Narwhal-Bullshark
DAG BFT Protocols Made Practical

Alberto Sonnino
Byzantine Fault Tolerance

> $\frac{2}{3}$
Blockchains

1. make transaction
Blockchains

1. make transaction

2. submit transaction
Blockchains

1. make transaction
2. submit transaction
3. sequence and verify
Blockchains

1. make transaction

2. submit transaction

3. sequence and verify

4. store
The best example

1. Send 5 coins to Bob

2. Send 5 coins to Bob

3. Payment authorised?

4. Store
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

Transactions

Worker 1

Worker 2

Worker n

Narwhal mempool

Primary
Narwhal
The workers and the primary

Client transactions

Narwhal mempool

Worker 1

Worker 2

Worker n

Transactions

Transactions

Transactions

Batch

Batch

Batch
Narwhal
The workers and the primary
The workers and the primary

Narwhal mempool

Worker 1 → Batch
Digest

Worker 2 → Batch
Digest

Primary

Worker n → Batch
Digest

Transactions

Client transactions

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

The primary machine

block header

G1

G2

G3

certificate

H V C

H V C

H V C

C C
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

- Client transactions
- Narwhal mempool
- Certificates
- Garbage collection
- Partially Synchronous Consensus (HotStuff)
- Ordered transactions
- State machine replication execution
HotStuff on Narwhal
Enhanced commit rule
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
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)
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

\[ r_1 \quad r_2 \]

L1
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 [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).

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 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).
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

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

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

 hashCode

 Add crash-recovery after-the-fact
>Add the synchroniser after-the-fact
 Add epoch changes after-the-fact

• What is the minimum state you need to persist across crash-recovery?
• The synchroniser will eventually be your bottleneck / source of instability
• Epoch changes are more complex than they look (sync new validators/update configs from chain) — Advise: kill the node and reboot it.
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

- Many concurrency bugs found on WAN benchmarks under high load
- Spent months optimising blinding
Evaluation
Our biggest mistakes

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

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

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

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

 sırasında: ❌ 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

- Debugging / perf improvement nightmare

### Isolate modules as in papers
### (Use grpc and magic network stack)
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
� Separating 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

Fixed input rate

For a long time (minutes)
Evaluation

Typical mistakes

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

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

Typical mistake

Narwhal mempool

Tusk

Propose batch 5 (pointer)

Load txs from pre-populated store & commit

Narwhal mempool

Tusk

Load txs from pre-populated store & commit

Narwhal mempool

Tusk

Load txs from pre-populated store & commit

Narwhal mempool

Tusk

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

 расчетальных ошибках

- Don't forget 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

Ordered transactions

output after 400 ms

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

bench_start_time
sample_tx_id -> send_time
Evaluation
Instrument the codebase

batch_diggest \rightarrow sample\_tx\_id
batch_diggest \rightarrow batch\_bytes

bench_start_time
sample\_tx\_id \rightarrow send\_time
Evaluation
Instrument the codebase

batch_diggest -> sample_tx_id
batch_diggest -> batch_bytes
block_diggest -> batch_diggest

bench_start_time
sample_tx_id -> send_time
Evaluation
Instrument the codebase

batch_digest $\rightarrow$ sample_tx_id
batch_digest $\rightarrow$ batch_bytes
block_digest $\rightarrow$ batch_digest

bench_start_time
sample_tx_id $\rightarrow$ send_time

batch_digest $\rightarrow$ commit_time
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

\[
samples = \text{commit\_time} - \text{send\_time}
\]

\[
\text{latency} = \text{average}(\text{samples})
\]
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
Evaluation
Scalability
Evaluation
Performance under faults
Evaluation
Typical mistakes

יש לו להפרדה בין המאבקות וה丐ות בירוחב כולל את התיקונים עצמאיים בין ה_runsים והאם להגדיר את תamanho התראה שם בירוחב האפס

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