Echidna: A New Consensus Algorithm for Efficient State Machine Replication

1st Rui Pedro Bernardo Morais
Instituto de Telecomunicações,
University of Beira Interior
Covilhã, Portugal
ru.morais@ubi.pt

2nd Paul Andrew Crocker
Instituto de Telecomunicações,
University of Beira Interior
Covilhã, Portugal
https://orcid.org/0000-0001-6824-6136

3rd Simão Melo de Sousa
Nova Lincs, Release Laboratory
University of Beira Interior
Covilhã, Portugal
https://orcid.org/0000-0001-9129-4136

***Abstract*—This paper introduces a novel algorithm, called Echidna, which solves multi-valued consensus, as well as two different approaches built upon it to achieve efficient State Machine Replication (SMR) with very low latency, Sphinx and Cerebrus, making them specially suited for decentralized payment systems.**

Sphinx has a leader-based design with a single proposer, while Cerebrus employs a leaderless strategy with multiple proposers. Both designs were implemented and their performance was evaluated in comparison with state-of-the-art decentralized solutions. Results showed that Sphinx and Cerebrus not only deliver comparable performance to them in terms of latency, but also present significant robustness under faults, while achieving enough scalability to handle typical usage levels of centralized payment systems.

***Index Terms*—consensus, state machine replication, decentralized payment systems**

I. Introduction

State Machine Replication (SMR) has long been a fundamental component in distributed systems [1]. Traditionally, it was used as a way to guarantee redundancy and, consequently, reliability in a centralized system. However, due to the emergence of cryptocurrencies and blockchain technology, recent research has focused mainly on SMR applied to decentralized systems, where consensus plays a critical role.

One of the most remarkable works in this area is the pivotal FLP (Fischer, Lynch, and Paterson) impossibility theorem [2], asserting that no consensus algorithm can simultaneously be asynchronous, deterministic and terminate. Attempts to skirt around this theorem have culminated in the emergence of a variety of consensus algorithm typologies, including deterministic partially synchronous, randomized asynchronous, leader-based, and leaderless algorithms.

Notwithstanding these theoretical considerations, the most consequential factors in real-world settings pivot on practical utility and performance metrics. Primarily, these encompass scalability, latency and resilience, i.e., how they are impacted by faults and Byzantine behavior. Invariably, these elements exist in a state of delicate trade-off, as enhancements to one may inadvertently undermine another, requiring strategic design choices.

This work is funded by FCT/MCTES through national funds and when applicable co-funded EU funds under the project UIDB/50008/2020 and also by the University of Beira Interior.

This paper delves into the creation of a consensus algorithm uniquely adapted to achieve efficient SMR. Our design prioritizes low latency and robust resilience, followed by scalability, making it specially suited for decentralized payment systems. We posit that to genuinely rival established centralized payment systems, a decentralized payment system must match, if not surpass, their performance. Thus, our focus is on fostering an algorithm that delivers this very capability.

A. Related Work

This section provides an overview of some remarkable research related to SMR and consensus protocols.

Classical consensus protocols, such as Paxos [3] and Raft [4], have been subjected to rigorous study given their critical role in distributed computing, but both only provide a solution to non-Byzantine faults within a network.

The PBFT (Practical Byzantine Fault Tolerance) protocol [5], with its pioneering approach, has fueled extensive investigations in the domain of BFT, though it falls short in terms of scalability.

Subsequent protocols improve on it, such as HotStuff [6] and SBFT [7], by offering improvements over PBFT's message complexity by requiring only a linear number of messages in the common case, while Tendermint [8] improved on its view-change complexity with a new termination mechanism.

Other categories of BFT protocols were also developed, such as optimistic BFT protocols, like Zyzzyva [9], with better performance in the good case; and probabilistic BFT protocols like Honeybadger [10] and BEAT [11], that achieve both safety and liveness in purely asynchronous networks.

In cryptocurrencies, the Nakamoto's consensus protocol [12], adopted widely in blockchain systems, offers significant robustness against Sybil attacks, but suffers from high latency, low throughput, and substantial computational resource usage, stirring environmental concerns.

Stellar [13] and Algorand [14] try to solve those issues by employing a Byzantine agreement protocol with asymmetric quorums and using a cryptographic sortition mechanism for committee selection, respectively.

Other approaches use a multi proposers design instead of a single one, like MirBFT [15] and Red Belly [16], but the former still has a single point of failure in the leader, called

"primary", and the latter is based on a binary consensus algorithm, which needs to be instantiated multiple times, while our design is built on a single instance of multi-valued consensus.

Narwhal [17], Bullshark [18] and Nero [19] have very good performance, but they are specifically designed for a Directed Acyclic Graph (DAG) and they have a more complex implementation than traditional blockchains.

B. Contributions

This paper makes the following contributions:

- A new algorithm called Echidna that solves the Multi-valued Byzantine Consensus with a new simple fast mechanism to handle conflicting values.
- A new algorithm called Sphinx that solves the State Machine Replication problem with a leader-based approach by building upon Echidna. Due to its mechanism to handle conflicting values, Sphinx can have lower time-outs than other leader-based consensus algorithms, since conflicting proposals of different leaders are resolved by Echidna. Consequently, the latency of the consensus is lower.
- A new algorithm called Cerebrus that solves the State Machine Replication problem with a leaderless approach that uses Echidna as a building block. Cerebrus avoids the leader bottleneck, since any process can propose a set of values, but does not have the shortcomings of leaderless algorithms, such as low scalabilty, since the decided value is a set of at least 2f + 1 proposals instead of a single proposal.
- We implement Sphinx and Cerebrus in Rust in an open source code, and benchmark it against the state of the art consensus algorithm Bullshark in the common case, under crash-faults and with conflicting values, showing that they have comparable latency and resilience and that they also have good scalability.

The paper is divided in the following way: Section 2 starts by defining formally our goal and the building blocks that we use to achieve that goal under the rules of the system model. Section 3 presents the algorithm Echidna with the corresponding formal proofs of correctness. Section 4 showcase the algorithms Spinx and Cerebrus, also with formal proofs of correctness and a description of how they can be applied to decentralized payment systems. Section 5 is dedicated to the implementation and evaluation of both algorithms and the last section of the paper concludes it.

II. Preliminaries

A. Goal

Our goal is to achieve State Machine Replication (SMR) [1], which is a commonly used method for creating copies of services modeled as deterministic state machines. The fundamental principle of this method is to ensure that all processes initiate from an identical state and process requests from clients in the same sequence. This way, it is assured that the states of the processes stay consistent and never deviate.

This problem is composed of the following two properties:

- Agreement: All correct processes should receive all requests.
- Order: The sequence of received requests remains consistent across all processes.

B. Building Blocks

1) Multi-valued Byzantine Consensus: We use Multi-valued Byzantine consensus as a building block to solve the SMR problem, where the goal is to achieve consensus on a value v from a set that has arbitrary but finite size, i.e., $v \in V$ and $|V| > 2$. The problem can be formally defined in terms of three properties:

- Agreement: Every correct process decides the same value.
- Termination: Every correct process eventually decides.
- Validity: If a correct process decides v, then v was proposed by some process.

C. System Model

Our system is composed of processes that follow the stated algorithms, designated by "correct", and those that can follow any other algorithm or not respond, designated by "byzantine".

Processes communicate by exchanging signed messages that ensure the authenticity of the sender and the message. We model the network communication using a variant of the partially synchronous system model [20] with the gossip property [8].

This means that beyond an uncertain point in time, called the Global Stabilization Time (GST), all messages sent at time t by correct processes are guaranteed to be delivered before $t + \Delta$ to all the other correct processes.

The bound Δ and when GST starts are system parameters whose values are not required to be known for the correctness of our algorithms, however they ensure that they terminate.

III. Echidna: Multi-Valued Byzantine Consensus

An instance of the consensus protocol, see Algorithm 1, starts with each correct process broadcasting a vote on a specified value v, which is acquired externally from the consensus instance.

Subsequently, a timer for that round is initiated, and the process waits for at least 2f + 1 and the timer's expiration to generate a new message in the subsequent round.

If the process receives at least of 2f + 1 commits on value v, it decides that value (line 27).

In a situation where the process receives at least 2f + 1 votes on value v in round r, and no value has been committed yet, it commits that value with proof round r and broadcasts a COMMIT message containing that information (line 15).

Otherwise, if there is no committed value, it votes for the value that has received the most messages (both votes and commits). If there is a tie, it votes for the value with the highest hash (line 20).

Given the possibility that distinct processes may commit different values and consensus may not terminate, it becomes

necessary to safely alter a committed value if required. There are two scenarios that may trigger this change:

- The process receives at least 2f + 1 messages with a different value v' in a round r' that is more recent than the proof round pr of its committed value v (line 15).
- The process receives at least of f + 1 COMMIT messages with a different value v' in a round r' that is more recent than the proof round pr of its committed value v (line 32).

Line 36 and the timers between rounds guarantee that the algorithm terminates, since GST will eventually be reached and all correct processes will receive the same messages and, consequently, produce the same response messages. Since the actual delay of the network is unknown, the timer can be increased in each round.

A. Correctness Proofs

We now prove formally that Echidna solves the Multi-valued Consensus problem.

lemma 1: For all $f \geq 0$, any pair of process sets that have a voting power of 2f + 1 or more, there must be at least one correct process shared between them.

Proof: Given that the sum of all voting power is represented by n, where n is equal to 3f + 1, we can see that double the voting power, or 2(2f + 1), is equivalent to n + f + 1. What this implies is that if we consider the intersection between two process sets that each have voting power equal to 2f + 1, they must have an overlap of at least f + 1 voting power. This overlap will contain at least one correct process, because the total voting power that can be contributed by faulty processes is only f. This direct reasoning leads us to our stated result. ■

lemma 2: Echidna satisfies Agreement.

Proof: Let r be the first round such that some correct process p decides v. We now prove that if some correct process q decides v' in some round $r' \geq r$, then $v = v'$.

In case $r = r'$, q has received at least $2f + 1$ commits of v v' in r r', while p has received at least $2f + 1$ commits of v v in r r.

By Lemma 1, two sets of messages of voting power $2f + 1$ intersect in at least one correct process. As a correct process sends only one message per r, then $v = v'$.

In case $r' > r$, by the decision rules, p has received at least $2f+1$ voting power equivalent of commits of value v in round $r - 1$ with proof round pr.

On the other side, q has received at least $2f + 1$ voting power equivalent of commits of value v' in round $r' - 1$ with proof round pr'. By Lemma 1, that means that at least one correct process c changed its committed value from v to v'.

According to the algorithm, a correct process only changes its committed value iff it receives at least $f + 1$ commits of a different value with a more recent proof round than pr (line 32) or at least $2f + 1$ messages of a different value in a more recent round than pr (line 15).

This would only be possible if some correct process that committed value v other than c changed its vote, but that

Algorithm 1: Echidna

1 **Initialization:**
2 $instance_p := 0$
3 $round_p := 0$
4 $committedValue_p := nil$
5 $decision_p[] := nil$
6 $voteTimer := nil$
7 $value_p := nil$

8 **Function** $Start(instance_p, value)$:
9 **if** $round_p = 0$
10 $value_p \leftarrow value$
11 **broadcast** $\langle$VOTE, $instance_p, round_p, v\rangle$
12 $voteTimer \leftarrow running$
13 **schedule** $OnTimeoutVote(instance_p, round_p)$ to be executed **after** $timeoutVote(round_p)$
14 $round_p \leftarrow round_p + 1$

15 **upon** $2f + 1$ $\langle *, instance_p, r, *, v\rangle$ **with** $r > round_p$ **AND** ($r > proofRound_p$ **OR** $committedValue_p = nil$) **do**
16 $round_p \leftarrow r$
17 $committedValue_p \leftarrow v$
18 $proofRound_p \leftarrow r$
19 **broadcast** $\langle$COMMIT, $instance_p, round_p, v\rangle$

20 **upon** $2f + 1$ $\langle *, instance_p, round_p, *, *\rangle$ **AND** $voteTimer = expired$ **do**
21 $round_p \leftarrow round_p + 1$
22 **if** $committedValue_p \neq nil$ **then**
23 **broadcast** $\langle$COMMIT, $instance_p, round_p, proofRound_p, committedValue_p\rangle$
24 **else**
25 $value_p \leftarrow$ max($value$.count()) (if tie, the $value$ with the highest id)
26 **broadcast** $\langle$VOTE, $instance_p, round_p, nil, value_p\rangle$

27 **upon** $2f + 1$ $\langle$COMMIT, $instance_p, r, v\rangle$ **while** $decision_p[instance_p] = nil$ **do**
28 **if** $valid(v)$ **then**
29 $decision_p[instance_p] = v$
30 $instance_p \leftarrow instance_p + 1$
31 $StartRound(0)$

32 **upon** $f + 1$ $\langle$COMMIT, $instance_p, r, pr, v\rangle$ **with** $r > round_p$ **AND** ($pr > proofRound_p$ **OR** $pr = -1$) **do**
33 $round_p \leftarrow r$
34 $committedValue_p \leftarrow v$
35 $proodRound_p \leftarrow r$

36 **upon** $f + 1$ $\langle *, instance_p, r, *, *v\rangle$ **AND** $r \geq proofRound_p > -1$ **AND** $committedValue_p \neq *v$ **do**
37 **rebroadcast** proof of $committedValue_p$ from round $proofRound_p$

38 **Function** *OnTimeoutVote*($height, round$) :
39 $voteTimer \leftarrow expired$

process would need the same conditions to change, which is a contradiction. ■

lemma 3: Echidna satisfies Termination.

Proof:

There are two possible cases:

1) No correct process has committed a value v before GST is reached. When GST is reached all correct processes will vote, commit and decide on the value with the highest count.
2) Some correct processes p have committed value v' before GST in round r with proof round pr. There are two possible cases:
   a) $p \leq f$. There are at least $f+1$ correct processes q that can vote or commit value v, with v.count() $>$ v'.count() (if v'.count() $>$ v.count() they can never commit v because at least $f+1$ will never change from value v' to v).
   If q commit, p will eventually receive the proof of that commit and switch their commits from v' to v because the round r is more recent than proof round pr (line 32). Eventually all correct processes decide v.
   If q do not commit, they will eventually receive the messages from round pr, because p will re-broadcast it (line 37), and commit v' because $commited_q = nil$ (line 23). Eventually all correct processes decide v'.
   b) $p > f$. All correct processes will eventually commit v' because they cannot commit other value because p do not change their commits and either the rule of line 15 or the rule of line 32 will eventually be triggered. Eventually all correct processes decide v'.

■

lemma 4: Algorithm 1 satisfies Validity.

Proof: Trivially follows from the rule at line 16 which ensures that only valid vs can be decided. ■

Theorem 1: Echidna satisfies the Validity Predicate-based Byzantine Consensus.

Proof: Trivially follows from Lemma 2, Lemma 3 and Lemma 4. ■

IV. State Machine Replication

In this section we use Echidna consensus algorithm as a building block to achieve State Machine Replication. We do that using the following two different approaches:

- Sphinx: leader-based SMR, where there is only a single proposer in each round. The decided value is the value returned by the corresponding Echidna consensus instance and contains a single proposal.
- Cerebrus: leaderless SMR, where any process can be a proposer. The decided value is the value returned by the corresponding Echidna consensus instance and contains at least 2f + 1 proposals.

A. Sphinx: Leader-Based State Machine Replication

In every round of the Sphinx algorithm, a leader is chosen who has the responsibility of suggesting a value. This suggested value is then transferred to the associated Echidna consensus instance. If the leader fails to suggest a value before the deadline, a new round begins, electing a new leader until a value is suggested.

Depending on the network conditions, it is feasible that proposed values may conflict; however, the Echidna consensus will resolve this conflict. When the related Echidna consensus instance delivers a value, the Sphinx consensus instance concludes, making this value the final decision. Following this, a fresh instance commences.

Algorithm 2: Sphinx: Leader-Based State Machine Replication

```
1  Initialization:
2      instance_p := 0
3      decisions_p[] := nil
4      proposeTimer := nil
5      proposal_p := nil
6  upon start do Start(0)
7  Function Start(round) :
8      round_p ← round
9      if proposer(instance_p, round_p) = p then
10         proposal ← getValue()
11         broadcast ⟨PROPOSAL, instance_p, proposal⟩
12     else
13         schedule OnTimeoutPropose(instance_p) to
       be executed after timeoutPropose()
14 upon ⟨PROPOSAL, instance_p, proposal⟩ for the first
     time do
15     proposal_p ←
   Echidna.StartRound(instance_p, proposal)
16     decisions_p[instance_p] ← proposal_p
17     instance_p ← instance_p + 1
18     reset initial values and empty message log
19     StartRound(0)
20 Function OnTimeoutPropose(height) :
21     StartRound(round_p + 1)
```

1) Correctness Proofs: We now prove formally that Sphinx solves the State Machine Replication problem.

lemma 5: Sphinx satisfies Agreement.

Proof: Since Echidna satisfies Agreement and Termination, all correct processes will receive the same decided values. ■

lemma 6: Sphinx satisfies Order.

Proof: Echidna consensus instances are executed sequentially (line 17), so all correct processes will receive the same decided values in the same order. ■

Theorem 2: Sphinx satisfies State Machine Replication.

Proof: Trivially follows from Lemma 5 and Lemma 6. ■

B. Cerebrus: Leaderless State Machine Replication

Contrary to Sphinx, in Cerebrus every correct process can propose a value. Once a correct process has received at least 2f + 1 proposals and the timer has expired, it passes that set of proposals as a value to the corresponding Echidna consensus instance, which will resolve conflicts between different sets of proposals and return only one. This will be the decided value of the consensus instance and a new one will start.

Algorithm 3: Cerebrus: Leaderless State Machine Replication

1 **Initialization:**
2 $instance_p := 0$
3 $decisions_p[] := nil$
4 $proposals_p[] := nil$
5 $proposeTimer := nil$
6 $value_p := nil$
7 **upon** start **do** *Start*()
8 **Function** *Start()* :
9 $v \leftarrow getValue()$
10 **broadcast** $\langle$PROPOSAL, $instance_p, v\rangle$
11 $proposeTimer \leftarrow running$
12 **schedule** $OnTimeoutPropose(instance_p)$ to be executed **after** $timeoutPropose()$
13 **upon** $\langle$PROPOSAL, $instance_p, proposal_q\rangle$ **do**
14 **add** $proposal_q$ to $proposals_p[]$
15 **upon** $2f + 1$ $\langle$PROPOSAL, $instance_p, *\rangle$ **AND** $proposeTimer = expired$ for the first time **do**
16 $value_p \leftarrow proposals_p[]$
17 $value_p \leftarrow$ Echidna.$StartRound(instance_p, value_p)$
18 $decisions_p[instance_p] \leftarrow value_p$
19 $instance_p \leftarrow instance_p + 1$
20 **reset** initial values and **empty** message log
21 $StartRound()$
22 **Function** *OnTimeoutPropose*($height$) :
23 $proposeTimer \leftarrow expired$

1) Correctness Proofs: We now prove formally that Cerebrus solves the State Machine Replication problem.

lemma 7: Cerebrus satisfies Agreement.

Proof: Since Echidna satisfies Agreement and Termination, all correct processes will receive the same decided values. ■

lemma 8: Cerebrus satisfies Order.

Proof: Echidna consensus instances are executed sequentially (line 19), so all correct processes will receive the same decided values in the same order. ■

Theorem 3: Cerebrus satisfies State Machine Replication.

Proof: Trivially follows from Lemma 7 and Lemma 8. ■

C. Application to Decentralized Payment Systems

Both Cerebrus and Sphinx can be employed to achieve consensus in a decentralized payment system. Depending on the specific implementation, the values passed on to the underlying Echina consensus could be valid transactions or ids (hashes) of valid transactions.

The consensus definitions used can be adapted to this use case, in the following way:

- Sphinx satisfies Validity Predicate-based Byzantine Consensus [21] if the decided value satisfies a given predicate valid(), which in this context would be a proposal containing only valid transactions.
- Cerebrus satisfies Set Byzantine Consensus [16] if the decided value is a set of transactions and this set is a valid non-conflicting subset of the union of the decided set of proposals.

In the case of Cerebrus it is still necessary to reconcile the decided set of proposals, removing conflicting transactions, to prevent double-spends.

To ensure a fair process, the initial proposal in the reconciliation process must be randomized, or at the very least, sequenced in a round-robin manner, as suggested in [16].

V. Implementation and Evaluation

For the purpose of our study, we created two separate branches from a forked version of the Narwhal project [1] in Rust, one for Sphinx and the other for Cerebrus, and we also open-sourced the implementations [2].

Narwhal proved to be an ideal foundation for our work, given its comprehensive benchmarking scripts for performance measurement under a variety of conditions. It is close to a production system, incorporating networking, cryptography, and persistent storage.

We implement three different benchmarking tests with the same number of processes and transaction load but under different conditions: in the common case, under crash-faults and with conflicting transactions. We start by submitting a fixed rate transaction load of 5000 and double up until 80000 transactions for a duration of 1 minute each, with every transaction having 532 bytes and we define 200 ms for the timeout between different rounds. Each process is represented by a e2-medium instance in Google Cloud Platform located in Europe.

In the comparative evaluation, Sphinx and Cerebrus are evaluated against the Bullshark branch, the highest performing algorithm on the repository.

A. Benchmark in the common case

When operating under saturation, both Cerebrus and Sphinx display similar performance characteristics due to the former having a dynamic unlimited block size. This implies that Sphinx, even with its single proposer, can accommodate the same volume of transactions as all the proposers of Cerebrus.

[1] https://github.com/facebookresearch/narwhal
[2] https://github.com/Fiono11/narwhal

As illustrated in Figure 1, Cerebrus exhibits superior scalability compared to Sphinx because it distributes the transaction load across all processes rather than consolidating it on a singular leader. Even though Sphinx and Cerebrus do not match Bullshark in terms of scalability, their performance is sufficient for real-world applications such as decentralized payment systems.

It is also worth noting that the saturation point is reached more quickly when there are fewer processes. This is an inherent design feature of the benchmarking tests employed, which distribute the same transaction load across all processes. Thus, with fewer processes, the individual transaction load per process increases, resulting in earlier saturation.

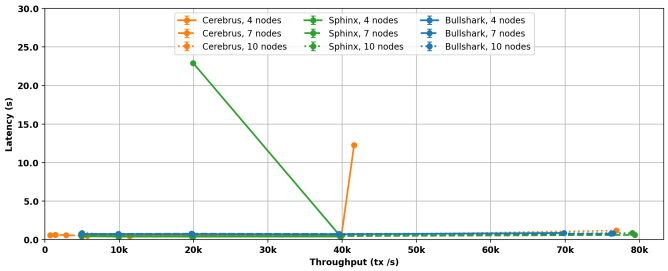

Fig. 1. Benchmark in the common case

B. Benchmark under crash-faults

As can be seen in Figure 2, Cerebrus demonstrates higher resilience compared to Sphinx when handling crash-faults, primarily because it does not rely on a single leader, thereby eliminating a single point of failure. Furthermore, under conditions of saturation, Cerebrus's performance remains largely unaffected, showing a level of robustness comparable to the state-of-the-art Bullshark consensus algorithm. This underlines Cerebrus's potential as a viable, resilient solution for decentralized systems.

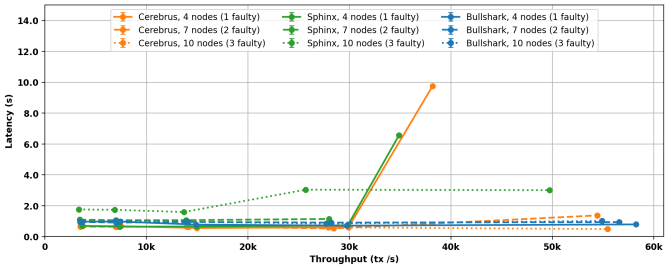

Fig. 2. Benchmark under crash-faults

C. Benchmark with conflicting values

Figure 3 reveals that the scalability of Sphinx and Cerebrus remains on a par when dealing with conflicting transactions. This is attributed to all the proposers in Cerebrus introducing conflicting transactions, thereby negating the advantage of multiple proposers over a single one. However, Sphinx exhibits lower latency, given that all its processes are correct and there is no proposal timeout during the proposal phase, unlike Cerebrus.

Despite demonstrating lower scalability in this scenario, both Sphinx and Cerebrus outperform Bullshark in terms of latency.

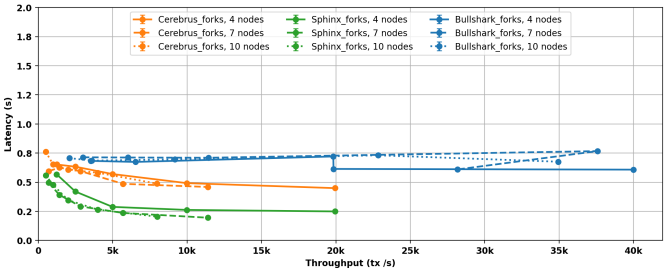

Fig. 3. Benchmark with conflicting values

VI. Conclusion

In conclusion, this paper presented Echidna, a groundbreaking algorithm for multi-valued consensus, and two distinct strategies based on it, Sphinx and Cerebrus, designed to deliver efficient SMR with extremely low latency. These algorithms show exceptional promise for their use in decentralized payment systems. Sphinx, with its leader-based single proposer design, and Cerebrus, with a leaderless multi-proposer approach, have been implemented and evaluated against current decentralized solutions.

The results indicate that both Sphinx and Cerebrus not only match state of the art in terms of latency but also display substantial resilience in the face of faults. Additionally, they possess the necessary scalability to handle the usage levels typically seen in centralized payment systems. This study underscores the potential of these innovative algorithms in enhancing the efficiency and robustness of decentralized payment systems, laying a solid foundation for future research and development in this field.

Acknowledgments

The authors would like to thank the support of the RELEASE, RELiablE And SEcure Computation Research Group of the University of Beira Interior and the Networks Architectures and Protocols group of the Instituto de Telecomunicações.

References

[1] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Comput. Surv.*, vol. 22, no. 4, p. 299–319, dec 1990. [Online]. Available: https://doi.org/10.1145/98163.98167

[2] M. Fischer, N. Lynch, and M. Paterson, "Impossibility of distributed consensus with one faulty process," *J. ACM*, vol. 32, pp. 374–382, 04 1985.

[3] L. Lamport, "Paxos made simple," *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pp. 51–58, December 2001. [Online]. Available: https://www.microsoft.com/en-us/research/publication/paxos-made-simple/

[4] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIX ATC'14. USA: USENIX Association, 2014, p. 305–320.

[5] M. Castro, B. Liskov *et al.*, "Practical byzantine fault tolerance," in *OsDI*, vol. 99, no. 1999, 1999, pp. 173–186.

[6] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham, "Hotstuff: Bft consensus with linearity and responsiveness," in *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, ser. PODC '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 347–356. [Online]. Available: https://doi.org/10.1145/3293611.3331591

[7] G. Golan Gueta, I. Abraham, S. Grossman, D. Malkhi, B. Pinkas, M. Reiter, D.-A. Seredinschi, O. Tamir, and A. Tomescu, "Sbft: A scalable and decentralized trust infrastructure," in *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2019, pp. 568–580.

[8] E. Buchman, J. Kwon, and Z. Milosevic, "The latest gossip on bft consensus," 07 2018.

[9] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: Speculative byzantine fault tolerance," vol. 51, 01 2007, pp. 45–58.

[10] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song, "The honey badger of bft protocols," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 31–42. [Online]. Available: https://doi.org/10.1145/2976749.2978399

[11] S. Duan, M. K. Reiter, and H. Zhang, "Beat: Asynchronous bft made practical," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 2028–2041. [Online]. Available: https://doi.org/10.1145/3243734.3243812

[12] L. Ren, "Analysis of nakamoto consensus." *IACR Cryptol. ePrint Arch.*, vol. 2019, p. 943, 2019.

[13] D. Mazières, "The stellar consensus protocol : A federated model for internet-level consensus," 2015.

[14] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich, "Algorand: Scaling byzantine agreements for cryptocurrencies," in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 51–68. [Online]. Available: https://doi.org/10.1145/3132747.3132757

[15] C. Stathakopoulou, T. David, M. Pavlovic, and M. Vukolić, "Mir-bft: High-throughput robust bft for decentralized networks," 2019. [Online]. Available: https://arxiv.org/abs/1906.05552

[16] T. Crain, C. Natoli, and V. Gramoli, "Red belly: A secure, fair and scalable open blockchain," in *2021 IEEE Symposium on Security and Privacy (SP)*, 2021, pp. 466–483.

[17] G. Danezis, L. Kokoris-Kogias, A. Sonnino, and A. Spiegelman, "Narwhal and tusk: A dag-based mempool and efficient bft consensus," in *Proceedings of the Seventeenth European Conference on Computer Systems*, ser. EuroSys '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 34–50. [Online]. Available: https://doi.org/10.1145/3492321.3519594

[18] A. Spiegelman, N. Giridharan, A. Sonnino, and L. Kokoris-Kogias, "Bullshark: Dag bft protocols made practical," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 2705–2718. [Online]. Available: https://doi.org/10.1145/3548606.3559361

[19] R. Morais, P. Crocker, and V. Leithardt, "Nero: A deterministic leaderless consensus algorithm for dag-based cryptocurrencies," *Algorithms*, vol. 16, no. 1, 2023. [Online]. Available: https://www.mdpi.com/1999-4893/16/1/38

[20] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *J. ACM*, vol. 35, no. 2, p. 288–323, apr 1988. [Online]. Available: https://doi.org/10.1145/42282.42283

[21] A. Spiegelman, N. Giridharan, A. Sonnino, and L. Kokoris-Kogias, "Bullshark: Dag bft protocols made practical," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 2705–2718. [Online]. Available: https://doi.org/10.1145/3548606.3559361