2019 Crypto Valley Conference on Blockchain Technology (CVCBT)

Short Paper: Albatross

An optimistic consensus algorithm

Pascal Berrang, Philipp von Styp-Rekowsky, Marvin Wissfeld
Nimiq Foundation
research@nimiq.com

Bruno França, Reto Trinkler
Trinkler Software
company@trinkler.software

***Abstract*—Distributed ledgers have the potential to revolutionize the financial landscape by removing trusted third parties. At the heart of most distributed ledgers is their consensus protocol. The consensus protocol describes the way participants in a distributed network interact with each other to obtain and agree on a shared state. While classical Byzantine fault tolerant (BFT) algorithms are designed to work in closed, size-limited networks only, modern distributed ledgers – and blockchains in particular – often focus on open, permissionless networks.**

In this paper, we present a novel blockchain consensus algorithm for open, permissionless networks, called *Albatross*, inspired by speculative BFT algorithms. We argue that the protocol is secure under regular PBFT security assumptions and has a theoretical performance close to the maximum for single-chain Proof-of-Stake consensus algorithms. We conclude by describing our future work related to Albatross.

I. Introduction

Since the launch of Bitcoin in 2009, many more blockchains and distributed ledgers have emerged. While this technology has the potential to revolutionize our markets, its main problem is still scalability. While traditional, custodial ledgers can achieve throughputs of thousands of transactions per second, distributed, non-custodial ledgers are currently still lacking behind. Among others, one approach towards solving the scalability problem lies in alternative consensus protocols.

Consensus protocols have been subject to research for decades. The most fundamental difference between classical Byzantine fault tolerant (BFT) consensus protocols and modern distributed ledgers is that the former are designed to work in closed, size-limited networks only, whereas the latter focus on open, permissionless networks.

The most famous classical consensus algorithm is PBFT, or practical Byzantine fault tolerance [1]. While PBFT is a major component in many blockchain consensus algorithms, classical consensus theory has evolved significantly. Nowadays, the BFT algorithms providing the highest performance are speculative BFT algorithms.

Speculative BFT refers to a class of algorithms that has two modes for consensus: (1) the *optimistic* mode, where it is assumed that the nodes are well-behaved and little security measures are applied, instead preferring speed, and (2) the *pessimistic* mode, where no such assumption is made, and the only goal is to make progress even in the presence of malicious nodes. The optimistic mode allows these algorithms to compete with centralized systems in terms of speed.

In the following, we introduce our novel blockchain consensus algorithm, called Albatross. Albatross is a Proof-of-Stake (PoS) based algorithm for open, permissionless networks. Albatross is inspired by speculative BFT algorithms and assumes a cryptocurrency setting, i.e., the underlying token has a monetary value. In this paper, we introduce Albatross on an intuitive level and then provide more details on the individual parts of the protocol. We argue that the protocol is secure under regular PBFT security assumptions and has a theoretical performance close to the maximum for single-chain Proof-of-Stake consensus algorithms. We then conclude and discuss our future work.

II. Related Work

The area of consensus protocols is a vast and quickly developing landscape. In their SoK, Bano et al. [2] present an excellent classification and comparison of consensus protocols.

The main difference between Albatross and PBFT-based protocols such as Tendermint [3] is the speculative nature of our protocol. Being modeled after speculative BFT algorithms like Zyzzyva [4], Albatross strives for the best performance in the optimistic setting. Moreover, in contrast to Tendermint, Albatross allows for much larger validator sets.

Our validator selection is inspired by Byzcoin [5], which also introduces two different kinds of block types: key blocks determine the next validator set and micro blocks contain the transactions. Byzcoin uses a hybrid approach, determining key blocks via Proof-of-Work and relying on PBFT for the micro blocks. Albatross uses PBFT for the key block equivalent and an optimistic protocol for the micro blocks.

Albatross determines the list of validators using a Verifiable Random Function similarly to Algorand [6]. While Algorand offers an improved resistance against adaptive adversaries, this comes at the expense of forks and slower convergence.

III. Overview

In this section, we give a summary of Albatross. We will focus on the general structure of Albatross.

A. Validators

The participants responsible for producing blocks are called validators. There are two types of validators: potential and active validators. Given a certain point in time, a *potential validator* is any node that has staked tokens (i.e., tokens that are locked away), so that it can be selected for block production. An *active validator* is a potential validator that was selected to produce blocks and thus has an active part

978-1-7281-3669-1/19/$31.00 ©2019 IEEE
DOI 10.1109/CVCBT.2019.000-1

during the current *epoch*. An *epoch* is the period of time for which a set of active validators was selected.

The set of all active validators is called the *validator list* and is chosen at random from all potential validators. The probability of being chosen is proportional to the stake.

B. Blocks

There are two types of blocks produced by the validators:

Macro blocks are infrequent and used to change the validator list. They contain only the identities of the new active validators and the random seed used to select them. Macro blocks are produced with classical PBFT.

Micro blocks contain transactions. Each micro block is produced by a randomly chosen active validator and also contains a random seed produced by the validator. Micro blocks only need to be signed by the selected validator.

One macro block is always followed by m micro blocks. An *epoch* is composed of a macro block and the m micro blocks that preceded it.

C. Optimistic Mode

Next, we describe our protocol during an epoch, assuming honest, non-failing validators. We call this the *optimistic mode*.

We start with a macro block containing a random seed. This seed is used to randomly select the new validator list proportionally to the validators' stake. The seed also determines which validator, from the new validator list, is allowed to produce the first micro block in the following epoch.

1) The chosen validator, also called the *slot owner*, now produces and signs his micro block. It contains a new random seed produced by a *verifiable random function* (VRF) that takes as an input the previous block's seed. This new random seed is used to select the next block producer from the validator list.
2) Then, the next slot owner repeats the process and produces another micro block. This continues until the last micro block in the epoch, which will pick the validator that will be PBFT leader for the macro block.
3) The PBFT leader will produce the new random seed using the VRF, which will be used to select a new validator list. The leader creates a signed macro block proposal and sends it to the other validators. The signature is not part of the block and only required to ensure integrity. The validators then run the PBFT protocol to agree on the proposal.

We instantiate the VRF used in our protocol by BLS signatures [7]. Hence, producing the next random seed boils down to signing the previous random seed.

D. Misbehaving Validators

The protocol above constitutes the optimistic case. However, Albatross also needs to be able to withstand malicious or failing validators. To this end, we present measures for the three ways in which a validator can misbehave: producing an invalid block, creating a fork, and delaying a block.

1) Invalid Blocks: When a validator produces an invalid block the other validators ignore that block. Additionally, they will ignore any more blocks from that validator during the current slot to prevent DoS attacks.

2) Forks: Forks are not possible during a macro block, because PBFT is a forkless protocol. However, a fork can be created if a validator produces more than one micro block in the same slot. To deal with such forks and counter the *nothing-at-stake issue*, we introduce *slash inherents*. A slash inherent confiscates the stake of a validator that produced a fork. Anyone can create a slash inherent; they only need a proof that the validator forked, i.e., two block headers, at the same slot, signed by the same validator.

The consequences of a slashing a malicious validator are:

- The validator is no longer considered in the slot owner selection, but he can still participate in the PBFT protocol's voting phase.
- His stake is confiscated and divided equally between the rest of the validator list to incentivize reporting.

3) Delays: Validators can potentially take a long time to produce a block, either because they went offline or because they maliciously try to delay the block production. In both cases, we need to change the slot owner. To this end, we employ the *view change* protocol of PBFT. After receiving a block, each validator starts a timer. If he does not receive the next block before the timeout, he requests a view change. This procedure is used for micro and macro blocks.

IV. Protocol Details

We now give a more detailed specification of the different parts of our protocol. This specification aims at complementing the intuitive overview from the previous section.

A. Validator Signaling

There are two types of transactions that nodes can send to the network to change their validator status.

A *staking* transaction locks away a specified token amount. The sender is added to the registry of potential validators.

When a node wishes to stop being a validator, it needs to send an *unstaking* transaction. If the node is still an *active* validator, it is required to remain a validator until the end of the epoch. The deposit can only be withdrawn m blocks after the end of the epoch. This is required to allow the new validator list to punish past validators.

B. Validator Selection

There are two situations in which we need to select a random subset of validators: at the end of an epoch when a new validator list is chosen and at every micro and macro block to choose the next slot owner from the active validators.

1) Validator List: In Albatross the entire validator list is changed every epoch. We use the random seed present in every block to select the new set.

To select the validator list, we order the potential validators' keys deterministically and map the keys to their deposit amounts, such that each amount represents a range. For

example, if there are 10 tokens staked by the validator with key pk_a, 50 tokens by pk_b and 15 tokens by pk_c, then the corresponding ranges would be $[0, 9]$, $[10, 59]$, and $[60, 74]$.

We can now run **Algorithm 1** to select the validator list.

Algorithm 1 Validator List Selection Algorithm

```
validator list = [ ]
S = random seed
t = total amount staked;
i = 0
while validator list not full do
    r = hash(S || i) mod t
    v = potential validator whose range contains r
    add v to validator list
    i++
end while
```

2) Slot Owners: Given a random seed and a validator list, we can randomly choose an infinite ordered list of slot owners. While ideally only the first validator of that list will produce the block, the rest of the list can be relied upon in case of view changes (see Section IV-C).

The algorithm for calculating this infinite ordered list follows the previous **Algorithm 1**: First, we take the ordered validating keys of all the n active validators. If we number them from 0 to $n-1$ instead of assigning ranges and set $t = n$, we can produce a list of slot owners of arbitrary length.

C. View Change Protocol

If a validator does not produce a block during his slot, we need to select another validator instead. This process is initiated using the *view change* protocol and is closely modeled after the protocol of the same name in PBFT [1].

Given $3f + 1$ active validators (of which at most f are malicious; see Section V), an infinite list of slot owners $[s_1, s_2, \ldots]$, and a timeout parameter Δ, each active validator runs **Algorithm 2** after receiving a block. A *view change* number keeps track of the index within the infinite list.

Algorithm 2 View Change Algorithm

```
i = 0 (view change number)
loop
    wait for (i + 1) · Δ time
    if a valid block was received from s_i then
        terminate algorithm
    else
        send a view change message to all active validators
    end if
    if 2f + 1 view change messages are received then
        commit to not accepting a block from s_i in this slot
        i++
    end if
end loop
```

A *view change message* is a signed message containing a statement $\langle \text{VIEW-CHANGE}, i + 1, b \rangle$, where i is the current view change number and b is the current block number.

For the next block to be accepted, it must include the $2f+1$ view change messages accepting its producer at $i+1$. A block with a higher view change number has always priority over a block with a lower view change number.

D. Chain Selection

The chain selection algorithm needs to take into account malicious forks and view changes. We use the following cumulative conditions, from highest to lowest priority:

1) The chain with the most macro blocks.
2) The chain that has the highest view change numbers.
3) The chain with the most blocks.

If two chains tie on all three conditions, both chains are considered equal, and there is no clear chain to select. Thus, the next slot owner can build on top of either one.

V. Adversarial Model

We now discuss the adversarial model of Albatross. In particular, we give a bound on the maximum amount of malicious stake the protocol is resilient against. For our analysis, we rely on the safety and liveness of PBFT, which can tolerate up to $\lfloor \frac{n-1}{3} \rfloor$ of malicious nodes [1], [8].

In Albatross, the validator list is chosen randomly from the larger set of potential validators. For a validator list of size n, if someone controls a fraction p of the entire stake, then the probability of him gaining control of at least x validators is given by the cumulative binomial distribution:
$P(X \geq x) = \sum_{k=x}^{n} \binom{n}{k} p^k (1-p)^{n-k}$.

The maximum fraction of stake p an adversary may control before exceeding the bound given by PBFT derives from: $P(X \geq \lfloor \frac{n-1}{3} \rfloor) \leq \epsilon$, with $\epsilon > 0$ being practically negligible.

For example, if Albatross has a validator list greater than 500 validators ($\epsilon \leq 2 \cdot 10^{-5}$), we can consider the statement *"controlling less than $\frac{1}{4}$ of the total stake"* to be with overwhelming probability equivalent to the statement *"controlling less than $\frac{1}{3}$ of the validator list"*.

We argue that Albatross is secure as long as the statement *"malicious validators control less than $\frac{1}{4}$ of the total stake"* is true and $n \geq 500$.

In the following, we will briefly discuss how different types of adversaries impact our security guarantee. We also describe our underlying network model. A more detailed analysis is available and will be included in a future full version.

A. Static Adversary

A static adversary can corrupt specific nodes at the beginning of the protocol but later cannot change which nodes are corrupted. As long as the adversary does not corrupt nodes controlling more than $\frac{1}{4}$ of the total stake at any point in time, the protocol remains secure.

B. Adaptive Adversary

Adaptive adversaries can only corrupt a given number of nodes but, at any time, can change which nodes are corrupted. We consider the case in which the adversary can only corrupt validators controlling not more than $\frac{1}{4}$ of the stake in total.

Such an adversary could harm the liveness of the protocol by continously corrupting the current slot owner and refusing to produce blocks. Since the slot owner is only selected in the previous block, the success of this attack depends on the network and the time to produce a block. Independently of the strategy used, an adaptive adversary would need to corrupt nodes on the order of seconds. We discuss ways to increase resistance against such adversaries in the future work section.

C. Network Model

We follow the same network model as in PBFT. The protocol provides safety even in a fully asynchronous network. To achieve liveness, the protocol relies on partial synchrony.

Also, similar to PBFT, Albatross favors safety in case of network partitions. From the *CAP theorem* [9], we know that when suffering a network partition a blockchain can only maintain either consistency or availability. While PBFT will stop in the presence of a network partition, Albatross can still produce a few micro blocks before temporarily halting.

VI. Performance Analysis

In this section, we give a theoretical analysis of the performance of Albatross. We show that, in the *optimistic* case, it achieves the theoretical limit for single-chain PoS algorithms.

A. Optimistic Case

The best possible case is when the network is synchronous (all messages are delivered within a maximum delay d), d is smaller than the timeout Δ and all validators are honest.

The macro blocks have a message complexity of $\mathcal{O}(n^2)$ since they are produced with PBFT. However, as they constitute a tiny percentage of all blocks, the overall performance is mostly correlated with the micro block production.

Micro blocks have a message complexity of $\mathcal{O}(1)$. The latency, if we ignore the time spent verifying blocks and transactions, is equal to the block propagation time, which is on the order of the network delay d.

In conclusion, Albatross, in the optimistic case, produces blocks as fast as the network allows it.

B. Pessimistic Case

If we relax some of the assumptions made for the optimistic case, Albatross still has a performance superior to PBFT. There are two different cases that we will consider:

Malicious validators: The worst case, while still maintaining security, is a scenario with f validators out of $3f + 1$ being malicious and refusing to produce blocks. In this case, we can expect one view change every three blocks, requiring $\mathcal{O}(n)$ messages and waiting for a timeout. Thus, the message complexity is $\mathcal{O}(n)$, and the latency is on the order of Δ.

Network delay larger than timeout: If $d > \Delta$, every block will only be produced after a given number of view changes and several short-lived forks may be created for each block. Since we still only rely on the view change protocol, the message complexity is $\mathcal{O}(n)$, and the latency is $> d$.

VII. Conclusion

In this paper, we described and analyzed Albatross, a novel consensus algorithm inspired by speculative BFT algorithms. The most notable differences between classical PBFT and Albatross are: (1) making it permissionless by selecting a validator list proportionally to stake, (2) increasing resistance to adaptive adversaries by only selecting block producers on the previous block using a VRF, and (3) increasing performance by relying on speculative execution of blocks.

Despite sacrificing strong finality, Albatross has a strong confirmation which, when coupled with low block latency, means that transactions can have a very high probability of being final in just a few seconds.

VIII. Future Work

We are currently in the process of extending and improving Albatross on several ends.

This includes reducing the message complexity of macro blocks using strategies similar to Handel [10] or Byzcoin [5]. Moreover, we want to further increase resistance of validators against adaptive adversaries by employing our own version of a secret single leader election that allows the next slot owner to determine his role secretly and provide a verifiable proof only when publishing his block. We also consider replacing the view change mechanism by verifiable delay functions to reduce communication complexity further.

We are building a simulator to evaluate the performance and determine suitable parameters for our protocol. We will run our simulation on realistic network topologies as measured from live blockchain networks.

References

[1] M. Castro, B. Liskov *et al.*, "Practical byzantine fault tolerance," in *OSDI*, vol. 99, 1999, pp. 173–186.

[2] S. Bano, A. Sonnino, M. Al-Bassam, S. Azouvi, P. McCorry, S. Meiklejohn, and G. Danezis, "Consensus in the age of blockchains," *arXiv preprint arXiv:1711.03936*, 2017.

[3] E. Buchman, J. Kwon, and Z. Milosevic, "The latest gossip on BFT consensus," *arXiv preprint arXiv:1807.04938*, 2018.

[4] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: speculative byzantine fault tolerance," in *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6. ACM, 2007, pp. 45–58.

[5] E. K. Kogias, P. Jovanovic, N. Gailly, I. Khoffi, L. Gasser, and B. Ford, "Enhancing bitcoin security and performance with strong consistency via collective signing," in *25th USENIX Security Symposium (USENIX Security 16)*, 2016, pp. 279–296.

[6] 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*. ACM, 2017, pp. 51–68.

[7] D. Boneh, B. Lynn, and H. Shacham, "Short signatures from the weil pairing," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2001, pp. 514–532.

[8] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 4, no. 3, pp. 382–401, 1982.

[9] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *Acm Sigact News*, vol. 33, no. 2, pp. 51–59, 2002.

[10] N. Gailly, N. Liochon, O. Bégassat, and B. Kolad, "Handel: Practical multi-signature aggregation for large byzantine committees," 2019. [Online]. Available: https://github.com/ConsenSys/handel