BlazeDAG

a hyper-scalable L1 riding on Narwhal

Flavius Burca
flavius.burca@arthera.net

Abstract

Recent works and emerging Proof-of-Stake blockchains rely on DAG (Direct Acyclic Graph) structures to achieve high TPS. In mathematics, particularly graph theory, a graph structure can be briefly defined as a finite set of N nodes, each two of which can have between them an edge. Therefore, one can uniquely define a graph $G = (N, M)$ where N is the set of nodes and M consists of pairs (A, B) from $N \times N$ where A and B are distinct. If all pairs (A, B) in set M satisfy $(A, B) = (B, A)$, the graph is undirected. Otherwise, G is a directed graph, meaning that edges define not only a link between nodes but also a direction. In the environment of directed graphs, a finite set of n nodes $A_1, A_2, \dots, A_n, A_1$ is defined as a *cycle* or *loop* if the directed edges $(A_1, A_2), (A_2, A_3), \dots, (A_n, A_1)$ exist. **A DAG is a directed graph with no such loops.**

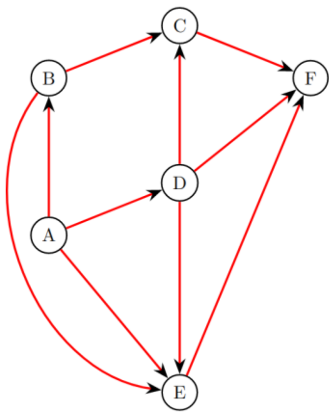


Figure 1. Example of a directed graph

In this paper we show that it's possible to reach a consensus with minimum delay on the global ordering of transactions using DAG transports that provide causality and non-equivocation. This approach is an improvement over existing partial synchrony DAG consensus protocols like Tusk and Bullshark where transactions are added to the DAG only after consensus steps/timers have been completed, introducing "empty" time slots when no action is executed. It is true, however, that during "empty" slots, the DAG keeps accumulating transactions, but the addition of timers/delays by the consensus protocols delays block finality.

We propose an optimized approach to achieve BFT consensus that is entirely independent of the DAG transport and does not add any constraints to it. We show that a maximum of two broadcast latencies are needed to reach consensus when the network is stable and that no additional steps/timers are needed, allowing the consensus to progress as fast as possible and speeding up block finality.

While BlazeDAG draws on recent work from Narwhal and Tusk and Bullshark, it does not fully adhere to them. The proposed architecture is optimized to spread transactions using a DAG structure at network speed to achieve remarkably high transaction throughput. To determine the total order of accumulated transactions, network participants interpret their DAG locally without exchanging any messages. As long as the DAG transport can provide reliable and causally ordered transaction dissemination, consensus can be reached in a simple manner.

We show that a DAG transport is sufficient to solve most of the BFT consensus problem by enabling reliable, causally ordered broadcast of transactions. Moreover, using a DAG transport that adheres to these rules, a partial synchrony model to achieve BFT consensus on the total order of transactions without adding any DAG delay can be achieved efficiently, without adding any DAG constraints.

We need to separate the following responsibilities to scale BFT consensus:
1. a transport to reliably distribute blocks of transactions that regulates communication, optimizes throughput, and tracks only the causal ordering of transactions in a DAG structure;
2. a global sequential commit order for transactions;

DAG messages contain blocks of transactions or transaction references in each message. Messages contribute to the total ordering of committed transactions at no additional cost, have a non-equivocating property, and carry causality information that simplifies consensus protocol design.

Existing solutions

At the time of writing this paper, the recent works in DAG-based systems are:

Narwhal: a DAG transport with a layer-by-layer structure. Each layer has at most one message per sender and refers to $2f + 1$ messages in the preceding layer.

Narwhal-HotStuff: a BFT consensus protocol based on HotStuff where Narwhal plays the role of a “mempool”. The DAG is only used to spread transactions while Narwhal-HS adds messages outside of Narwhal.

Tusk and its ancestor *DAG-Rider* built randomized consensus “riding” on Narwhal. The protocols have zero message overhead (no network communication). Both protocols group DAG layers into waves. The DAG waits for the consensus protocol to inject an input value in each wave. While this does not appear to delay the DAG, it takes an average of 4.5 rounds to reach block finality.

Bullshark is the latest protocol for the partial synchrony model built on Narwhal. It has the same wave structure like DAG-Rider and Tusk, but it’s designed to have 8-layer waves. Each layer serves a different step in the protocol. Bullshark is also a zero-message overhead protocol, however, due to its wave-by-wave structure, the DAG needs to wait for additional steps/timers to insert transactions into the DAG, thus delaying the DAG. If the leader of a wave is faulty or slow, some DAG layers need to wait to fill until consensus timers expire.

Below is a comparison of these protocols:

Protocol	***Type***	***Zero-message overhead***	***DAG delay***	***Length of commit chain****
Aleph	ASYNC	No	Yes (coin-tosses)	constant
Narwhal-HS, 2021	PS	No	No	6
DAG-Rider, 2021	ASYNC	Yes	Yes (coin-tosses)	4
Tusk, 2021	ASYNC	Yes	Yes (coin-tosses)	3
Bullshark, 2022	PS with ASYNC fallback	Yes	Yes (timers)	2
BlazeDAG	PS	Yes	No	2

PS = Partial Synchrony
ASYNC = Asynchronous

BlazeDAG follows the trend as a partial synchrony consensus with zero-message overhead that *does not add any DAG delays* and *has a commit chain length of 2, outperforming current protocols by far.* It’s a perfectly balanced solution that is simple, elegant, and fast that can progress at DAG network speed compared to existing protocols.

Benchmarks conducted with 20 validators in geo-distributed deployments on AWS show a 830k TPS throughput at the moment of writing this paper.

Overview

System model, goals, and assumptions

We assume a blockchain system with a set of n validators V = $\{v_1, . . . , v_n\}$. We assume a computationally bounded adversary that can control the network and can corrupt up to f_{max} validators where $f_{max} < \frac{n}{3}$ where n is the total number of validators.

We say that corrupt validators are Byzantine or faulty and all other validators are honest or correct. Byzantine validators may act arbitrarily: vote incorrectly, don’t respond, change protocol rules, etc. In contrast, honest validators always follow protocol rules.

To capture real-world networks, we assume eventually asynchronous reliable communication links among honest validators. That is, there is no bound on message delays and there is an unknown, but finite number of messages that can be lost. Moreover, for simplicity, we assume that recipients can verify the sender's identities.

We also assume a known interval of time and say that the execution of the protocol is eventually synchronous if there is a global stabilization time (GST), after which all messages sent among honest validators are delivered within time. An execution is synchronous if GST occurs at time 0 and asynchronous if GST never occurs.

DAG Transport

Validators store messages delivered in a local DAG. A message m inserted into the local DAG needs to have the following guarantees:

> **Reliability**: there are sufficient copies m stored by validators such that eventually, all honest validators can download it.
>
> **Non-Equivocation**: messages sent by each validator are numbered. If a validator delivers a message with index k from a particular sender, other validators deliver the same message with index k from the initial sender.
>
> **Causal Ordering**: the message references previous messages the sender delivered (including its own).

Because real-world networks can only provide asynchronous, eventually-reliable communication, the local copies of the DAG may differ for network participants. This means each validator must reach an independent conclusion of total ordering. However, given a DAG transport that can provide Reliability, Non-equivocation, and Causal Ordering, the protocol can be simple enough, as we can see further.

The chosen DAG transport is Narwhal and we model the consensus logic on top of DAG rounds. We show that Narwhal's architecture allows us to piggy-back on its broadcast mechanism to add consensus logic without delaying DAG rounds.

Randomator

We use a global random coin that is unpredictable by the adversary. An instance $w, w \in \mathbb{N}$ of the coin is invoked by a validator $v_i \in \Pi$ by calling $choose\_leader_i(w)$. This call returns a validator $v_j \in \Pi$, which is the chosen leader, for instance, $w.$ Let X_w be the random variable that represents the probability that the coin returns validator v_j as the return value of the call $choose\_leader_i(w)$. The global perfect coin has the following guarantees:

- **Agreement:** If an honest validator v_i calls $choose\_leader_i(w)$ with result v_1 and another honest validator v_j call $choose\_leader_j(w)$ with result v_2 , then $v_1 = v_2$
- **Termination**: If at least $f+1$ honest validators call $choose\_leader_i(w)$, then every $choose\_leader_i(w)$ call eventually returns
- **Unpredictability**: As long as less than $f+1$ honest validators call $choose\_leader_i(w)$, the return value is indistinguishable from a random value except with negligible probability ϵ. Namely, the probability pr_i that the adversary can guess the returned validator v_j of the call $choose\_leader_i(w)$ is $pr \leq Pr[X_w = v_j] + \epsilon$

- **Fairness**: The coin is fair, i.e. $\forall w \in \mathbb{N}\ and\ \forall v_j \in \Pi\ then\ Pr[X_w = v_j] = 1/n$

See DAG-Rider for more details on how a coin implementation can be integrated into the DAG construction. It is important to note that the above-mentioned implementations satisfy Agreement, Termination, and Fairness with theoretical information guarantees. That is, the assumption of a computationally bounded adversary is required only for the unpredictability property. As we later prove, the unpredictability property is only required for Liveness. Therefore, since similarly to DAG-Rider generating randomness is the only place where cryptography is used, the Safety properties of BlazeDAG are post-quantum secure.

Reliable broadcast

DAG-Rider defines a reliable rebroadcast abstraction to allow the use of efficient gossip protocols. Formally, each sender v_k can broadcast messages by calling $r\_bcast_k(m, r)$, where m is a message and $r \in \mathbb{N}$ is a round number. Every validator v_i has an output $r\_deliver_i(m, r, v_k)$, where m is a message, r is a round number, and v_k is the validator that called the corresponding $r\_bcast_k(m, r)$. The reliable broadcast abstraction provides the following guarantees:

- **Agreement:** if an honest validator v_i outputs $r\_deliver_i(m, r, v_k)$, then every honest validator v_j eventually outputs $r\_deliver_j(m, r, v_k)$
- **Integrity**: for each round $r \in$ N and validator $v_k \in \Pi$, an honest validator v_i outputs $r\_deliver_i(m, r, v_k)$ at most once regardless of m.
- **Validity**: if an honest validator v_k calls $r\_bcast_k(m, r)$, then every honest validator v_i eventually outputs $r\_deliver_i(m, r, v_k)$

BlazeDAG

Building on Narwhal

Our model assumes an asynchronous DAG protocol where each validator holds a local copy of the DAG and advance rounds as soon as $2f + 1$ nodes from the current round are delivered. BlazeDAG deterministically achieves better latency compared to existing protocols and needs a maximum of two DAG rounds to achieve finality. More important, our protocol does not delay the DAG in any way, allowing the system to advance at network speed.

We start from the highly scalable and efficient DAG construction of Narwhal where transaction dissemination is reliable and decoupled from consensus. Each DAG message contains a block of transactions and a set of references to $N - f$ previous messages, and the DAG keeps growing at network speed.

The consensus logic of BlazeDAG adds no communication overhead. Validators look at their local version of the DAG and order all the blocks with no external communication, i.e., the DAG is interpreted as a consensus protocol. Validators may see slightly different versions of the DAG at any point in time due to the asynchronous nature of the network. The main challenge of a consensus protocol is to guarantee that all validators agree on the same total order of blocks.

The DAG has a round-based structure where each vertex is associated with a round number. Each validator maintains the current local round r, starting at zero. Validators continuously receive transactions from clients and accumulate them into a transaction list. 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.

Validators reliably broadcast each block they create to ensure the integrity and availability of the block. For practical reasons, we do not implement the standard push strategy that requires quadratic communication. Instead, we use a pull strategy to ensure we do not pay the communication penalty in the common case. 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:

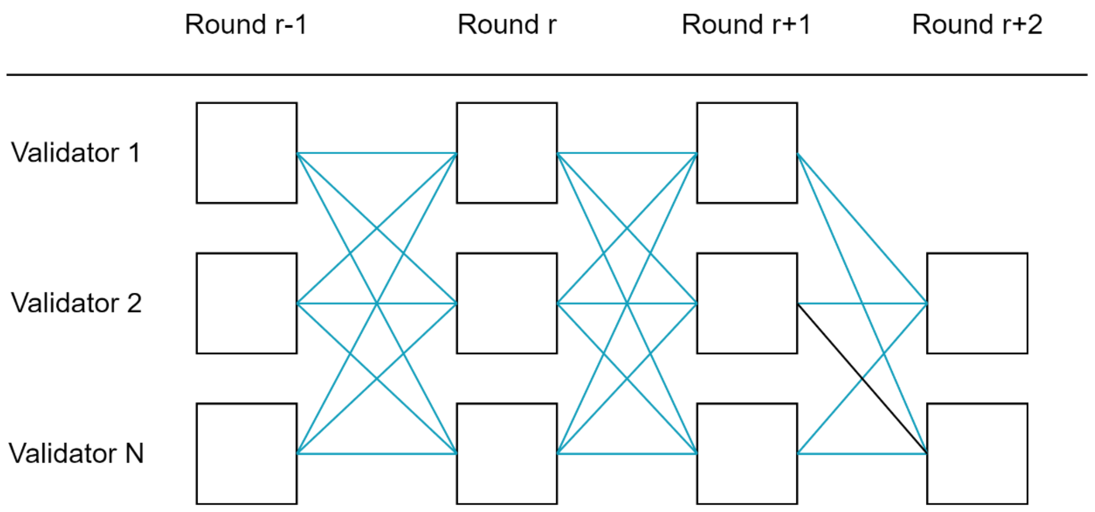
1. contain a valid signature from its creator
2. be at the local round r of the validator checking it
3. be at round 0 (genesis), or contain certificates for at least $2f + 1$ blocks of round $r - 1$
4. be the first one received from the creator for round r

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.

A certificate of availability includes $2f + 1$ signatures, i.e. 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 prevents equivocation of each block. Finally, since a block contains references to certificates from previous rounds, we get by an inductive argument that all blocks in the causal history are certified and available, satisfying causality.

The local view of the DAG for each validator at a point in time is illustrated below:



Non-equivocation
For any validator v and round r, if two validators have a block from v in round r in their local views, then both parties have the same block (same transactions and references). Hence, recursively, the block's causal history in both views is the same. The non-equivocation property of the DAG prevents Byzantine validators from lying, drastically simplifying the consensus logic.

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, 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 DoS 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.

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 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.

Reliable Broadcast
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).

The BlazeDAG protocol

The proposed algorithm operates in a wave-by-wave manner. It achieves total ordering of blocks available in the DAG with zero extra communication. Every validator locally interprets its local view of the DAG and uses the coin to determine the leader of the current round.

Briefly, the method works as follows:

- **Waves**. Each view V_w has a leader known to everyone elected using the Randomator’s $choose\_leader(w)$. The leader of V_w issues a **block proposal**, followed by **votes** or **complaints** for the leader’s proposal embedded inside the DAG.
- **Proposals**. When a leader enters V_w, it broadcasts its proposed block as $P(w)$. Initially $w = 1$. The leader’s proposal $P(w)$ contains the leader’s block and its $N - f$ causal predecessors.
- **Votes**. The first broadcast of a validator that causally follows $P(w)$ in V_w is a vote for the leader’s proposal. All validators in V_w (including the leader) deliver their vote for V_w.
- **Complaints**. If $P(w)$ is not received in a predefined timeframe, validators broadcast a complaint indicate that V_w is not progressing. In V_w a party can either vote or complain, but not both.
- **Commit**. A commit happens in V_w if there are $f + 1$ votes for $P(w)$.
- **Next wave**. Validators enter V_{w+1} if there are $f + 1$ votes for $P(w)$ or $2f + 1$ complaints.

Ordering Commits
When a validator observes that a leader’s $P(w)$ becomes committed, it orders newly committed transactions as follows:

1. Let r’ be the highest wave w’ < w for which $P(w')$ is in the causal history of $P(w)$. $P(w')$ is recursively ordered.
2. The remaining causal predecessors of $P(w)$ which were not ordered are appended to the sequence.

The final sequence of blocks comes from topologically sorting the DAG using a pre-order depth-first search algorithm.

The protocol is completely independent of DAG layers. Proposals, votes and complaints are not dependent on DAG rounds, allowing the DAG to continue spreading messages. This is possible because we don't need to consider leader equivocation, because DAG prevents it. Also, a leader does not need to justify its proposal because it is justified through its causal history. Finally, the safety of a commit is conditioned by advancing to the next wave after $f + 1$ votes or $2f + 1$ complaints are received, guaranteeing intersection with $f + 1$ commit votes.

Pseudo-code

- *let R_r be the round r*
- *let V_w be the wave w*
- *let L_w be the elected leader of wave w*
- *let $LH_r(w)$ be the header proposed by L_w in round r that contains information about wave w*
- *let $LC_r(w)$ be the certificate of availability broadcasted by L_w in round $r$that contains information about wave w*

```
1. A new DAG round $R_r$ starts
2. All parties enter $V_w$
3. Each party elects leader $L_w$ for $V_w$ and starts a timer for $V_w$
4. $L_w$ broadcasts his header $LH_r(w)$ for wave $w$ to all parties. This is the first broadcast of $L_w$ for wave $V_w$. Other parties broadcast their own headers as well.
5. All parties include the vote for $LH_r(w)$ in their certificate for $R_r$ and broadcast the certificate. Their certificate causally follows the leader's proposal and includes $N - f$ certificates from $R_{r-1}$.
6. After receiving $f + 1$ votes for $LH_r(w)$, all parties enter wave $V_{w+1}$
7. If the timer expires before receiving $f + 1$ votes, a complaint will be included in the next broadcast of $R_{r+1}$
8. After receiving $2f + 1$ complaints in $R_{r+1}$ all parties enter wave $V_{w+1}$
```

Consensus operates in epochs of R rounds each. Before the beginning of an epoch, each node locks some stake. Nodes can freely lock any value from their accounts into stake. The stake remains locked for the subsequent epoch and only gets unlocked at the start of the epoch after. The stake may also be delegated: a node may lock some stake and delegate it to another node to act with the authority of such stake for the epoch.

At the beginning of each period, all nodes determine the stake weight of all nodes in the system, denoted as n → ψ(n) — this includes the stake locked by node n, as well as any directly or indirectly delegated stake locked by other nodes for n. We denote the totality of stake as Ψ = n ψ(n). All decisions relating to leader election are interpreted based on the total stake during the epoch.

Lemma 1:
If L_w is honest, at the end of round R_r *at least* $f + 1$ *parties have* $LC_r(w)$

Proof:
$LC_r(w)$ includes $2f + 1$ votes. Because at most f of them can be Byzantine, at least $f + 1$ honest parties have checked and stored the certificate.

Lemma 2:
At the end of round R_{r+1}, a validator will receive $2f + 1$ complaints if the $LC_r(w)$ is absent from round R_r.

Proof:
All $2f + 1$ honest parties of round R_r will complain about the missing $LC_r(w)$ in round R_{r+1}. Due to the reliable broadcast property of the DAG transport, at least $2f + 1$ parties will receive the complaints.

Protocol scale-out

We aim to achieve horizontal scalability for BlazeDAG by allowing a validator to increase its processing capacity by adding more worker nodes. To do this, we need to ensure all components of the protocol can scale: transport and transaction processing.

Phase 1 – Scaling the DAG Transport with Narwhal

BlazeDAG relies on the Narwhal DAG transport to aggregate transactions at network speed, ensuring partial liveness for low-latency transaction submission for clients. The protocol has a master node and worker nodes. The master node runs the 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 master 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.

Transferring and storing transaction data is a parallel process. Worker nodes create batches of transactions and send them to worker nodes of each of the other validators. Once a quorum of workers has received an acknowledgment, the cryptographic hash of the batch is shared with the master node of the validator for inclusion in a block. Worker nodes are fronted by a load balancer to ensure all worker nodes receive transactions data at a similar rate.

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

The master node's blocks are small because they only include hashes of transactions. Worker nodes handle actual transaction data, and they constantly create and share batches in the background. Small batches, in the order of a few hundred to a few thousand transactions ensure transactions do not suffer more than some maximum latency. As a result, most of the batches are available to other validators before the master's blocks arrive. This reduces latency since there is less wait time from receiving a master node block to signing it and while waiting to advance the round (since we still need $2f + 1$ master blocks) workers continue to stream new batches to be included in the next round's block.

Block ordering is performed only by the master node. Committing a leader proposal and its causal history is a matter of topologically sorting the DAG back to the last committed proposal using a pre-order depth-first search.

Phase 2 – Scaling transaction processing (BlazeTM)

The ordered sequence of blocks is passed to the Executor to execute the actual transactions in a way that that is consistent meaning that all validators must reach the same outcome after performing the execution. Our goal is to accelerate execution by optimistically running transactions in parallel, using all available worker nodes.

Recent advances like BlockSTM managed to achieve in-memory parallelism, but this is still limited by the hardware available to the validator node, since BlockSTM needs to run on a single machine.

We introduce BlazeTM, an execution engine that leverages a distributed high-performance transactional memory implementation to enable parallel processing of transactions on multiple machines. The engine uses a high-performance active replication protocol that wraps transactions in transactional request messages and executes them on all worker nodes in the same order. To achieve this, BlazeTM employs a speculative concurrency control protocol (SCC), which starts transaction execution speculatively upon delivery by the network service, assuming the optimistic order as the processing order. This approach avoids atomic operations, thereby enabling transactions to reach maximum performance between the optimistic and final delivery. The protocol also avoids conflict detection and other complex mechanisms to ensure that a sequence of transactions is executed within their final notifications.

The SCC protocol stores a list of committed versions for each shared object, enabling read-only transactions to execute in parallel to write transactions. As a result, write transactions are broadcasted between all workers, while read-only transactions are directly delivered to a single worker that will process it locally (each worker has the same state). Workers already have transaction data, so a minimal pull-mechanism is implemented to synchronize their state. Workers typically sit on the same gigabit LAN, meaning that synchronizing transaction data between them can be done extremely fast.

Garbage collection

The DAG is a local construct that will eventually converge to the same version in all validators, but the timing of this convergence is not guaranteed. As a consequence, validators may need to retain all blocks and certificates in an accessible manner to facilitate their peers in catching up and processing old messages.

The DAG follows a rigorous round-based approach, allowing validators to assess the validity of a block by utilizing information solely from the current round, ensuring the 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 are not required to examine the entire history to verify new blocks. However, if two validators discard different rounds, it may lead to disagreement on a block's causal history and a different ordering of histories when a new block is committed. This concern is addressed by consensus through agreement on the garbage collection round. Blocks from previous rounds can be safely stored off the primary validator, and later messages from prior rounds can be disregarded.

At first glance, it may seem plausible for an adversary to censor transactions by delaying them adequately for garbage collection to remove them from being actively sent. However, this is not the case. An honest node that discards an old round that did not make it into the DAG can re-introduce transactions to a later round. Therefore, although the actual DAG blocks may be censored, all transactions will ultimately be included in blocks.

Analysis

The protocol is minimally integrated into the DAG transport. BFT logic is embedded into the DAG structure simply by injecting payloads into the DAG transport in the form of proposals, votes, and complaints. Notably, the DAG transport is never slowed down, and the reliability and causality properties of the DAG transport make arguing about safety and liveness relatively easy.

Scenario 1: The leader is Byzantine

If the leader of a wave L_w is Byzantine, validators will eventually time out and broadcast a complaint message. When a validator sees $2f+1$ complaints about a wave, it enters the next wave L_{w+1}. DAG transmission continues normally and spreads transactions. Let's assume the first wave V_w proceeds normally and parties enter V_{w+1}. However, no message marked V_{w+1} arrive from L_{w+1}. DAG transmission continues, unaffected by the failure of L_{w+1} meaning that faulty waves have utility in spreading transactions in the DAG. After a given timeout, validators complain about V_{w+1}. After $2f+1$ complaints are collected, L_{w+2} broadcasts $P(w+2)$. His proposal includes all messages that have accumulated in its causal history.

Scenario 2: The leader is late

This scenario is the same as the previous one with L_{w+1} actually issuing $P(w+1)$, which is too slow to arrive at other validators. These validators complain about a wave failure and will enter V_{w+2} committing $P(w+1)$. Since we know that $P(w+2)$ causally follows P$(w+1)$, when $P(w+2)$ commits, it will indirectly commit $P(w+1)$.

Safety

Lemma 1:
We assume $P(w)$ is committed. If w' is the minimal wave, where $w' > w$ such that $P(w')$ becomes committed, then for every $w \leq q < w'$, $P(q+1)$ causally follows $P(w)$.

Proof:
If $P(w)$ is committed, then $P(w)$ has $f+1$ votes. The leader's proposal $P(q+1)$ can be valid in two cases:

i) $P(q+1)$ has $f+1$ valid votes in V_q. This occurs only for $q = w$, if $w' \geq q + 1$ is the minimal wave that becomes committed.
ii) If for $w \leq q < w'$, $P(q+1)$ references $2f + 1$ valid complaint messages.

In the first case, valid votes for V_w causally follow $P(w)$, and $P(w+1)$ causally follows $P(w)$.

In the second case, one of $2f + 1$ complaint messages for V_q is sent by a party that sent a valid vote for V_w. By definition, a vote in V_w must precede a complaint in V_q, otherwise it is not considered a (valid) vote. Hence, $P(q+1)$ causally follows a complaint in V_q which follows $P(w)$.

Lemma 2:
If $P(w)$ is committed, then every valid $P(q)$, where $q > w$, causally follows $P(w)$.

Proof:
Following an induction on *Lemma 1*

Lemma 3:
If an honest validator commits $P(w)$, and another honest validator commits $P(w')$, where $w' > w$, then the sequence of transactions committed by $P(w)$ is a prefix of the sequence committed by $P(w')$.

Proof:
When $P(w')$ becomes committed, the commit ordering rule recursively applies to valid proposals in its causal past. According to *Lemma 2*, the committed $P(w)$ is a causal predecessor of every valid $P(s)$, for w $< s \leq w'$, and by recursion, it will eventually commit.

Liveness

We assume a known interval of time and say that execution of the protocol is eventually synchronous if there is a Global Stabilization Time (GST) after which all messages sent among honest validators are delivered within time. An execution is synchronous if GST occurs at time 0, and asynchronous if GST never occurs.

Waves are semi-synchronized through Reliable Broadcast after GST. Let T_Δ be the upper bound on a DAG broadcast after GST. If an honest leader L_w enters V_w within T_Δ, all honest validators will enter V_w within T_Δ. In $2 * T_\Delta$ all validators will receive $P(w)$ and all votes for V_w. Assuming the wave timer is set to $3 * T_\Delta$, once validators enter V_w, a future wave V_q where w $< q$ will not interrupt a commit in V_{w} because validators must collect $f+1$ votes or $2f+1$ complaints for V_{w}, thus entering V_{q} is not possible.

Communication cost

Without Narwhal to deliver messages reliably, BlazeDAG would need a reliable broadcast implementation over authenticated channels or signature verfications for each broadcast that would eventually lead to quadratic communication among parties. Choosing Narwhal as the underlying DAG transport, BlazeDAG benefits from Narwhal's certificates of availability, eliminating the need for quadratic communication.

Commit latency

The commit in BlazeDAG is max. two Narwhal DAG rounds: one proposal followed by votes.

Conclusion

Recent blockchain systems constantly push the boundaries of how many TPS a distributed system can process. However, performance under real network conditions or during attacks has a serios impact on these systems, as it was proved repeatedly over time. Also, scaling approaches often refer to sharding. We present a different approach, where blockchain scaling can be done through the horizontal scaling of data-center resources.

We implemented BlazeDAG on top of Narwhal and ran benchmarks to experimentally demonstrate its power. The system was able to reach *up to 830,000 tx/sec with 2 second latency* in an AWS deployment with 20 geographically distributed single-machine validators over 5 continents at the moment of writing this paper. The code is available in our GitHub repository and can be made available on request. A significant improvement in BlazeDAG is the commit latency of 2 in terms of DAG rounds: in round r the leader broadcasts its proposal followed by $f+1$ votes or $2f+1$ or complaints in $r+1$.

Additionally, the protocol can maintain its throughput within periods of asynchrony or faults if the consensus layer is live. This is because BlazeDAG does not delay the DAG in any way through additional steps or timers. The scale-out design of Narwhal and the simplicity of BlazeDAG allows the presented system to increase throughput beyond this limit to potentially millions of transactions per second through scale-out, without impacting latency.

We demonstrated that BlazeDAG is a major improvement to existing DAG protocols (Tusk, Bullshark, Aleph and DAG Rider). A simple consensus can be implemented on top of a reliable DAG transport (Narwhal), allowing the DAG to progress at network speed with no added delays, achieving record TPS numbers, very fast block confirmation and superior scalability by horizontally scaling validator nodes.

References

[1] Narwhal and Tusk: A DAG-based Mempool and Efficient BFT Consensus
https://arxiv.org/pdf/2105.11827.pdf

[2] Bullshark: DAG BFT Protocols Made Practical
https://arxiv.org/pdf/2201.05677.pdf

[3] All You Need is DAG
https://arxiv.org/pdf/2102.08325.pdf

[4] Blockmania: from Block DAGs to Consensus
https://arxiv.org/abs/1809.01620

[5] Block-STM
https://arxiv.org/pdf/2203.06871.pdf

[6] Aleph: Efficient Atomic Broadcast in Asynchronous Networks with Byzantine Nodes
https://arxiv.org/pdf/1908.05156.pdf

[7] DAG Meets BFT - The Next Generation of BFT Consensus
https://decentralizedthoughts.github.io/2022-06-28-DAG-meets-BFT/

[8] Solana: A new architecture for a high performance blockchain
https://solana.com/solana-whitepaper.pdf

[9] Asymptotically Optimal Validated Asynchronous Byzantine Agreement
https://doi.org/10.1145/3293611.3331612

[10] Mir-BFT: High-Throughput BFT for Blockchains
http://arxiv.org/abs/1906.05552

[11] Divide and Scale: Formalization of Distributed Ledger Sharding Protocols
https://arxiv.org/abs/1910.10434