 of section 'AWS Benchmarks' set up an experimental testbed on AWS.

A.4 Evaluation workflow

A.4.1 Major Claims Our evaluation (Section 7) demonstrates the following major claims (Cx):

- (C1) Narwhal as a Mempool has advantages over the existing simple Mempool as well as straightforward extensions of it
- (C2) The scale-out is effective, in that it increases throughput linearly as expected.
- (C3) 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).
- (C4) Narwhal provides robustness when some parts of the system inevitably crash-fail or suffer attacks.

A.4.2 Experiments We run the following experiments.

Experiment (E1): [Common Case]: Evaluation of the common-case (no faulty validators) with various committee sizes.

[Preparation] Follow step 5 of the tutorial linked in Appendix A.3. In particular, set the following variable in `fabfile.py`:

```
bench_params = {
    'nodes': [10, 20, 50],
    'workers: 1,
    'collocate': True,
    'rate': [20_000, 50_000, 100_000],
    'tx_size': 512,
    'faults': 0,
    'duration': 300,
    'runs': 2,
}
```

Adjust in the input rate `rate` to reproduce the desired data point of Figure 6

[Execution] Run the following command: `fab remote`.

[Results] Follow step 6 of the tutorial linked in Appendix A.3. In particular, set the following variable in `fabfile.py`:

```
plot_params = {
    'faults': [0],
    'nodes': [10, 20, 50],
    'workers': [1],
    'collocate': True,
    'tx_size': 512,
    'max_latency': [3_500, 4_500]
}
```

Experiment (E2): [Scalability]: Evaluation with many workers on different machines (no faulty validators).

[Preparation] Follow step 5 of the tutorial linked in Appendix A.3. In particular, set the following variable in `fabfile.py`, where `X` is the desired number of workers:

```
bench_params = {
    'nodes': [4],
    'workers: X,
    'collocate': False,
    'rate': [100_000, 300_000],
    'tx_size': 512,
    'faults': 0,
    'duration': 300,
    'runs': 2,
}
```

Adjust in the input rate `rate` to reproduce the desired data point of Figure 7.

[Execution] Run the following command: `fab remote`.

[Results] Follow step 6 of the tutorial linked in Appendix A.3. In particular, set the following variable in `fabfile.py`:

```
plot_params = {
    'faults': [0],
    'nodes': [4],
    'workers': [1, 4, 7, 10],
    'collocate': False,
    'tx_size': 512,
    'max_latency': [3_500, 4_500]
}
```

Experiment (E2): [Crash-faults]: Evaluation with 0, 1 and 3 crash-faults.

[Preparation] Follow step 5 of the tutorial linked in Appendix A.3. In particular, set the following variable in `fabfile.py`, where `X` is the desired number of crash-faults:

```
bench_params = {
    'nodes': [10],
    'workers: 1,
    'collocate': True,
    'rate': [30_000, 70_000],
    'tx_size': 512,
    'faults': X,
    'duration': 300,
    'runs': 2,
}
```

Adjust in the input rate `rate` to reproduce the desired data point of Figure 8.

[Execution] Run the following command: `fab remote`.

[Results] Follow step 6 of the tutorial linked in Appendix A.3. In particular, set the following variable in `fabfile.py`:

```
plot_params = {
    'faults': [0, 1, 3],
    'nodes': [10],
    'workers': [1],
    'collocate': True,
    'tx_size': 512,
    'max_latency': [10_000, 5_000]
}
```

A.5 General Notes

Please be mindful that the experiments on AWS described above can be (very) expensive.