Algorand Protocol Description

Argimiro, CoinFabrik

December 22, 2022

Contents

1 Introduction

We aim to describe how the Algorand protocol works. We used [GHM+17], [Mic16] and Algorand's official specifications as our main references. In the following document we present a detailed pseudocode interpretation of the program run by an Algorand node with at least one online (i.e., participating) account. We attempt to explain the 5 main stages of the code supporting the protocol in a clear and concise way. This work aims to provide a basis for an Algorand blockchain simulator currently in development.

Global parameters:

- R = sortition seed renewal rate (in number of rounds). Set to 2 in the specs. as of December 2022
- MAX_STEPS = maximum number of allowed steps in the main algorithm. Defined as 255 in the specs.
- τ_{step} = expected number of members on a regular step committee
- τ_{final} = expected number of members on a final consensus achieving committee
- T_{step} = fraction of expected members for a voting committee on a given step
- T_{final} = fraction of expected members for a voting committee on the final step (step = 2). If observed, achieves final consensus on the given block hash
- $\lambda_{proposal}$ = time interval for the node to accept block proposals, after which it chooses the observed block with the highest priority (lowest hash)
- λ_{block} = waiting time for the full block to be received once decided by vote. If no full block is received, the node falls back to the empty block.

- SL = the account balances lookback interval, in number of rounds (integer). Set to 320 in specs.

On top of that, each node may run in favour of different accounts. To do this, it needs the following parameters. Let $A = \{a\}$ be a collection of accounts a.

Account parameters:

- Public keys a_{pk} for every account linked to the node (doubles as their respective addresses)
- Public participation keys $a_{p\_partkey}$ for all accounts registered as online (private participation keys are kept securely by respective users and should not be directly accesible by the node)
- Two levels of ephemeral keys, signed according to the signing scheme described in specs. (where first level keys are signed with the participation keys, and second level (aka. leaf) ephemeral keys are signed with first level ephemeral keys). These keys are used for actual participation in the consensus (abstracted as a_{sk} in the following code) They are ephemeral because they live for a single round, after which they are deleted
- A balance table $BalanceTable$ for all accounts linked to the node, for the round $SL + R$ rounds before the one currently running

2 Main algorithm

Assume right now that a given node is fixed, we are at round r, this node has access to a set of accounts $a \in A$. Moreover, the account owner has computed for each round a round leaf ephemeral key a_{sk}. Assume also the node has access to contextual information about the ledger state, that being the current last confirmed block, the last $n > 322$ confirmed blocks, balance and ephemeral keys of all online accounts in that interval of rounds. For the sake of clarity and readability, we abstract access to that data through a context structure (ctx in code), accesible at all times by the node. The main algorithm goes over the following stages on every round.

- block creation: each node may assemble a block, which includes transactions and metadata.
- block proposal: each block is going to run the VRF algorithm to decide whether it should participate in the next stage. If it gets selected, then it is going to assemble and then send the block proposal to the other nodes.
- soft vote: each node will undergo two rounds of voting to reduce all posible block options to a dicotomy: either a block proposed by a chosen online network user or an empty block.
- certify vote: each node undergoes a series of n voting rounds ($253 \geq n \geq 1$) to certify either the block received by the prior stage or an empty block. If $n = 1$, the consensus is said to be FINAL. Otherwise, the consensus will be TENTATIVE.
- block confirmation: the block chosen in the certification stage is added to the ledger, as FINAL or TENTATIVE depending on the ammount of users observing it as either (as defined by the threshold parameters for each kind of consensus).

The following algorithm outlines the process of running a round for a given node:

Algorithm 1 Main node algorithm

```
1: function NodeMain()
2:     asmBlock ← BlockCreation(transactionPool)
3:     propBlockHash ← BlockProposal(asmBlock, r, accounts A)
4:     SVBlockHash ← SoftVote(propBlockHash)
5:     CVBlockHash ← CertifyVote(SVBlockHash)
6:     BlockConfirmation(CVBlockHash)
```

It starts by assembling the block, going through the proposal process, running both voting stages, and by the time it finishes either a block has been confirmed and added to the ledger (as TENTATIVE or FINAL), or it gets stuck awaiting for the recovery protocol to be started by the network.

2.1 Block Creation

Algorithm 2 Block creation

```
1: function BlockCreation()(transactionPool)
2:     Pick transactions from transactionPool
3:     return Block
```

Arguments:

- The local pending transaction pool, which should be accesible to the node

Description:
The Algorand protocol specs ([GHM+17]) do not ensure a specific way or internal order in which transactions are added to the block. More research is needed on how block data is put together. Most of the block metadata is added in this stage, with the exception of the seed which is computed during the block proposal stage (2.2)

Returns: A full block, consisting of a variable number of transactions, and the following metadata:

- round r
- genesis identifier
- upgrade vote
- timestamp
- seed
- reward updates
- crytographic commitment to txn sequence
- crytographic commitment to txn sequence SHA256
- previous block hash
- txn counter
- newly expired participation keys

2.2 Block Proposal

Algorithm 3 Block proposal

function $BlockProposal(Block\ B,\ r,\ Accounts\ A)$
  $ProposalMsgsSent = \{\}$ //Set of proposal messages sent by this node
  for $a \in A$ **do**
    $\langle sorthash, \pi, j\rangle \leftarrow \mathsf{Sortition}(a_{sk}, \mathsf{getSortitionSeed}(ctx, r), t, role,$
      $\mathsf{getSortition}_w(ctx, r, a_{pk}), \mathsf{getSortition}_\mathsf{W}(ctx, r))$
    if $j > 0$ **then**
      $priority \leftarrow \min\{\mathsf{Hash}(sorthash||n) : 1 \leq n \leq j\}$
      $msg \leftarrow \mathsf{MSG}(a.pk, \mathsf{Signed}_{a.sk}(priority, \langle sorthash, \pi\rangle))$
      $ProposalMsgsSent \leftarrow ProposalMsgsSent \cup msg$
      $\mathsf{GOSSIP\_MSG}(a.pk, Signeda.sk(priority, \langle sorthash, \pi\rangle))$

  while $elapsed\ time < \lambda_{proposal}$ **do**
    $incomingMsgs.push(listen())$
  $msgs \leftarrow incomingMsgs[r, step = 0]$
  $lowestHashMsg \leftarrow m \| m.priority = \min\{m.priority \in incomingMsgs : \forall m$ s.t.
      $\mathsf{verifySortition}(m.sorthash, m.pi, m.pk) \& \mathsf{ValidateMsg}(m)\}$

  $FullBlockToPropose \leftarrow empty\_block(r)$
  if $lowestHashMsg \in msgsSent$ **then**
    $FullBlockToPropose \leftarrow B$
    $FullBlockToPropose.seed \leftarrow \mathsf{ComputeSeed}(ctx, r, B)$
    $\mathsf{GOSSIP\_MSG}(a_{pk}, \mathsf{Signed}_{a_{sk}}(B, \langle sorthash, \pi\rangle))$
  else
    while $elapsed\ time < \lambda_{block}$ **do**
      $incomingMsgs.push(listen())$
      $m \leftarrow incomingMsgs.back()$
      if $\mathsf{ValidateMsg}(m) \& lowestHashMsg.blockHash = \mathsf{Hash}(m.FullBlock)$ **then**
        $FullBlockToPropose \leftarrow m.FullBlock$

  return $H(FullBlockToPropose)$

Arguments:

- A block B received from the previous stage
- The number of round currently executing, r
- A list of online accounts $A = \{a\}$ associated with this node
- ephemeral keys
- public keys associated to these accounts

Description:
The **block proposal** stage (step 0 on specs.) is the first stage of the consensus algorithm. First, the node loops through all their online accounts ($a \in A$), running the sortition subroutine (sortition()). Functions Sortition(), getSortitionSeed(), $\text{sortition}_{\text{w}}()$ and $\text{sortition}_{\text{W}}()$ described in Section 3. The sortition subroutine determines if an account a is selected to *sorthash*, a proof of the sortition π, and an integer $j \geq 0$ that is used to prioritize votes.

If $j > 0$ then the address a gets to vote and proceeds as follows. For this account a and this triplet $\langle sorthash, \pi, j \rangle$ it computes the hashes $\{\text{Hash}(sorthash \| n) : 1 \leq n \leq j\}$ and keeps the minimum (line 6), which we set as the *priority* of this account for this round (here Hash() is a cryptographic hash function, assumed to be a random oracle). The GOSSIP_MSG() call which happens next, gossips (broadcasts) the priority, sortition hash and proof to other nodes. It further includes the public key a_{pk} for the account a and the signature of the triplet $(priority, \langle sorthash, \pi \rangle)$ with the leaf ephemeral key associated with the account a and the round r.

In the lines 11 to 15, the algorithm accepts block proposals (in the same format as gossiped) from other accounts until $\lambda_{proposal}$ is reached. For each block proposal received it only keeps the one with the lowest priority validating the block before accepting it as a possible proposal.

To validate a block proposal, the node verifies the sortition proof and hash (utilizing the verifySortition() function). If the vote message was invalid, the node can decide to flag the proposing user as malicious and choose to ignore any further messages received from that public address.

Finally, in lines 16-25, the algorithm will select a block as follows. Assume the elapsed time has been reached. If the selected proposal came from this node, the seed is computed calling ComputeSeed() from the input block B, the context helper structure ctx and the round number r, and the complete block is gossiped to the network.

However, if the proposal did not come from this node, then the node waits until λ_{block} to receive the full block from other nodes in the network. In case of timeout in this last stage, it falls back to the empty block.

Returns:

- The hash of a full block $BlockHash$, whether a block proposed by a user (including users whose accounts are linked to this node) or an empty block in any of the special cases mentioned above. It contains all transactions and metadata added in block creation (2.1), as well as the computed seed.

2.3 Soft Vote

The soft vote stage (step 1, called reduction in [GHM+17]) aims to reduce any amount of potentially conflicting proposed blocks in the stage prior into a binary choice, either a block proposed by a user or an empty one.

Algorithm 4 Soft Vote

function $SoftVote(BlockHash)$
  $state \leftarrow 1$
  CommitteeVote$(ctx, r, step, \tau_{step}, BlockHash)$
  $hblock \leftarrow$ CountVotes$(ctx, r, step, T_{step}, \tau_{step}, \lambda_{block} + \lambda_{step})$
  $empty\_hash \leftarrow$ Hash$(Empty(r,$ Hash$(ctx.last_block)))$
  if $hblock = TIMEOUT$ **then**
    CommitteeVote$(ctx, r, step, \tau_{step}, empty_hash)$
  else
    CommitteeVote$(ctx, r, step, \tau_{step}, hblock)$
$hblock \leftarrow$ CountVotes$(ctx, r, step, T_{step}, \tau_{step}, \lambda_{step})$
if $hblock = TIMEOUT$ **then return** $empty\_hash$ **else return** $hblock$

Arguments:

- A block hash, $BlockHash$, received from the previous stage (Sect 2.2).

Description:
In the first step and with the input block *block hash* as input, together with other parameters, the node does a preliminary vote count (with CommitteeVote()) and counts votes after λ seconds have passed using the subroutine CountVotes() both subroutines being described in Section 3 in more detail.

In the second step, committee members vote for the hash that received at least the ammount of votes set by the threshold $\tau_{step} * T_{step}$ for this step, or the hash of the default empty block if no hash was observed to receive enough votes. Here, TIMEOUT is a special value to signify that the vote count could not observe a threshold surpassing value in the required time window.
Returns:

- A block hash, chosen by a majority of users in the soft vote / reduction stage (could be the empty block hash)

2.4 Vote Certification

Algorithm 5 CertifyVote

function CertifyVote$(blockHash)$
  $localBlockHash \leftarrow blockHash$
  $step \leftarrow 2$
  while $step < 256$ **do**

    $\langle blockHash, bConfirmed \rangle \leftarrow$ BLOCK_STEP$(blockHash, localBlockHash, step)$
    if $bConfirmed$ **then return** $blockHash$
    $step++$

    $\langle blockHash, bConfirmed \rangle \leftarrow$ EMPTY_STEP$(blockHash, step)$
    if $bConfirmed$ **then return** $blockHash$
    $step++$

    CommonCoinFlipVote$(blockHash, step)$
    $step++$

  while $True$ **do**
    NOP //await asynchronous recovery

Arguments:

- A block hash, $BlockHash$, received from the previous stage (soft vote)

Description:
The certify vote stage is the core of the voting algorithm. The main loop executes a maximum of 254 times (from step 2 to step 255, called "cert" and "down" respectively on the specs.). This is to avoid an attacker indefinitely postponing consensus in a compromised network, giving them a small chance to force consensus on their desired block with each iteration.

The CertifyVote algorithm comprises three important procedures:

1. the block step, where, if selected as a committee member, votes for the block selected in the soft voting stage and then attempts to confirm it by listening to other cast votes until a threshold is met. If this step times out and no consensus is reached, it moves on to the empty step.

2. Very similar in nature to the previous one, in this procedure a vote is cast for the value in *blockHash* and then it attempts to confirm an empty block.

3. if no consensus is reached on the empty block, it arrives to the common coin procedure, where it attempts another vote and vote count for the empty block. If here, no consensus is reached, it uses the pseudo-randomness of the VRF hash output to "flip a coin" by looking at the last bit of the lowest sortition hash owned by observed committee members. This makes it so even though the coin flip is random, most users will observe the same outcome and will in turn set their *blockHash* values according to the observed bit. In turn, this prevents an adversary from being able to easily predict beforehand what the next vote of a user would be if they were a committee user, and therefore the attacker cannot force consensus sending their vote messages to targeted users in a scenario where the voting is split pretty evenly but no block has quite enough votes to push the threshold.

If no consensus is reached before the step counter reaches *MAX_STEPS*, the program stops in an infinite loop and awaits for the asynchronous recovery protocol to kick in.

Returns:

- A block hash certified by a majority of users, to be confirmed and added to the ledger in the next stage as TENTATIVE or FINAL depending on the observed voting share

2.5 Block Confirmation

Algorithm 6 BlockConfirmation

function $BlockConfirmation(blockHash)$
  $r \leftarrow \mathsf{CountVotes}(ctx, round, FINAL, T_{final}, \tau_{final}, lstep)$
  if $r = blockHash$ **then**
    return $\langle FINAL, BlockOfHash(blockHash)\rangle$
  else
    return $\langle TENTATIVE, BlockOfHash(blockHash)\rangle$

Arguments:

- A block hash, $BlockHash$ received from the previous stage (Sect. 2.4)

Description:
Finally, the proposed block (be it an empty block or a specific one proposed by a designated user) is confirmed and added to the ledger. We introduce the notions of FINAL and TENTATIVE consensus. Specifically, if a block is confirmed in the very first step of the certify vote stage, its

vote is cast as FINAL. If enough users reproduce this behavior, the vote count on line 2 for votes cast in this fashion is over the required threshold, and the consensus is labeled as FINAL. This means that the block is simultaneously added to the ledger and confirmed, as well as immediately confirming any prior blocks that could have been labeled TENTATIVE. If this node is not able to ensure that enough users considered the block represented by hblock as FINAL, the consensus is labeled as TENTATIVE, and it may be confirmed by a later FINAL block. The BlockOfHash() function outputs the full block for the provided hash, either by returning a locally available copy or by requesting it from other users.

Returns:

- A full block to be added to the ledger, along with a flag indicating if the consensus achieved is TENTATIVE or FINAL in nature

3 Subroutines

Algorithm 7 CountVotes

```
function CountVotes(ctx, round, step, T, τ, λ)
    startTime ← currentTime()
    counts ← {} // hash table, new keys mapped to 0
    voters ← {}
    msgs ← incomingMsgs[round, step]
    while TRUE do
        m ← msgs.next()
        if m = ⊥ then
            if ElapsedTime > startTime + λ then
                return TIMEOUT
        else
            ⟨votes, value, sorthash⟩ ← ProcessMsg(ctx, τ, m)
            if m.pk ∈ voters ∨ votes = 0 then continue
            voters ∪ = {m.pk}
            counts[value] + = votes
            if counts[value] ≥ T ∗ τ then return value
```

Arguments:

- ctx = a helper structure to retrieve ledger context information (e.g. the last confirmed block)
- $round$ = current round number
- $step$ = current step number
- T = fraction of expected committee members to get the confirmation threshold
- τ = expected number of committee members
- λ = maximum time to keep running the vote count and waiting for network messages

Description:
This subroutine counts the votes for any observed block hash value, for a given round and step numbers. It returns as soon as it finds a value exceeding the specified threshold (which will also vary according to round and step). If no value is observed to have the required amount of votes in the predetermined temporal window, it finishes with a TIMEOUT. It is important to notice that every committee user's vote is processed at most once for the given context.

Messages m are structured .... The function ProcessMsg returns....

completar que es un msj, que es mpk, etc

This subroutine counts the votes for any observed block hash value, for a given round and step numbers. It returns as soon as it finds a value exceeding the specified threshold (which will also vary according to round and step). If no value is observed to have the required amount of votes in the predetermined temporal window, it finishes with a TIMEOUT. It is important to notice that every committee user's vote is processed at most once for the given context.

Returns:

- the first block hash value observed to achieve the required threshold, or a TIMEOUT constant if none was found in the required time window.

Algorithm 8 CommitteeVote

function CommitteeVote($ctx, round, step, T, \tau, \lambda$)
  $role \leftarrow \langle committee, round, step \rangle$
  $\langle sorthash, \pi, j \rangle \leftarrow$ Sortition$(user.sk, ctx.seed, \tau, role, ctx.weight[user.pk], ctx.W)$
// only committee members originate a message
  if $j > 0$ **then**
    GOSSIP_MSG$(user.pk,$ Signed$_{user.sk}(round, step, sorthash, \pi,$ Hash$(ctx.last_{block}), value))$

Arguments:

- ctx = a helper structure to retrieve ledger context information (e.g. the last confirmed block)
- $round$ = current round number
- $step$ = current step number
- T = fraction of expected committee members to get the confirmation threshold
- τ = expected number of committee members
- λ = maximum time to keep running the vote count and awaiting for network messages

Description:
This subroutine verifies that the caller is a member of the specified committee (for a given round and step number), and in that case casts a vote for the block passed as argument.

Returns:
The subroutine has no return value, it only gossips the specified vote if the caller is a committee member.

Algorithm 9 ComputeSeed

function $ComputeSeed(ctx, r, B)$
  if $B \neq empty\_block$ **then**
    return $VRF_{get_{SK_a}(ctx,r)}(ctx.LastBlock.seed||r)$
  else
    return Hash$(ctx.LastBlock.seed||r)$

Arguments:

- ctx = a helper structure to retrieve ledger context information (e.g. the last confirmed block)
- r = current round number
- B = the block whose seed is being computed

Description:
This subroutine computes the required sortition seed for the given round number, which goes in the proposed block's metadata. If the block is empty, the seed is a hash of the previous block's seed. The $get_{SK_a}(ctx, r)$ helper function gets the relevant account's secret ephemeral keys (according to the signing scheme described in specs, the keys 160 rounds prior to r). This roughly corresponds to the secret key from a round b time before block $r - 1 - (r \ mod \ R)$, where R is the sortition seed's renewal rate, r is the current round's number, and b is the upper bound for the maximum ammount of time that the network might be compromised.

Returns:

- the computed seed for the given block, ledger context and round

Algorithm 10 getSortitionSeed

function $getSortitionSeed(ctx, r, a_{pk})$
return $ctx.block[r - 1 - (r \ mod \ R)].seed$

Arguments:

- ctx = a helper structure to retrieve ledger context information (e.g. the last confirmed block)
- $round$ = current round number
- a_{pk} = the account's public key for the look up table

Description:
This helper function gets the relevant sortition seed for the current round r, according to the seed lookback parameter R. Conceptually, it corresponds with the seed computed R rounds prior to r, refreshed every R rounds.
Returns:

- a sortition seed to be used in the round r

Algorithm 11 getSortitionWeight

function $getSortition_w(ctx, round, a_{pk})$
return $ctx.balanceTable[r - (R + SL)][a_{pk}]$

Arguments:

- ctx = a helper structure to retrieve ledger context information (e.g. the last confirmed block)
- r = current round number
- a_{pk} = the account's public key for the look up table

Description:
This helper function retrieves the stake $R + SL$ rounds prior to r, for an account with public key a_{pk}
Returns:

- the relevant account's stake

Algorithm 12 getSortitionTotalStake

function $getSortition_W(ctx, r)$
  return $\sum_{a_{pk} \in ctx.balanceTable[r-(R+SL)]} balanceTable[r-(R+SL)][a_{pk}]$

Arguments:

- ctx = a helper structure to retrieve ledger context information (e.g. the last confirmed block)
- r = current round number

Description:
This helper function returns the sum of all stake for $R + SL$ rounds prior to r.
Returns:

- the total stake at play in the relevant round (according to lookback parameters)

Algorithm 13 Sortition

function $\mathsf{Sortition}(sk, seed, \tau, role, w, W)$
  $\langle hash, \pi \rangle \leftarrow \mathsf{VRF}(seed||role)$
  $p \leftarrow \frac{t}{W}$
  $j \leftarrow 0$
  while $\frac{hash}{2^{hashlen}} \notin [\sum_{k=0}^{j} \mathsf{B}(k; w, p), \sum_{k=0}^{j+1} \mathsf{B}(k; w, p))$ **do**
    $j++$
  return $\langle hash, \pi, j \rangle$

Arguments:

- sk = a user's secret key (an ephemeral key for the given round, according to key specs)
- $seed$ = the sortition seed to be used
- τ = the expected committee size for the given role
- $role$ = a flag specifying the role for the sortition to take place (e.g. block proposer)
- w = the user's weight (i.e., its relevant stake)
- W = the total relevant stake for the given round

Description:
The Sortition procedure is one of the most important subroutines in the main algorithm, as it is used in multiple stages and contexts. Generally, it manages to pseudo-randomly but verifiably (through the use of a Verifiable Random Function) select a user with probability proportional to their weight (i.e., stake) by returning a j parameter, which indicates the number of times that specific user was chosen. Algorithmically, every monetary unit the user has is considered a "sub-user", and then each one of them is selected with probability $p = \frac{\tau}{W}$, where τ is the expected amount of users to be selected for the given role. The semi-open interval $[0, 1)$ is then split into consecutive intervals using an accumulated binomial distribution, and wherever the fraction $\frac{hash}{2^{hashlen}}$ belongs to the interval, that's the ammount of chosen sub-users for the subroutine caller.

Returns:

- an integer j that will be positive (larger than 0) if the user has been selected, and its size corresponds to the amount of sub-users for a given committee member

Algorithm 14 VerifySortition

function VerifySortition($pk, seed, \tau, role, w, W$)
  if $\neg$VerifyVRF$_{pk}(hash, \pi, seed||role)$ **then return**0
  $p \leftarrow \frac{t}{W}$
  $j \leftarrow 0$
  while $\frac{hash}{2^{hashlen}} \notin [\sum_{k=0}^{j} \mathsf{B}(k; w, p), \sum_{k=0}^{j+1} \mathsf{B}(k; w, p))$ **do**
    $j++$
  return j

Arguments:

- pk = a user's public key (their address)
- $seed$ = the sortition seed to be used
- τ = the expected committee size for the given role
- $role$ = a flag specifying the role for the sortition to take place (e.g. block proposer)
- w = the user's weight (i.e., its relevant stake)
- W = the total relevant stake for the given round

Description:
The sortition verification procedure takes Sortition's output and utilizes VRF properties to verify the validity of said output. Once the check is passed, it repeats Sortition's sub-user selection procedure, and outputs the amount of times the user was chosen according to their respective observed stake.

Returns:

- an integer j that will be positive (larger than 0) if the user has been selected, and it's size corresponds to the amount of sub-users for a given committee member

Algorithm 15 BLOCK_STEP

function BLOCK_STEP($hblock, blockHash, step$)
  $bConfirmed \leftarrow FALSE$
  $hb \leftarrow hblock$
  CommitteeVote$(ctx, round, step, \tau_{step}, hb)$
  $hb \leftarrow$ CountVotes$(ctx, round, step, T_{step}, \tau_{step}, \lambda_{step})$
  if $hb = TIMEOUT$ **then**
    $hb \leftarrow blockHash$
  else if $hb \neq emptyHash$ **then**
    for $step < s' \leq step + 3$ **do**
      CommitteeVote$(ctx, round, s', \tau_{step}, hb)$
    if $step = 1$ **then**
      CommitteeVote$(ctx, round, FINAL, \tau_{final}, hb)$
    $bConfirmed \leftarrow TRUE$
  return $\langle hb, bConfirmed \rangle$

Arguments:

- $hblock$ = a block hash value to be voted on
- $blockHash$ = the block hash value locally observed by the node to be the winning choice

- *step* = currently executing step

Description:
The BLOCK_STEP subroutine handles the first part of the certifyVote's main loop. It first casts a vote on the observed hblock value possibly coming off of a coin toss vote (see commonCoinFlipVote). It then attempts to count incoming votes. In case of a timeout (i.e., no block passed the threshold), it returns the block hash. Otherwise, and assuming the highest voted value is not an empty block hash, it casts votes for the next three steps for the block seen as selected. Then, in the special scenario where we are in step 1, we cast our vote as FINAL, suggesting that this block is a candidate for a final consensus (if enough committee members observe this fact). Finally, the procedure returns the selected block and a flag to either continue or return from the main procedure with a certified vote.
Returns:

- a tuple $\langle hb, bConfirmed \rangle$, comprised of a block hash value and a boolean flag to assert whether the given value was observed to be confirmed in any way or if further steps are needed

Algorithm 16 EMPTY_STEP

function EMPTY_STEP($hblock, step$)
  $bConfirmed \leftarrow FALSE$
  $hb \leftarrow hblock$
  CommitteeVote($ctx, round, step, \tau_{step}, hb$)
  $hb \leftarrow$ CountVotes($ctx, round, step, T_{step}, \tau_{step}, \lambda_{step}$)
  if $hb = TIMEOUT$ **then**
    $hb \leftarrow emptyHash$
  else if $hb = emptyHash$ **then**
    for $step < s' \leq step + 3$ **do**
      CommitteeVote($ctx, round, s', \tau_{step}, hb$)
    $bConfirmed \leftarrow TRUE$
  return $\langle hb, bConfirmed \rangle$

Arguments:

- *hblock* = a block hash value to be voted on
- *step* = currently executing step

Description:
The EMPTY_STEP procedure works in a very similar fashion to BLOCK_STEP, with two main differences: the block to attempt to confirm is the empty block, and there is no final consensus achievable in this scenario.
Returns:

- a tuple $\langle hb, bConfirmed \rangle$, comprised of a block hash value and a boolean flag to assert whether the given value was observed to be confirmed in any way or if further steps are needed

Algorithm 17 CommonCoinFlipVote

```
function CommonCoinFlipVote(hblock, blockHash, step)
    hb ← hblock
    CommitteeVote(ctx, round, step, τ_step, hb)
    hb ← CountVotes(ctx, round, step, T_step, τ_step, λ_step)
    if hb = TIMEOUT then
        if CommonCoin(ctx, round, step, T_step, τ_step, λ_step) = 0 then
            hb ← blockHash
        else
            hb ← emptyHash
```

Arguments:

- $hblock$ = a block hash value to be voted on
- $blockHash$ = the block hash value locally observed by the node to be the winning choice
- $step$ = currently executing step

Description:
This procedure casts a vote for the previous step's selected block, then does a vote count, and finally flips a coin (last bit of lowest sortition hashed user, see CommonCoin) that's random, but consensuated between a majority of users. This guarantees that an attacker could not guess what the next vote will be for a given user, and introduces a tool to facilitate reaching a tentative consensus in the next stages.
Returns:

- a block hash value, decided randomly in between the locally chosen block hash or an empty hash, to be voted on in the immediately following step

Algorithm 18 CommonCoin

```
function CommonCoin(ctx, round, step, τ)
    minhash ← 2^hashlen
    for m ∈ incomingMsgs[round, step] do
        ⟨votes, value, sorthash⟩ ← ProcessMsg(ctx, τ, m)
        for 1 ≤ j < votes do
            h ← Hash(sorthash||j)
            if h < minhash then minhash ← h
    return minhash mod 2
```

Arguments:

- ctx = a helper structure to retrieve ledger context information (e.g. the last confirmed block)
- $round$ = current round number
- $step$ = current step number
- τ = the expected committee size for the given role

Description:

The CommonCoin procedure takes advantage of the randomness of sortition hashes, and attempts to get a consensus random bit (a coin toss) getting the least significant bit of the lowest hashed sortition hash concatenated with sub-user index, for all committee voters in the given round and step. Since sortition hashes are random, even if an attacker happened to be a committee member

with the lowest observable hash, the coin value's randomness would still be preserved, and in successive steps the probability of them being chosen again and able to manipulate this stage would severely diminish.

Returns:

- a random bit (either 0 or 1), observed to be the least significant bit of the hashed sortition hash of a committee member for the given step, that should be observed by a majority of users to be the same (conceptually, a consensus coin toss)

References

[GHM+17] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 51–68. ACM, 2017.

[Mic16] Silvio Micali. ALGORAND: the efficient and democratic ledger. *CoRR*, abs/1607.01341, 2016.