Bullshark

DAG BFT Protocols Made Practical

Alberto Sonnino
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
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

Client transactions

Narwhal mempool

Worker 1

Worker 2

Primary

Worker n
Narwhal
The workers and the primary

Client transactions

Narwhal mempool

Worker 1

Worker 2

Worker n

Primary
Narwhal
The workers and the primary
Narwhal
The workers and the primary

Client transactions → Transactions → Worker 1 → Batch

Transactions → Worker 2 → Primary → Batch

Transactions → Worker n → Batch
Narwhal
The workers and the primary

Narwhal mempool

Client transactions

Transactions

Worker 1

Digest

Batch

Work 2

Digest

Batch

Worker n

Digest

Primary

Batch

'mempool protocol'
Narwhal
The primary machine

block header

G1

G2

G3

H

H

H
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
HotStuff on Steroids

Just by replacing the mempool
HotStuff on Narwhal
Overview
HotStuff on Narwhal
Enhanced commit rule

C1
HotStuff on Narwhal
Enhanced commit rule
HotStuff on Narwhal
Enhanced commit rule

Faulty HotStuff Leader!
Blocks may still be ‘saved’
HotStuff on Narwhal
Enhanced commit rule
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
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

L1
r1  r2  r3  r4  r5
L2

L1

L2
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 [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).
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.

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
Summary

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
Engineering

Lessons Learned
Evaluation
Our biggest mistakes

ulant
Add crash-recovery after-the-fact
Add the synchroniser after-the-fact
Add epoch changes after-the-fact
Do not benchmark from day 1
Start with fancy crypto
Hide away serialisation
Complex networked systems

Isolate modules as in papers
(Use gRPC and magic network stack)
Evaluation
Our biggest mistakes

اخذ التعافي بعد الأمر على التوالي
اخذ التزامن على التوالي بعد الأمر
اخذ التغييرات على الجيل بعد الأمر على التوالي

• ما هو المعدات الأدنى التي تحتاج إلى نسخة بعد التعافي من خراب؟
• التزامن سيصبح مصدراً للمدخلات وصعوبة الصيانة في النهاية
• التغييرات على الجيل أكثر تعقيداً مما يبدو (تنزيل موروثات الجيل الجديد وتحديث موروثات من النواحي من الشبكة) — ننصح: قتل النود وتشغيله من جديد.
Evaluation
Our biggest mistakes

ulant
Add crash-recovery after-the-fact
Add the synchroniser after-the-fact
Add epoch changes after-the-fact
Do not benchmark from day 1

• Many concurrency bugs found on WAN benchmarks under high load
• Spent months optimising blinding
Evaluation

Our biggest mistakes

😔 Add crash-recovery after-the-fact
😔 Add the synchroniser after-the-fact
😔 Add epoch changes after-the-fact
😔 Do not benchmark from day 1
😔 Start with fancy crypto
😔 Hide away serialisation

• **Huge complexity; resulted redundant crypto operations**
• **Crypto serialisation was a bottleneck**
Evaluation
Our biggest mistakes

 schlecht  Add crash-recovery after-the-fact
 schlecht  Add the synchroniser after-the-fact
 schlecht  Add epoch changes after-the-fact
 schlecht  Do not benchmark from day 1
 schlecht  Start with fancy crypto
 schlecht  Hide away serialisation

 schlecht  Complex networked systems

- Harder crash-recovery / should start with collocated workers
Evaluation
Our biggest mistakes

ドラジニア
Add crash-recovery after-the-fact
ドラジニア
Add the synchroniser after-the-fact
ドラジニア
Add epoch changes after-the-fact
ドラジニア
Do not benchmark from day 1
ドラジニア
Start with fancy crypto
ドラジニア
Hide away serialisation
ドラジニア
Complex networked systems

• Debugging / perf improvement nightmare
alberto@mystenlabs.com

Alberto Sonnino
EXTRA

Benchmark of BFT Systems
Evaluation
Typical mistakes

😢 Forgo persistent storage
😢 Do not sanitise messages
😢 Local/LAN benchmark + ping
😢 Many nodes on same machine
😢 Change parameters across runs
😢 Set transaction size to zero
😢 Preconfigure nodes with txs

😢 Send a single burst of transactions
😢 Benchmark for a few seconds
😢 Start timer in the batch maker
😢 Evaluate latency w/ only the first tx
�� Separate latency and throughput
�� Only benchmark happy path
Evaluation
Set the benchmark parameters

Faults: 0 node(s)
Committee size: 10 node(s)
Transaction size: 512 B
**Evaluation**

Set the benchmark parameters

- **Faults:** 0 node(s)
- **Committee size:** 10 node(s)
- **Transaction size:** 512 B
- **Header size:** 1,000 B
- **Max header delay:** 200 ms
- **GC depth:** 50 round(s)
- **Sync retry delay:** 5,000 ms
- **Sync retry nodes:** 3 node(s)
- **batch size:** 500,000 B
- **Max batch delay:** 200 ms
Evaluation
Typical mistakes

- Forgo persistent storage
- Do not sanitise messages
- Local/LAN benchmark + ping
- Many nodes on same machine
- Change parameters across runs
- Set transaction size to zero
- Preconfigure nodes with txs

- Send a single burst of transactions
- Benchmark for a few seconds
- Start timer in the batch maker
- Evaluate latency w/ only the first tx
- Separate latency and throughput
- Only benchmark happy path
Evaluation
Benchmark clients

For a long time (minutes)

Fixed input rate
Evaluation

Typical mistakes

 الانترنت Forgo persistent storage

インターネット Do not sanitise messages

インターネット Local/LAN benchmark + ping

インターネット Many nodes on same machine

インターネット Change parameters across runs

インターネット Set transaction size to zero

😡 Preconfigure nodes with txs

😡 Send a single burst of transactions

😡 Benchmark for a few seconds

😡 Start timer in the batch maker

😡 Evaluate latency w/ only the first tx

😡 Separate latency and throughput

😡 Only benchmark happy path
Evaluation

Typical mistake

Propose batch 5 (pointer)

Load txs from pre-populated store & commit
Evaluation
Typical mistakes

 😫 Forgo persistent storage
 😫 Do not sanitise messages
 😫 Local/LAN benchmark + ping
 😫 Many nodes on same machine
 😫 Change parameters across runs
 😫 Set transaction size to zero
 😫 Preconfigure nodes with txs

 😫 Send a single burst of transactions
 😫 Benchmark for a few seconds
 😫 Start timer in the batch maker
 😫 Evaluate latency w/ only the first tx
 😫 Separate latency and throughput
 😫 Only benchmark happy path
Evaluation

Typical mistake

send 50k txs (once)

Benchmark client → Narwhal mempool → Tusk → output after 400 ms

 charisma

TPS = 50k / 400ms = 125k tx/s
Evaluation
Instrument the codebase

- Benchmark client
- Narwhal mempool
- Batch Maker
- Proposer
- Tusk

bench_start_time
sample_tx_id → send_time
**Evaluation**

**Instrument the codebase**

$batch\_digest \rightarrow sample\_tx\_id$

$batch\_digest \rightarrow batch\_bytes$

bench\_start\_time

$sample\_tx\_id \rightarrow send\_time$
Evaluation

Instrument the codebase

batch_digest -> sample_tx_id
batch_digest -> batch_bytes
block_digest -> batch_digest

bench_start_time
sample_tx_id -> send_time
Evaluation
Instrument the codebase

- batch_digest -> sample_tx_id
- batch_digest -> batch_bytes
- block_digest -> batch_digest
- bench_start_time
- sample_tx_id -> send_time
- block_digest -> commit_time

Diagram:
- Benchmark client
- Batch Maker
- Proposer
- Tusk
- Narwhal mempool
- Ordered transactions
**Evaluation**

**Compute throughput**

\[ \text{total\_time} = \text{last\_commit\_time} - \text{bench\_start\_time} \]

\[ \text{BPS} = \frac{\text{total\_bytes}}{\text{total\_time}} \]

\[ \text{TPS} = \frac{\text{BPS}}{\text{transaction\_size}} \]
Evaluation
Compute latency

bench_start_time
sample_tx_id -> send_time

batch_digest -> sample_tx_id
batch_digest -> batch_bytes

block_digest -> batch_digest
block_digest -> commit_time

samples = commit_time - send_time
latency = average(samples)
Evaluation

Typical mistakes

?): Forgo persistent storage

?): Do not sanitise messages

?): Local/LAN benchmark + ping

?): Many nodes on same machine

?): Change parameters across runs

?): Set transaction size to zero

?): Preconfigure nodes with txs

?): Send a single burst of transactions

?): Benchmark for a few seconds

?): Start timer in the batch maker

?): Evaluate latency w/ only the first tx

?): Separate latency and throughput

?): Only benchmark happy path
Evaluation

Throughput latency graph
Evaluation
Throughput latency graph

Change only input rate
Evaluation
Throughput latency graph
Evaluation

Throughput latency graph
Evaluation
Throughput latency graph

Longer benchmarks
Evaluation

Throughput latency graph

Breaking point!
Evaluation
Typical mistakes

🧫 Forgo persistent storage
🧫 Do not sanitise messages
🧫 Local/LAN benchmark + ping
🧫 Many nodes on same machine
🧫 Change parameters across runs
🧫 Set transaction size to zero
🧫 Preconfigure nodes with txs

🧫 Send a single burst of transactions
🧫 Benchmark for a few seconds
🧫 Start timer in the batch maker
🧫 Evaluate latency w/ only the first tx
🧫 Separate latency and throughput
🧫 Only benchmark happy path
Evaluation
Throughput latency graph
Evaluation
Scalability
Evaluation
Scalability

![Graph showing latency vs. throughput for different systems and worker counts.](image-url)
Evaluation
Scalability
Evaluation
Performance under faults
Evaluation
Typical mistakes

-insert text here-

- Forgo persistent storage
- Do not sanitise messages
- Local/LAN benchmark + ping
- Many nodes on same machine
- Change parameters across runs
- Set transaction size to zero
- Preconfigure nodes with txs

- Send a single burst of transactions
- Benchmark for a few seconds
- Start timer in the batch maker
- Evaluate latency w/ only the first tx
- Separate latency and throughput
- Only benchmark happy path
Evaluation
Still many caveats

- Perfect load balance
- Transaction deduplication
- Synthetic load
- No Byzantine adversary
- No network adversary
- Only AWS network