Partitionable Asynchronous Cryptocurrency Blockchain

Kendric Hood, Joseph Oglio, Mikhail Nesterenko, and Gokarna Sharma[1]
Department of Computer Science, Kent State University, Kent, OH 44242, USA
khood5@kent.edu, joglio@kent.edu, mikhail@cs.kent.edu, sharma@cs.kent.edu

***Abstract*—We consider operation of blockchain-based cryptocurrency in case of partitioning. We define the Partitionable Blockchain Consensus Problem. The problem may have an interesting solution if the partitions proceed independently by splitting accounts. We prove that this problem is not solvable in the asynchronous system. The peers in the two partitions may not agree on the last jointly mined block or, alternatively, on the starting point of independent concurrent computation. We introduce a family of detectors that enable a solution. We establish the relationship between detectors. We present the algorithm that solves the Partitionable Blockchain Consensus Problem using our detectors. We extend our solution to multiple splits, message loss and to partition merging. We simulate and evaluate the performance of detectors, discuss the implementation of the detectors and future work.**

***Index Terms*—Blockchain, Consensus, Network partitions, Detectors, Validity, Confirmation, Branch, High frequency trading.**

I. Introduction

Peer-to-peer networks are an attractive way to organize distributed computing. Blockchain is a technology for building a shared, immutable, distributed ledger. Blockchain is typically maintained by a peer-to-peer network. A prominent blockchain application is cryptocurrency such as Bitcoin [1] or Ethereum [2]. In a blockchain cryptocurrency, the ledger records financial transactions. The transactions are appended to the ledger block by block. The application is decentralized and peers have to agree on each block. Such consensus is the foundation of blockchain algorithms.

Classic robust distributed consensus algorithms [3], [4] use cooperative message exchange between peers to arrive at a joint decision. However, such algorithms require participants to know the identity of all peers in the network. This is not often feasible for modern high-turnover peer-to-peer networks.

Instead, Bitcoin uses Nakamoto consensus [1] where the participants compete to add blocks to the ledger. This algorithm does not require complete network knowledge. Due to the simplicity and robustness of the algorithm, Nakamoto consensus became widely-used in cryptocurrency design.

For their proper operation, blockchain consensus algorithms assume that the network remains connected at all times. The network may not confirm transactions while parts of the network are unable to communicate. Alternatively, a single primary partition makes progress while the others are not utilized. This is not accidental as the problem of concurrently using partitions is difficult to handle: the partitioned peers may approve transactions that conflict and thus violate the integrity of the combined blockchain. Ultimately, the problem is stated in the classic CAP Theorem [5] claiming that it is impossible co-satisfy consistency, availability and partition-tolerance in a distributed system.

It is assumed that partitioning interruptions are infrequent or insignificant for network operation. However, this may not be the case. While long network-wide splits may be rare, brief separations are common. This can happen, for example, if groups of peers are connected via a small set of channels. If these channels are congested, the groups are effectively cut off from each other and the network is temporarily partitioned.

As blockchain becomes a larger component of the financial market, the pressure for the system's availability will increase. Many financial applications, such as high frequency trading [6], are sensitive to even a slight delay. A system delay that lasts a few milliseconds may cost its users substantial time and money. Thus, considering the blockchain-based cryptocurrency that is available through partitioning is required for this technology to realize its potential.

Our contribution. In this paper, we formally state the Partitionable Blockchain Consensus Problem. What enables the solution is splitting the accounts in case of partition: once the peers detect a network split, they decrease the amount of money available on each account. For example, in case of a split, all peers decrease the available amount on all accounts by half. The effect of such split is that the money is distributed between the partitions and blockchain operation continues while the integrity of the complete blockchain is preserved.

We study partitionable consensus in the asynchronous system model. The model does not place assumptions on the peers' relative computation power or message propagation delay and thus has near-universal applicability. Deterministic consensus is impossible in the asynchronous system even if a single peer crashes [7]. Intuitively, peers are not able to distinguish a crashed process from a very slow one. This impossibility is circumvented with crash failure detectors [8], [9] that provide minimum synchrony to allow a solution.

We pattern our investigation on this classic approach. We show the impossibility of a partitionable consensus solution in the asynchronous system. The intuition for the impossibility is as follows. For correct post-network-split system operation, the peers have to agree on the last common pre-split state. If there is no direct way to determine whether the split occurs, the peers may only infer this information from message

[1]This research was supported in part by National Science Foundation under Grants No. CCF-1936450 and CAREER CNS-2045597.

978-0-7381-1420-0/21/$31.00

communication. Yet, in an asynchronous system, a sender may not know whether its message was delivered or lost due to split. Hence the lack of solution.

We introduce a family of detectors that encapsulate the necessary partitioning knowledge and enable such a solution. We then present an algorithm that solves the Partitionable Blockchain Consensus Problem using the detectors. To simplify the presentation, we first consider a single split with no message loss and no subsequent merging. We then extend our solution to accommodate message loss, multiple splits and temporary splits and merging. We implement our algorithm and evaluate its performance.

Related work. There is a number of recent publications dealing with the implementation or modification of classic consensus [10], [11], [12]. One promising approach to bypass the impossibility of deterministic asynchronous consensus is to use a fast asynchronous randomized consensus algorithm [13], [14] for blockchain construction [15], [16], [17]. There are plenty of recent studies presenting blockchain design based on Nakamoto consensus [18], [19], [20], [1], [21]. There are papers that combine classic and Nakamoto consensus [22], [23]. Recent research on Nakamoto-based blockchain often focuses on improving its speed and scalability [24], [25], [26], [27]. One promising blockchain acceleration technique is to concurrently build a DAG of blocks [28], [29]. The state-of-the-art on the blockchain consensus algorithms can be found in this recent survey [30]. To the best of our knowledge, none of the above blockchains allow multiple partitions to confirm transactions concurrently. That is, these solutions are not partitionable.

Relatively few publications focus on partitionable blockchains. There are some studies where the partitionable classic consensus either addressed directly [31] or using failure detectors [32], [33]. Partitionable consensus has similarities with the group membership problem, which deals with presenting a consistent membership set to the processes despite process and link failures [34], [35], [36]

Tran *et al.* [37] consider an algorithm that implements partitionable blockchain consensus in the context of swarm robotics. In swarm robotics, the robot swarms may experience network partitions due to navigational and communication challenges or in order to perform certain tasks efficiently. Their solution extends EVS and hence is not suitable for partitionable blockchain under dynamic membership as we consider in this paper.

II. Notation

Communication model. A *peer* is a single process. All peers operate correctly and do not have faults. A *partition* is a collection of peers that can communicate. This communication is done through message passing. A broadcast sends a message to every peer in its partition. Communication channels have infinite capacity and are FIFO. The channels are reliable unless the network is split. A *(network) split* separates one partition into two. Peers are split into two arbitrary non-empty sets. A message sent before the network split is always delivered; a message sent after the split is delivered only to the recipients that are in the same partition as the sender.

Each peer contains a set of variables and commands. A *network state* is an assignment of a value to each variable from its domain. An *action* is an execution of a command. An action transitions the network from one state to another. A *computation* is a sequence of states resulting from actions. Actions in a computation are atomic and do not overlap. We consider the network split to be a particular kind of action. A computation is either infinite or ends in a state where no action may be executed. A *computation segment* is a portion of a computation that starts and ends in a state. A computation segment from the initial state to a particular state is a computation *prefix*. A, possibly infinite, computation segment following a particular state is a *suffix*.

We assume fair action execution and fair message receipt. Specifically, in any computation, any action is eventually either executed or disabled; any sent message is eventually received. We assume there is at most one split per computation. That is, the network starts as one partition and it may split into two. Observe that this means that fairness does not apply to a network split action: the split may not happen. Since we are considering the purely asynchronous system model, there may be no re-connections. If a split occurs, the two partitions exist for the rest of the computation. We relax the single split and no re-connection assumption further in the paper.

Accounts and transactions. An *account* is a means of storing funds. Each account has a unique identifier. A *transaction* is a transfer of funds from the source to the target account. For simplicity, we assume that there is a single source and a single target account. Each transaction has a unique identifier as well.

A *client* is an entity that submits transactions to the network via broadcast. There may be multiple clients. The transaction identifiers for each client are monotonically increasing. A client submits each transaction to a single partition. If transactions are submitted to two partitions, they are considered separate transactions.

Blockchain. Peers mine transactions. Such a mined transaction is a *block*. That is, to simplify the presentation, we assume a single transaction per block. A mined block cannot be altered. Besides a transaction, each block contains an identifier of another block. Thus, a block is linked to another block. A *blockchain* is a collection of such linked blocks. A *genesis* is the first block in the blockchain. The genesis is unique. There are no cycles in the blockchain. That is, the blockchain is a tree. A *branch* of a tree is a chain of blocks from the genesis to one of the leaves of the tree. See, for example, a branch from the genesis to block 1 in Figure 2.

The *main chain* is the longest branch in a blockchain. Ties are broken deterministically. A *permanent branch* is infinite. We assume that there is at most one permanent branch per partition.

Each peer operates as follows. If it receives a transaction, it stores it. The peer attempts to mine one of the pending transactions by linking it to the tail of its main chain. If it succeeds, the block is immediately broadcast. The peer

continues while there are pending transactions. A peer may quit trying to mine a transaction and switch to mining another one. For example, if a new block arrives, a peer may switch to mining on top of it. We make the following assumption: if a peer receives infinitely many new transactions, then the peer either receives infinitely many mined blocks or mines infinitely many blocks itself.

Global blockchain (tree) is the collection of all blocks mined by all peers. A *fork* is the case of multiple blocks linking to the same block. A fork happens when several peers succeed in concurrently mining blocks. See Figure 2 for an example. Since the genesis block is unique, it may never be in a fork. If there is a network split, a *seed* is the last block mined on any branch before split. That is, at the time of a split, the last block on every branch is a seed.

Consider a block b on the blockchain tree and a branch that leads from the genesis to b. A *balance* for any particular account a with respect to b is the sum of all funds that are transferred into a by the transactions of the blocks in this branch minus the funds that are transferred out of a.

A transaction is *valid* if its application leaves the source account with a non-negative balance. The transaction is *invalid* otherwise. The transactions may differ depending on the particular branch of the tree. Therefore, a transaction may be valid in one branch and invalid in another. If a peer mines a transaction, it is valid relative to its main chain. That is, peers mine only valid transactions.

A transaction is *confirmed* if it is in the permanent branch. It is *rejected* if it is not in the permanent branch. A transaction is *resolved* if it is either confirmed or rejected. A transaction is *permanently valid* if it is valid indefinitely or until it is resolved. We assume that in each partition, at least one client submits infinitely many permanently valid transactions.

Account splits. In case of a network split, the account balances may also be split. That is, the peers consider the amount of funds available on a particular account to be a fraction of the original amount. Accounts are split in the seed: the last pre-split block in the blockchain. The blockchain may have multiple branches and, therefore, multiple seeds. Thus, accounts may potentially be split in different seeds. See Figure 2 for example.

To eliminate a trivial case, we assume that at least some funds are distributed between partitions. That is, we exclude the case where a partition is left with zero funds for all accounts. Otherwise, we place no restrictions on the way that accounts are split between partitions so long as the total on each account balance in both partitions post-split is equal to the pre-split account balance in the seed block. For example, suppose there is an account a that has a balance of 100. Then a network split occurs. Account a may be split into a_1 and a_2. Accounts a_1 and a_2 cannot be in the same partition. Account a is split $70/30$, thus the balance of a_1 is 70 and a_2 is 30. If the accounts are split unevenly, each peer must know to which partition it belongs. If a is halved, i.e. split $50/50$, then the peers do not need to identify which partition they are in. Observe that a split affects the validity of a transaction. If the submitted transaction is not valid after split, it is not mined.

A *branch merge* is an arbitrary interleaving of transactions of two branches such that the order of transactions of each branch is preserved. Two branches are *mergeable* if all transactions mined before the split are resolved uniformly and any branch merge retains the validity of all transactions of the two branches.

Proposition 1: Branches from different partitions are mergeable if they are split on the same seed.

Detectors. A *detector* is a mechanism that provides information to the algorithm that it may not able to determine otherwise. Specifically, a detector is an algorithm whose actions may include information available outside the model. The *pure asynchronous* system has no detectors. The output variables of a detector provide information for other algorithms to use. The actions of the detector and the algorithm that uses it, interleave in a fair manner.

Detector A is *weaker* than detector B, if there exists an algorithm such that it accepts every computation of A and produces a computation of B. Detector A is *equivalent* to B, denoted $B \equiv A$, if both A is weaker than B and B is weaker than A. Detector A is *strictly weaker* than B, denoted $B \succ A$, if A is weaker than B but not equivalent to B.

III. The Partitionable Blockchain Consensus Problem and Solution Impossibility

Definition 1: The Partitionable Blockchain Consensus problem is the intersection of the following three properties:
confirmation validity: *no invalid transaction is confirmed;*
branch compatibility: *permanent branches are mergeable;*
progress: *every permanently valid transaction is eventually confirmed.*

The first two properties are safety while the progress property is liveness [38]. The algorithm satisfying these properties under partitioning ensures that valid transactions are confirmed regardless of the split.

We show that it is not possible to achieve partitionable blockchain consensus in the pure asynchronous system.

A transaction is *split-invalidated* if it is valid unless a split occurs. For example, if an account has a balance of 10, and this account is split $50/50$, then a transaction that spends 6 is split-invalidated. To exclude a trivial solution which rejects all split-invalidated transactions, we introduce the following definition. An algorithm is *regular* if there exits a computation of this algorithm where a split-invalidated transaction mined before the split is confirmed. An algorithm is *strictly regular* if it confirms all split-invalidated transactions mined before the split.

Lemma 1: A regular algorithm that solves the Partitionable Blockchain Consensus Problem may not confirm a split-invalidated transaction.

Lemma 1 states that a regular solution to the Partitionable Blockchain Consensus Problem may not resolve a split-invalidated transaction. However, this means that there is no regular solution to this problem at all. Hence the below theorem.

Theorem 1: There does not exist a regular solution to the Partitionable Blockchain Consensus Problem in the pure asynchronous system.
See [39] for a formal proof.

IV. Partitioning Detectors

The partitionable blockchain consensus problem is not solvable without detectors. The lack of solution is due to the impossibility of ascertaining whether the message reached all peers. Let us discuss detectors that may circumvent this and enable a solution. A propagation detector *PROP* addresses this concern directly: for each peer and for each block, *PROP* outputs whether this block is delivered to peers of the entire network or just for a single partition.

Let us give a more precise specification for *PROP*. Assume *PROP* is running concurrently with a blockchain consensus algorithm *ALG*. In *PROP*, each peer contains an input variable in and an output variable out. Initially, both variables contain $\perp$. Algorithm *ALG* can write to *in* and read from *out*. Let b be a block mined by *ALG* in some computation. If there is a suffix of the computation of *PROP* where $in = b$, then the computation of *PROP* consists of a prefix where $out = \perp$ followed by a suffix where either (i) $out =$ **true** if the block in *ALG* was received by all peers in the network or (ii) a suffix where $out =$ **false** if the block was delivered only to a partition. That is, *PROP* correctly classifies block receipts, does not make mistakes or changes its decision. This definition is extended to an arbitrary number of blocks in a straightforward manner: there is an input and output variable for each block in each peer.

Another detector *AGE* outputs whether the block was mined before or after the split. The detector classifies the pre-split block as *old*, and post-split block as *new*. The formal definition is similar to that of *PROP* above.

Since the block is broadcast right after mining and message transmission is reliable, the only way for a message not to be delivered to all peers is if there is a split in the network. Hence the following lemma.

constants
p // identifier of this peer
b // block whose age is evaluated

variables
$flips$ // array of changes in output from each peer, initially zero
$ages$ //array of most recent outputs of *WAGE*, initially old

commands
$ages[p]$ **not** $= WAGE(b) \longrightarrow$
$ages[p] = WAGE(b)$
increment $flips[p]$
$broadcast(flips[p], ages[p])$

receive $numFlips, age$ **from** $id \longrightarrow$
$flips[id] := numFlips$
$ages[id] := age$

$\diamond AGE(b) \longrightarrow$ // implements output of $\diamond AGE$
output $ages[id]$ for the id with minimum $flips[id]$

Fig. 1: Implementation of $\diamond AGE$ using *WAGE*.

Lemma 2: The propagation detector is equivalent to the age detector. That is: $PROP \equiv AGE$.

Detector eventual *AGE*, denoted $\diamond AGE$, is similar to *AGE*. Like *AGE*, detector $\diamond AGE$ outputs whether the block was mined before or after the split. However, $\diamond AGE$ is not reliable: $\diamond AGE$ may make a finite number of mistakes. Specifically, given a block mined by algorithm *ALG* running concurrently with $\diamond AGE$, if there is a suffix of computation of $\diamond AGE$ such that where $in = b$, then the computation of $\diamond AGE$ contains a suffix where $out = new$ if the block is mined before split or $out = old$ if the block is mined post-split.

To put another way, for each peer and for each block, $\diamond AGE$ is guaranteed to eventually output the correct result. To distinguish $\diamond AGE$, we call *AGE* the perfect age detector.

Lemma 3: The $\diamond AGE$ detector is strictly weaker than AGE. That is: $\diamond AGE \prec AGE$.
Formal proofs for the lemmas in this sections are in [39].

Detector *WAGE*, pronounced "weak-age", has output similar to *AGE* and $\diamond AGE$. However, unlike these two detectors, *WAGE* may make infinitely many mistakes subject to the following constraints. For each block for at least one peer per partition, the suffix of the computation of *WAGE* contains only correct output; for all other peers, every suffix contains infinitely many states with correct output. To put another way, *WAGE* ensures that at least one peer per partition eventually starts classifying blocks correctly and all other peers at least alternate their classifications without permanently settling on incorrect output. However, $\diamond AGE$ may be implemented using only *WAGE*. Figure 1 shows this implementation. Below lemma formalizes this statement.

Lemma 4: Eventual age detector is equivalent to weak age detector. That is: $\diamond AGE \equiv WAGE$.

Detector *SPLIT* outputs whether a split in the system has occurred. Observe that the message delivery is unreliable only in case of a network split. Therefore, the occurrence of the split can be determined if *PROP* indicates that a certain block has not propagated to the whole system. The converse is not in general true. Just the fact of a split does not allow peers to determine whether the particular block has reached every peer. Hence the following lemma.

Lemma 5: The split detector is strictly weaker than the propagation detector. That is: $SPLIT \prec PROP$.

This theorem summarizes the above lemmas.

Theorem 2: The relationship between partitioning detectors is as follows:

$$PROP \equiv AGE \succ \diamond AGE \equiv WAGE,\ PROP \succ SPLIT.$$

V. Algorithm *PART*

In this section we present an algorithm, we call *PART*, that solves the Partitionable Blockchain Consensus Problem. This algorithm is shown in Figure 3.

Algorithm description. The gist of the algorithm is as follows. The peers construct the blockchain tree and read the output of *AGE* to classify which portion of this tree is old and which is new. This way the peers agree on the same old block

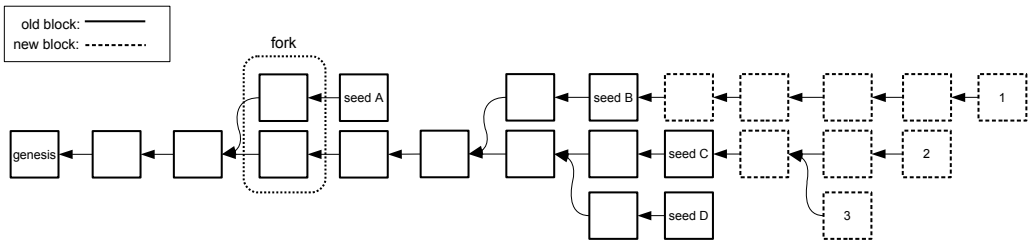

Fig. 2: Global blockchain tree generated by *PART*.

to be the seed used for splitting the accounts. The peers then proceed to mine new blocks according to the split accounts in the separate partitions on the basis of this seed.

Let us describe the algorithm in detail. Each peer maintains its copy of the blockchain bc, and a priority queue txs of all received transactions. The transactions are arranged in the order of their identifiers in txs. If a new transaction is received, it is entered into txs. We assume that txs is never empty. See Figure 2 for an example of a blockchain.

PART keeps track of the most recent output of the *AGE* detector in the $currentAge$ variable. When $currentAge$ is new, the block is mined with split accounts. The value of $currentAge$ is recorded in the mined block. Once the block is mined, *PART* checks the output of *AGE* against the new block and sets $currentAge$ accordingly.

Function $mainChain()$ of *PART* operates as follows. It consults *AGE* for all blocks in bc and constructs $trueTree$ with only those blocks whose recorded age matches the output of *AGE*. If *AGE* is perfect, the $trueTree$ contains all of bc. Function $mainChain()$ then builds $oldTree$ that contains only old blocks that are connected to the genesis block. Function $mainChain()$ then finds the longest branch $oldBrach$ in $oldTree$. Ties are broken deterministically. Then, $mainChain()$ examines the new branches of $trueTree$ connected to the tail of $oldBranch$ and selects the longest branch which it stores in $newBranch$. Function $mainChain()$ returns the concatenation of $oldBranch$ and $newBrach$, To summarize, $mainChain()$ operates on the subtree whose age agrees with the output of *AGE* and returns the longest old branch connected to the longest new branch. We refer to the output of $mainChain()$ as *main chain*.

Let us discuss the operation of $mainChain()$ using the example in Figure 2. The $oldTree$ there includes all blocks in the branches that run from the genesis to seeds A, B, C and D. Branches that run to C or D are of equal length and longer than the others. Assume the tie is broken in favor of C. Two new branches are attached to C. The branch that runs to block 2 is selected since it is the longer one. Function $mainChain()$ returns the branch that runs from the genesis to block 2. Note that even though the branch that runs to 1 is longer overall, old blocks are considered first. Thus, branch to 1 has a shorter old blocks branch.

Function $nextValid()$ returns the fist valid transaction in txs that is not in the main chain. Function $resetMining()$ checks whether the peer is currently mining the next valid transaction, and if not, it restarts mining.

PART operates as follows. Each peer continuously attempts to mine the first valid transaction in txs that is not in the main chain of bc. If mined, the recorded age of the block is compared to the output of the *AGE* detector. If they are the same, the newly mined block is added to bc and broadcast. If not, new age is recorded in $currentAge$, and the block is discarded. Then, the mining of the next transaction starts.

If the peer receives a block nb mined by another peer, it checks if this block is linked to any of the blocks in bc. If not then this block is added to $unlinked$, which is a set of such blocks. If nb is linked to bc, the peer inserts this block into bc then checks if any of blocks in $unlinked$ may now also be linked to bc.

Correctness proof.

Lemma 6: The main chain of every peer increases indefinitely.

Intuitively, the peer either mines the blocks itself, or receives infinitely many mined blocks. Hence, the main chain keeps increasing. See [39] for the formal proof.

Lemma 7: PART satisfies the progress property of the Partitionable Blockchain Consensus Problem.

Proof. Let t be a permanently valid transaction. Let us consider the suffix of the computation where each peer p receives t. Once p receives t, it enters t into txs. Each client submits transactions in the increasing order of their identifiers. That is, in any computation, there is only finitely many transactions with identifiers smaller than t.

By Lemma 6, the main chain of every peer increases indefinitely. Every block in this main chain is mined by some peer. Hence, there must be at least one peer p that mines infinitely many blocks in this main chain. By the design of the algorithm, each peer mines the valid transaction with the smallest identifier that is not in its main chain. Therefore, p may only mine finitely many blocks before t. Thus, an infinite main chain must contain t. □

Lemma 8: In *PART*, the permanent branches of all peers have a common prefix up to and including the seed block.

Intuitively, due to reliable message transmission, blocks mined before the split will be received by all peers. The formal proof is provided in [39].

variables
bc // tree of mined blocks, rooted in genesis
$unlinked$ // set of received blocks with missing intermediate links
txs // queue of received transactions, prioritized by id
$currentAge$ // true if accounts are split after partitioning

functions
$mainChain()$
$trueTree$:= blocks in bc whose age match AGE output
$oldTree$:= branches with only old blocks of $trueTree$ rooted in genesis
$oldBranch$:= longest branch in $oldTree$
$newTree$:= branches with only new blocks of $trueTree$ rooted in $tail(oldBranch)$
$newBranch$:= longest branch in newTree
return $oldBranch + newBranch$

$nextValid()$ // returns the first valid // transaction in txs that is not in $main(bc)$

functions (cont.)
$resetMining()$
if not mining $nextValid()$
startMining $nextValid()$ with $currentAge$
commands
receive transaction $t \longrightarrow$
$enqueue(txs, t)$
$resetMining()$

mine block $nb \longrightarrow$
$currentAge := AGE(nb)$
$insert(nb, bc)$
$broadcast(nb)$
$resetMining()$

receive mined block $nb \longrightarrow$
if possible to $insert(nb,bc)$ // add new block to blockchain
$insert(nb, bc)$
while exists b in $unlinked$ that can be inserted into bc
$insert(b, bc)$
else
add nb to $unlinked$
$resetMining()$

Fig. 3: Algorithm *PART*.

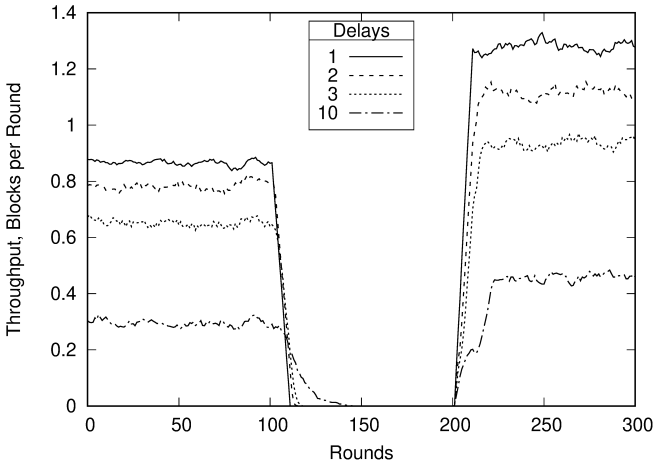

Fig. 4: *PART*+◇*AGE*. Split at round 100, detector recognizes it at round 200.

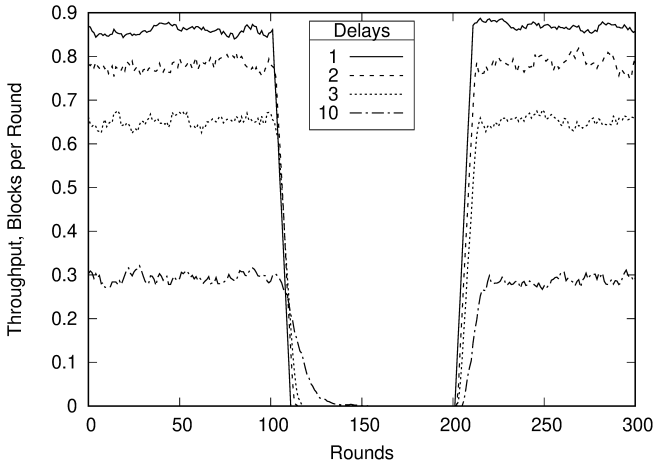

Fig. 5: *PART*+◇*AGE*. No split in computation, detector mistakenly recognizes it at round 100, corrects at round 200.

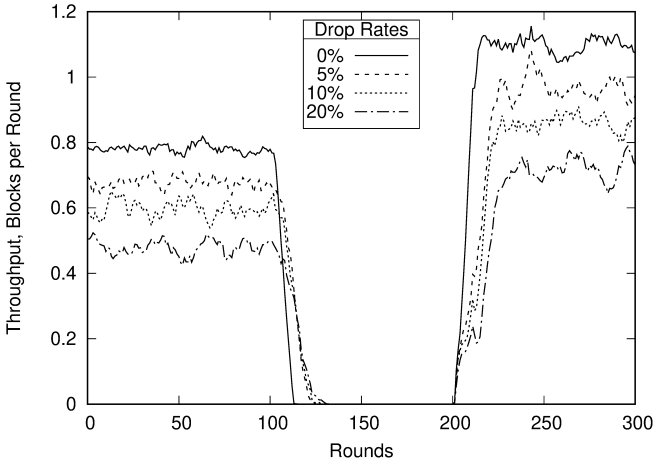

Fig. 6: *PART* + ◇*AGE* with message loss. Split at round 100, detector recognizes it at round 200. Message delay is 2 rounds.

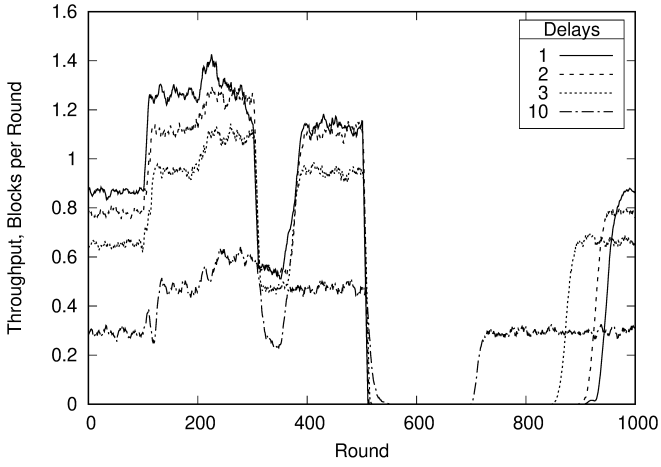

Fig. 7: *PART* with perfect *AGE* and multiple splits. No message loss. Split at round 100, second split at round 200. Merge second split at round 300, merge first split at round 500.

Lemma 9: Algorithm *PART* satisfies the branch compatibility property of the Partitionable Blockchain Consensus Problem.

Proof. According to Lemma 8, the main chains of all peers include the same seed block. Due to Proposition 1, branches from different partitions are mergeable. Hence the lemma. □

Theorem 3: Algorithm *PART* solves the Partitionable Blockchain Consensus Problem.

Proof. *PART* never confirms an invalid transaction. Thus, *PART* satisfies confirmation validity. Branch compatibility satisfaction is shown in Lemma 9. Progress satisfaction by *PART* is shown in Lemma 7. □

Observe that the presented algorithm operates correctly even if the detector makes finitely many mistakes. That is, if the detector is $\diamond AGE$. We call this combination $PART + \diamond AGE$. Indeed, once $\diamond AGE$ converges to the correct output for all blocks and each peer receives all pre-split blocks, all peers agree on the seed. From this point, *PART* operates as with perfect *AGE*.

Per Theorem 2, $\diamond AGE$ may be implemented using a weaker detector *WAGE*. Let $PART + WAGE$ be the combination of *PART* and such an implementation. Such a combination also solves the partitioning problem. Hence the following theorem.

Theorem 4: Algorithm $PART + \diamond AGE$ and $PART + WAGE$ solve the Partitioning Blockchain Consensus Problem.

VI. Performance Evaluation

Setup. We evaluate the performance of *PART* using an abstract simulation. We study the behavior of our algorithm through computations that we construct. The code for our simulation is available on GitHub [40].

The simulated network consists of n peers. An individual computation is a sequence of rounds. In every round, each peer may receive new messages from each other peer, do local computation, and send messages to other peers.

Each peer in the network has a unique channel to every other peer. Message delivery is FIFO. In a single round, a peer may receive messages from each sender. Message propagation may take several rounds. Each message is delayed by a number of rounds. This delay is selected uniformly at random. That values range from 1 to maximum delay d. Concurrent messages from the same sender do not impede other messages. That is, multiple messages from the same sender may be received in the same round.

The transaction submission rate is constant: one transaction per round. A submitted transaction is broadcast by a randomly selected peer. Block mining is simulated. Mining time is as follows. Each peer has an oracle that tells the peer whether it mined a block. In every round, the probability of mining a block for each peer is uniformly distributed between 1 and $d \cdot n$. The network size is 100 peers. A split separates the network into two equally sized partitions of 50 peers. Overall transaction rate or mining rate does not change in the event of a split.

We measure algorithm throughput: the ratio of confirmed to submitted transactions. We measure throughput under various delays. We plot the rolling average of number of confirmed transactions over the last 10 rounds. We run 100 experiments per each data point. To eliminate startup effects, we do not plot the first 100 rounds.

Results and analysis. The results of our experiments are shown in Figures 4, 5, 6, and 7.

In Figure 4, we show the results of the experiments with *PART* and $\diamond AGE$. The split occurs in round 100. However, $\diamond AGE$ continues to classify all blocks as old until round 200. The detector operates correctly afterwards. Between rounds 100 and 200, while the detector classifies blocks incorrectly, transactions are not confirmed. Therefore, the throughput decreases. Once the detector corrects itself, the old blocks are ignored by the algorithm, new blocks are mined and the throughput recovers, This post-split throughput of the algorithm is higher since, instead of causing forks, blocks are concurrently confirmed in the two partitions.

Figure 5 also shows the results of *PART* with $\diamond AGE$. In this case, there is no split, however, between rounds 100 and 200, the detector classifies all blocks as new. The detector recovers and starts classifying all blocks as old after round 200. The algorithms behavior is similar to that shown in Figure 4. Note, that there is no actual split in this experiment. Therefore, after the detector recovers, the number of forks does not decrease.

VII. Mulitple Splits, Message Loss, Partition Merging

Multiple splits. Let us consider the case of multiple split actions in a single computation. That is, a network partition may further split. Each individual split event separates a partition into two. We introduce two new detectors to handle this case.

The perfect multiple block age detector *MAGE* is the modification of *AGE*. *MAGE* operates as follows. For each block b, *MAGE* outputs the number of split events that happened in the partition where this block is mined. A way to think about *MAGE* is to consider that it repeatedly acts as a single-split *AGE* detector for each partition. For example, if the partition is never split, *MAGE* outputs 0. If the network splits into two partitions A and A', *MAGE* outputs 1 for the peers of both partitions. If A splits into B and B', then *MAGE* outputs 2 for the peers in B and B' and still 1 for the peers of A'.

The eventual multiple block age detector $\diamond MAGE$ and weak multiple block age detector *WMAGE* are defined similarly to their single-split counterparts. Algorithm *PART* operates with *MAGE*, $\diamond MAGE$ and *WMAGE* without modifications. We summarize this in the following theorem.

Theorem 5: Algorithms $PART + MAGE$, $PART + \diamond MAGE$, and $PART + WMAGE$ solve the Partitionable Blockchain Consensus Problem with multiple splits.

Message loss. To tolerate message loss, the peers in algorithm *PART* need to be able to recover lost messages. *Disconnected subtree* is a collection of linked blocks not connected to the genesis block. Such collection has a single root. If a peer has

a disconnected subtree, it is missing a block linking it to the genesis. This block may be delayed or lost.

The *block catchup procedure* recovers the missing blocks. It contains two actions: (i) if the peer contains a disconnected subtree, broadcast request for the block preceding its root; (ii) if a peer receives such request and has the requested block, broadcast this block. Note that the correctness of the block catchup procedure does not depend on the timing of the request action so long as it is eventually executed.

To be able to guarantee meaningful liveness, we restrict message loss as follows: if the same message is broadcast by the same peer infinitely many times, it is also delivered infinitely many times. If intermediate blocks are not delivered to a partition, there is no way to recover it after the split. Hence, we place another assumption. If a block b is delivered to one of the peers in the partition, every block in the branch of b, i,e, on the chain from the genesis to b is also delivered to one of the peers in this partition. With these assumptions we are able to state the following theorem.

Theorem 6: Algorithm *PART* with block catchup procedure solves the Partitionable Blockchain Consensus Problem with message loss.

Figure 6 evaluates the operation of *PART* with message loss.

Partition merging. There are two ways of handling temporary split. It can be considered a special case of message loss. In this case the above message loss version of *PART* operates correctly. However, in this *competitive merge*, the blocks mined by one of the partitions are discarded. This may be inefficient. Alternatively, we may implement *cooperative merge* that retains some of the blocks of both partitions. In this case *PART* and detectors need to be modified.

Detector *SMAGE* for split-merge *AGE*, correctly identifies whether the block was mined when partition was split. That is, for each block, the output for *SMAGE* is whether or not this block was mined during the split. For example, before the split, for each block *SMAGE* indicates *no-split*; for blocks mined after the split, *SMAGE* indicates *split*; after merge, for each subsequently mined blocks, *SMAGE* indicates *no-split* again. Detectors $\diamond SMAGE$ and *WSMAGE* are defined similarly to their split-only analogues.

To implement cooperative merge, algorithm *SMPART* modifies the original *PART* as follows. All peers run block catchup procedure that counters message loss. Each peer examines the output of *SMAGE* on the received block. If there was a switch *no-split*, *split*, *no-split* in at least one of the branches of the tree, then the peer determines that there was a split and then a merge. In this case, the peer finds the leaves of the two longest non-conflicting branches formed during partition. The peer then mines *merge block* that, rather than to a single block, links to these two blocks. This way, both branches are confirmed. If multiple such merge-blocks are mined, the block on the longest branch wins. Ties are broken deterministically. After the merge, the merge-block acts as a seed, the accounts are combined and the computation proceeds as pre-split.

The algorithm and detectors can be extended to multiple partition merge similar to multiple partition split. In case of multiple splits and merges, the accounts are combined on the basis of the account share each partition had pre-merge. Consider an example. Assume $50/50$ split. If the network splits into two A and A', then A splits into B and B', and then B' merges with A' into C. Then, B contains 25% of the account balances while C contains 75%.

Figure 7 shows the results of the experiments with thus implemented algorithm *SMPART* and *SMAGE* detector. The detector does not make mistakes. There is no message loss. There are two consecutive splits that then merge back.

VIII. Other Extensions and Future Work

Detector implementation, block purging. The pure asynchronous system allows us to reason about the essential properties of the algorithm that do not rely on timing assumptions. Nonetheless, we would like to outline certain implementation and usage aspects of the proposed algorithm and detectors.

The age detector may be implemented with checkpointing. The idea is as follows. The peers agree on a checkpoint block on every branch of the blockchain. Once the split occurs, if a certain block precedes the checkpoint block, it is considered old. A block is new if it follows the checkpoint block. To limit the rollback overhead, the checkpoints are moved closer to the leaves of the blockchain as the computation progresses.

Checkpoints can also be utilized to save memory space used to store the blockchain branches. Since the peers never roll back past checkpoint blocks, rather than storing individual old blocks, it is sufficient to just store the resultant checkpoint account balances. The old blocks may then be purged from memory.

Optimizing recovery time, utilizing mining rate. In present implementation, post-split, the peers have to wait for the blocks mined by other partition to resume operation. This recovery may compromise overall throughput. Thus, an efficient post-split catch up procedure, such as broadcasting the whole split-branch at once to the other partition, may need to be implemented.

In our experiments, the transaction submission and mining rate is fixed. In reality, both may vary in time and space. For example, the block mining rate is roughly proportional to the partition size. A partitioning detector may observe the fluctuations in the mining rate to predict network partitioning and merging.

Fault tolerance. Let us discuss how the proposed partitionable blockchain may withstand other faults. Peer crashes may be problematic as the blockchain has no way of determining whether the split partition is operational or it crashed. In this case, the crashed partition leads to the loss of its share of account balances. To enable crash tolerance, a crash failure detector [9], [8] may need to be incorporated in the design.

A robust blockchain needs to be tolerant to Byzantine faults [41] where affected peers may behave arbitrarily. Byzantine peers may compromise agreement on the seed block or on the split itself. We believe that our proposed algorithm may be made tolerant to such faults. However, a definitive study is left for future research.

REFERENCES

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system,"," 2008.

[2] G. Wood, "Ethereum: A secure decentralized generalized transaction ledger," *Ethereum project yellow paper*, vol. 151, pp. 1–32, 2014.

[3] M. Castro and B. Liskov, "Practical byzantine fault tolerance and proactive recovery," *ACM Trans. Comput. Syst.*, vol. 20, no. 4, pp. 398–461, Nov. 2002. [Online]. Available: http://doi.acm.org/10.1145/571637.571640

[4] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 3, p. 382–401, Jul. 1982. [Online]. Available: https://doi.org/10.1145/357172.357176

[5] 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.

[6] A. Kirilenko, A. Kyle, M. Samadi, and T. Tuzun, "The flash crash: High-frequency trading in an electronic market," *The Journal of Finance*, vol. 72, no. 3, pp. 967–998, 2017.

[7] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *J. ACM*, vol. 32, no. 2, p. 374–382, Apr. 1985. [Online]. Available: https://doi.org/10.1145/3149.214121

[8] T. D. Chandra, V. Hadzilacos, and S. Toueg, "The weakest failure detector for solving consensus," *Journal of the ACM (JACM)*, vol. 43, no. 4, pp. 685–722, 1996.

[9] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *Journal of the ACM (JACM)*, vol. 43, no. 2, pp. 225–267, 1996.

[10] I. Abraham, S. Devadas, K. Nayak, and L. Ren, "Brief announcement: Practical synchronous byzantine consensus," in *DISC*, 2017, pp. 41:1–41:4. [Online]. Available: https://doi.org/10.4230/LIPIcs.DISC.2017.41

[11] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich *et al.*, "Hyperledger fabric: a distributed operating system for permissioned blockchains," in *Proceedings of the Thirteenth EuroSys Conference*, 2018, pp. 1–15.

[12] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich, "Algorand: Scaling byzantine agreements for cryptocurrencies," in *OSDI*, 2017, pp. 51–68.

[13] M. Ben-Or, B. Kelmer, and T. Rabin, "Asynchronous secure computations with optimal resilience," in *Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing*, 1994, pp. 183–192.

[14] C. Cachin, K. Kursawe, F. Petzold, and V. Shoup, "Secure and efficient asynchronous broadcast protocols," in *Annual International Cryptology Conference*. Springer, 2001, pp. 524–541.

[15] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song, "The honey badger of bft protocols," in *CCS*. New York, NY, USA: ACM, 2016, pp. 31–42. [Online]. Available: http://doi.acm.org/10.1145/2976749.2978399

[16] Y. Lu, Z. Lu, Q. Tang, and G. Wang, "Dumbo-mvba: Optimal multi-valued validated asynchronous byzantine agreement, revisited," in *Proceedings of the 39th Symposium on Principles of Distributed Computing*, 2020, pp. 129–138.

[17] B. Guo, Z. Lu, Q. Tang, J. Xu, and Z. Zhang, "Dumbo: Faster asynchronous bft protocols," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 803–818.

[18] I. Bentov, R. Pass, and E. Shi, "Snow white: Provably secure proofs of stake." *IACR Cryptology ePrint Archive*, vol. 2016, no. 919, 2016.

[19] I. Eyal, A. E. Gencer, E. G. Sirer, and R. Van Renesse, "Bitcoin-ng: A scalable blockchain protocol," in *NSDI*, 2016, pp. 45–59.

[20] A. Kiayias, A. Russell, B. David, and R. Oliynykov, "Ouroboros: A provably secure proof-of-stake blockchain protocol," in *Annual International Cryptology Conference*. Springer, 2017, pp. 357–388.

[21] R. Pass and E. Shi, "Fruitchains: A fair blockchain," in *Proceedings of the ACM Symposium on Principles of Distributed Computing*, 2017, pp. 315–324.

[22] I. Abraham, D. Malkhi, K. Nayak, L. Ren, and A. Spiegelman, "Solida: A blockchain protocol based on reconfigurable byzantine consensus," *arXiv preprint arXiv:1612.02916*, 2016.

[23] R. Pass and E. Shi, "Hybrid consensus: Efficient consensus in the permissionless model," in *DISC*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[24] C. Decker, J. Seidel, and R. Wattenhofer, "Bitcoin meets strong consistency," in *Proceedings of the 17th International Conference on Distributed Computing and Networking*, 2016, pp. 1–10.

[25] L. Luu, V. Narayanan, C. Zheng, K. Baweja, S. Gilbert, and P. Saxena, "A secure sharding protocol for open blockchains," in *CCS*, 2016, pp. 17–30.

[26] R. Pass and E. Shi, "Thunderella: Blockchains with optimistic instant confirmation," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2018, pp. 3–33.

[27] M. Zamani, M. Movahedi, and M. Raykova, "Rapidchain: A fast blockchain protocol via full sharding." *IACR Cryptology ePrint Archive*, vol. 2018, p. 460, 2018.

[28] S. Popov, "The tangle," The IOTA Foundation, Tech. Rep., April 2018, https://iota.org/IOTA_Whitepaper.pdf.

[29] Y. Sompolinsky and A. Zohar, "Secure high-rate transaction processing in bitcoin," in *International Conference on Financial Cryptography and Data Security*, 2015, pp. 507–527.

[30] Y. Xiao, N. Zhang, W. Lou, and Y. T. Hou, "A survey of distributed consensus protocols for blockchain networks," 2019.

[31] R. Friedman and A. Vaysburd, "Fast replicated state machines over partitionable networks," in *Proceedings of SRDS'97: 16th IEEE Symposium on Reliable Distributed Systems*. IEEE, 1997, pp. 130–137.

[32] M. K. Aguilera, W. Chen, and S. Toueg, "Using the heartbeat failure detector for quiescent reliable communication and consensus in partitionable networks," *Theor. Comput. Sci.*, vol. 220, no. 1, p. 3–30, 1999. [Online]. Available: https://doi.org/10.1016/S0304-3975(98)00235-7

[33] D. Dolev, R. Friedman, I. Keidar, and D. Malkhi, "Failure detectors in omission failure environments," in *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '97. New York, NY, USA: Association for Computing Machinery, 1997, p. 286. [Online]. Available: https://doi.org/10.1145/259380.259501

[34] Ö. Babaoğlu, R. Davoli, and A. Montresor, "Group membership and view synchrony in partitionable asynchronous distributed systems: Specifications," *ACM SIGOPS Operating Systems Review*, vol. 31, no. 2, pp. 11–22, 1997.

[35] T. D. Chandra, V. Hadzilacos, S. Toueg, and B. Charron-Bost, "On the impossibility of group membership," in *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, 1996, pp. 322–330.

[36] G. V. Chockler, I. Keidar, and R. Vitenberg, "Group communication specifications: a comprehensive study," *ACM Computing Surveys (CSUR)*, vol. 33, no. 4, pp. 427–469, 2001.

[37] J. Tran, G. Ramachandran, P. Shah, C. Danilov, R. Santiago, and B. Krishnamachari, "Swarmdag: A partition tolerant distributed ledger protocol for swarm robotics," in *Ledger*, 2019.

[38] B. Alpern and F. Schneider, "Defining liveness," *Information processing letters*, vol. 21, no. 4, pp. 181–185, 1985.

[39] M. N. Kendric Hood, Joseph Oglio and G. Sharma, "Partitionable asynchronous cryptocurrency blockchain, ArXiv 3523102," December 2020.

[40] https://github.com/khood5/distributed-consensus-abstract-simulator.git.

[41] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," in *Concurrency: the Works of Leslie Lamport*, 2019, pp. 203–226.