Bullshark

DAG BFT Protocols Made Practical

Alberto Sonnino
Acknowledgements

Alexander Spiegelman
Neil Giridharan
Alberto Sonnino
Lefteris Kokoris-Kogias

Work done at Facebook Novi
Byzantine Fault Tolerance

> 2/3
Consensus on top of Narwhal

Goal of this project

**Simple**
- Zero-message overhead
- No view-change
- No common-coin

**Performant**
- Take advantage of Narwhal
- Exploit periods of synchrony
Current Designs

- Monolithic protocol sharing transaction data as part of the consensus
- Optimize overall message complexity of the consensus protocol
- Complex & Error-prone view-change protocol
Current Designs
Typical leader-based protocols
Current Designs
Typical leader-based protocols
Current Designs
Typical leader-based protocols
Narwhal
Dag-based mempool
The mempool is the key

Reaching consensus on metadata is cheap
Narwhal
The workers and the primary
Narwhal
The workers and the primary
Narwhal
The workers and the primary
Narwhal
The workers and the primary

Client transactions

Transactions

Worker 1

Digest

Worker 2

Batch

Digest

Primary

Digest

Worker n

Transactions

Transactions

Transactions

Digest

Batch

Transactions
Narwhal
The workers and the primary

Client transactions

Transactions

Worker 1

Worker 2

Worker n

Narwhal mempool

Digest

Batch

Transactions

Transactions

Transactions

' mempool protocol '
Narwhal
The primary machine
Narwhal
The primary machine
Narwhal
The primary machine
Narwhal
The primary machine

Round 1
Narwhal
The primary machine

Round 1
Byzantine 'Reliable' Broadcast
Narwhal
The primary machine
Modified Narwhal

Adapt Narwhal for partial-synchronous networks
Modified Narwhal

Even rounds: wait for the leader (or to timeout)
Modified Narwhal

Odd rounds: wait for enough votes (or to timeout)
Bullshark

Zero-message partially-synchronous consensus
Bullshark

Zero-message partially-synchronous consensus

* without asynchronous fallback
Bullshark
Just interpret the DAG

r1

r2
Bullshark
Deterministic leader every 2 rounds
Bullshark
The leader needs \( f+1 \) links from round \( r \)
Bullshark

The leader needs $f+1$ links from round $r$

One node supports $L1$!
**Bullshark**

The leader needs $f+1$ links from round $r$

Not enough support!
(Nothing is committed at this stage)
Bullshark

Elect the leader of r4
Bullshark
Leader L2 has enough support
Bullshark
Leader L2 has links to leader L1

First commit L1
Then commit L2
Bullshark

Commit all the sub-DAG of the leader
Bullshark
Commit all the sub-DAG of the leader
Implementation

- Written in Rust
- Networking: Tokio (TCP)
- Storage: RocksDB
- Cryptography: ed25519-dalek

https://github.com/asonnino/narwhal
Evaluation
Experimental setup on AWS

m5d.8xlarge
Evaluation

Throughput latency graph

Figure 2: Comparative throughput-latency performance of HotStuff, Tusk, and BullShark. WAN measurements with 10, 20, 50 parties. No faulty parties, 500KB maximum block size and 512B transaction size.

Figure 3: Maximum achievable throughput of HotStuff, Tusk, and BullShark, keeping the latency under 2.5s and 5s. WAN measurements with 10, 20, 50 parties. No faulty parties, 500KB maximum block size and 512B transaction size.

Tusk and BullShark maintain a good level of throughput: the underlying DAG continues collecting and disseminating transactions despite the crash-faults, and is not overly affected by the faulty parties. The reduction in throughput is in great part due to losing the capacity of faulty parties. When operating with 3 faults, both Tusk and BullShark provide a 10x throughput increase and about 7x latency reduction with respect to HotStuff.

9.3 Performance under asynchrony

HotStuff has no liveness guarantees when the eventual synchrony assumption does not hold (before GST), either due to (aggressive) DDoS attacks targeted against the leaders or adversarial delays on the leaders' messages as experimentally proven in prior work. That is, the throughput of the system falls to 0. The same can happen to the partially synchronous version of BullShark. The reason is that whenever a party becomes the leader for some round, its proposal can be delayed such that all other parties timeout for that round. In order to avoid this attack, Tusk and DAG-Rider elects leaders unpredictably after the DAG is constructed which makes such attacks impossible. The purpose of the fallback mode of BullShark is to maintain the same liveness properties as Tusk and DAG-Rider under asynchrony without compromising on performance during periods of synchrony. If the voting type of all parties is fallback, then BullShark acts as Tusk. In the fallback mode, BullShark thus renounces to its latency advantage with respect to Tusk in order to remain live under asynchrony. As any asynchronous protocol, the performance of both Tusk and BullShark during periods of asynchrony can be arbitrarily bad as they depend on the network conditions (which guarantee delivery after unbounded time).
Evaluation
Performance under faults

Figure 2: Comparative throughput-latency performance of HotStuff, Tusk, and BullShark. WAN measurements with 10, 20, 50 parties. No faulty parties, 500KB maximum block size and 512B transaction size.

Figure 3: Maximum achievable throughput of HotStuff, Tusk, and BullShark, keeping the latency under 2.5s and 5s. WAN measurements with 10, 20, 50 parties. No faulty parties, 500KB maximum block size and 512B transaction size.

Tusk and BullShark maintain a good level of throughput: the underlying DAG continues collecting and disseminating transactions despite the crash-faults, and is not overly affected by the faulty parties. The reduction in throughput is in great part due to losing the capacity of faulty parties. When operating with 3 faults, both Tusk and BullShark provide a 10x throughput increase and about 7x latency reduction with respect to HotStuff.

9.3 Performance under asynchrony
HotStuff has no liveness guarantees when the eventual synchrony assumption does not hold (before GST), either due to (aggressive) DDoS attacks targeted against the leaders [37] or adversarial delays on the leaders’ messages as experimentally proven in prior work [19, 22]. That is, the throughput of the system falls to 0. The same can happen to the partially synchronous version of BullShark.

Figure 4: Comparative throughput-latency under crash-faults of HotStuff, Tusk, and BullShark. WAN measurements with 10 parties. Zero, one, and three crash-faults, 500KB maximum block size and 512B transaction size.

The reason is that whenever a party becomes the leader for some round, its proposal can be delayed such that all other parties timeout for that round. In order to avoid this attack, Tusk and DAG-Rider elects leaders unpredictably after the DAG is constructed which makes such attacks impossible. The purpose of the fallback mode of BullShark is to maintain the same liveness properties as Tusk and DAG-Rider under asynchrony without compromising on performance during periods of synchrony. If the voting type of all parties is fallback, then BullShark acts as Tusk. In the fallback mode, BullShark thus renounces to its latency advantage with respect to Tusk in order to remain live under asynchrony. As any asynchronous protocol, the performance of both Tusk and BullShark during periods of asynchrony can be arbitrarily bad as they depend on the network conditions (which guarantee delivery after unbounded time). When
Conclusion

**Bullshark**

- Zero-message overhead, no view-change, no common-coin
- Disseminate data with Narwhal, exploits periods of synchrony

- **Paper:** https://sonnino.com/papers/bullshark.pdf
- **Code:** https://github.com/asonnino/narwhal