

# Algorand Protocol Description

Argimiro, CoinFabrik

December 22, 2022

## Contents

### 1 Introduction

We aim to describe how the Algorand protocol works. We used [?], [?] and the code 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 (aka. 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.
- $\tau_{step}$  = expected number of members on a regular step committee
- $\tau_{final}$  = expected number of members on a final consensus achieving committee
- $T_{step}$  = fraction of expected members for a voting committee on a given step
- $T_{step}$  = 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)
- $\lambda_{block}$  = 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.

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:**

- A pair of public and private keys  $a_{pk}, a_{sk}$  (although the secret key may be unnecessary)
- ephemeral keys....

### 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  $a_{sk}$ . The main algorithm goes over the following steps 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.

## 2.1 Block Creation

---

### Algorithm 1 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 ([?]) 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. Block metadata is added later (see below ?? for more details).

**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
- cryptographic commitment to txn sequence
- cryptographic commitment to txn sequence SHA256
- previous block hash
- txn counter
- newly expired participation keys

## 2.2 Block Proposal

1. Que  
hace la  
linea 9. No  
entiendo. 2.  
que es ctx ?  
3. Cual es la  
estructura  
del  
FullBlockToProp  
e.g, (B,  
seed,...)

---

**Algorithm 2** Block proposal

---

```
1: function BlockProposal(Block B, Accounts A)
2:   ProposalMsgsSent = {} //Set of proposal messages sent by this node
3:   for  $a \in A$  do
4:      $\langle \text{sorthash}, \pi, j \rangle \leftarrow \text{Sortition}(a_{sk}, \text{getSortitionSeed}(\text{round}), t, \text{role},$ 
        $\text{sortition}_w(a, \text{round}), \text{sortitionW}(\text{round}))$ 
5:     if  $j > 0$  then
6:        $\text{priority} \leftarrow \min\{\text{Hash}(\text{sorthash}||n) : 1 \leq n \leq j\}$ 
7:        $\text{UserPriorityHashTable}[a] = \text{priority}$ 
8:        $\text{msg} \leftarrow \text{MSG}(a.pk, \text{Signed}_{a.sk}(\text{priority}, \langle \text{sorthash}, \pi \rangle))$ 
9:        $\text{ProposalMsgsSent} \cup = \text{msg}$ 
10:       $\text{GOSSIP\_MSG}(a.pk, \text{Signed}_{a.sk}(\text{priority}, \langle \text{sorthash}, \pi \rangle))$ 
11:   while elapsed time < proposal time do
12:     incomingMsgs.push(listen())
13:      $\text{msgs} \leftarrow \text{incomingMsgs}[\text{round}, \text{step} = 0]$ 
14:      $\text{lowestHashMsg} \leftarrow m \parallel m.\text{priority} = \min\{m.\text{priority} \in \text{incomingMsgs} : \forall m \text{ s.t.}$ 
        $\text{verifySortition}(m.\text{sorthash}, m.\text{pi}, m.\text{pk}) \& \text{ValidateMsg}(m)\}$ 
15:      $\text{FullBlockToPropose} \leftarrow \text{empty\_block}(\text{round})$ 
16:     if  $\text{lowestHashMsg} \in \text{msgs.Sent}$  then
17:        $\text{FullBlockToPropose} \leftarrow B$ 
18:        $\text{FullBlockToPropose.seed} \leftarrow \text{ComputeSeed}(\text{ctx}, r, B)$ 
19:        $\text{GOSSIP\_MSG}(a.pk, \text{Signed}_{a.sk}(B, \langle \text{sorthash}, \pi \rangle))$ 
20:     else
21:       while elapsed time < fullblock time do
22:         incomingMsgs.push(listen())
23:          $m \leftarrow \text{incomingMsgs.back}()$ 
24:         if  $\text{ValidateMsg}(m) \& \text{lowestHashMsg.blockHash} = \text{Hash}(m.\text{FullBlock})$  then
25:            $\text{FullBlockToPropose} \leftarrow m.\text{FullBlock}$ 
return  $H(\text{FullBlockToPropose})$ 
```

---

### Arguments:

- A block  $B$  received from the previous stage
- A list of online accounts  $A = \{a\}$  associated with this node
- ephemeral keys
- public key 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()`, `sortitionw()` and `sortitionW()` described in Section ??). The sortition subroutine determines if an account  $a$  is selected to *sorthash*, a proof of the sortition  $\pi$ , 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  $\langle \text{sorthash}, \pi, j \rangle$  it computes the hashes  $\{\text{Hash}(\text{sorthash}||n) : 1 \leq n \leq j\}$  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, XX.) 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  $(\text{priority}, \langle \text{sorthash}, \pi \rangle)$  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 *proposaltime* 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.

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 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 *fullblocktime* 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, 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.

## 2.3 Soft Vote

This step is only undertaken by the round's committee, i.e., those accounts that have a  $j > 0$  in the previous step. The soft vote stage (step 1, labeled as reduction in [?]) 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.

Ojo, no todos estan. Revisar!

Que es la UserPriorityHash? Nunca vuelve a referenciarla De donde sale proposal time? Que quiere decir validar?

Que es reduction one/two,  $\lambda_{step}$ , hblock

---

**Algorithm 3** Soft Vote

---

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

---

**Arguments:**

- A block hash, *BlockHash*, received from the previous stage (Sect ??).

**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  $\lambda$  seconds have passed using the subroutine *CountVotes()* –both subroutines being described in Section ?? in more detail.

In the second step, committee members vote for the hash that received at least the amount of votes set by the threshold for this step, or the hash of the default empty block if no hash was observed to receive enough votes.

It relies heavily on the assumption that, if the block proposer was honest, most users will get to the soft vote stage with the same hblock parameter. However, if some proposers were dishonest, then no single hblock may be popular enough to cross the threshold (and therefore this stage will return an empty\_hash) and there will at most be one non-empty block that can be returned by soft vote for all honest users.

**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 4** CertifyVote

---

```
function CertifyVote(hblock)
  localBlockHash  $\leftarrow$  hblock
  step  $\leftarrow$  2
  while step < 256 do

    (hblock, bConfirmed)  $\leftarrow$  BLOCK_STEP(hblock, localBlockHash, step)
    if bConfirmed then return hblock
    step ++

    (hblock, bConfirmed)  $\leftarrow$  EMPTY_STEP(hblock, step)
    if bConfirmed then return hblock
    step ++

    CommonCoinFlipVote(hblock, step)
    step ++
  HangForever()
```

---

Entonces  
especifiquemoslo  
arriba.  
Porque no  
aparece  
Ven solo  
el hash y  
votan a  
partir del  
hash?  
cual  
threshold?  
que es ese if  
hblock=timeout?  
porque at  
most one?  
Igual, quizas  
sacaria esta  
'opinion'  
de la  
descrip del  
algoritmo

### Arguments:

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

### Description:

The certify vote stage is the core of the BA\* algorithm. The main loop executes a maximum of 254 times (from step 2 to step 255, labeled “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 *hblock* 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 *hblock* 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.

### 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 5 *BlockConfirmation*

---

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

---

### Arguments:

- A block hash, *BlockHash* received from the previous stage (Sect. ??)

### 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

What is the BA\* algorithm? Introduce it earlier What labels?

No se entiende lo de hangforever. Que quiere decir?

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 6 `CountVotes`

---

```

function CountVotes(ctx, round, step, T,  $\tau$ ,  $\lambda$ )
  startTime  $\leftarrow$  currentTime()
  counts  $\leftarrow$  {} // hash table, new keys mapped to 0
  voters  $\leftarrow$  {}
  msgs  $\leftarrow$  incomingMsgs[round, step]
  while TRUE do
    m  $\leftarrow$  msgs.next()
    if m =  $\perp$  then
      if ElapsedTime > startTime +  $\lambda$  then
        return TIMEOUT
    else
       $\langle$ votes, value, sorthash $\rangle \leftarrow$  ProcessMsg(ctx,  $\tau$ , m)
      if m.pk  $\in$  voters  $\vee$  votes = 0 then continue
      voters  $\cup =$  {m.pk}
      counts[value] += votes
      if counts[value]  $\geq T * \tau$  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
- $\tau$  = expected number of committee members
- $\lambda$  = 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.

**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 7** *CommitteeVote*

---

```
function CommitteeVote(ctx, round, step, T,  $\tau$ ,  $\lambda$ )
  role  $\leftarrow \langle \text{committee}, \text{round}, \text{step} \rangle$ 
   $\langle \text{sorthash}, \pi, j \rangle \leftarrow \text{Sortition}(\text{user.sk}, \text{ctx.seed}, \tau, \text{role}, \text{ctx.weight}[\text{user.pk}], \text{ctx.W})$  // only
  committee members originate a message
  if  $j > 0$  then
    GOSSIP_MSG(user.pk, Signeduser.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
- $\tau$  = expected number of committee members
- $\lambda$  = 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:**

Has no return value, only gossips the specified vote if the caller is a committee member

---

**Algorithm 8** *ComputeSeed*

---

```
function ComputeSeed(ctx, round, B)
  if B  $\neq$  empty_block then
    return VRFgetSKu(ctx, r)(ctx.LastBlock.seed||r)
  else
    return H(ctx.LastBlock.seed||r)
```

---

**Arguments:**

- *ctx* = a helper structure to retrieve ledger context information (e.g. the last confirmed block)
- *round* = current round number
- *B* = the block whose seed is being computed (that will be added as metadata)

**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  $\text{get}_{SKu}(\text{ctx}, \text{round})$  helper function gets the relevant user's secret key (according to the signing scheme described in specs). That is, the secret key from a round  $b$  time before block  $r - 1 - (r \bmod R)$ , where  $R$  is the sortition seed's renewal rate,  $r$  is the current round's number, and  $b$  is the upper bound for the loss of strong synchrony (according to the weak synchrony assumption, that is,  $s < b$ ).

TODO: reescribir detallando mejor el proceso de selecci3n de la seed

**Returns:**

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



---

**Algorithm 9** Sortition

---

```
function Sortition( $sk, seed, \tau, role, w, W$ )  
   $\langle hash, \pi \rangle \leftarrow \text{VRF}(seed || role)$   
   $p \leftarrow \frac{\tau}{W}$   
   $j \leftarrow 0$   
  while  $\frac{hash}{2^{hashlen}} \notin [\sum_{k=0}^j B(k; w, p), \sum_{k=0}^{j+1} B(k; w, p))$  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
- $\tau$  = 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 (aka. 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 (aka. 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  $\tau$  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 amount 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 10** VerifySortition

---

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

---

**Arguments:**

- $pk$  = a user's public key (their address)
- $seed$  = the sortition seed to be used
- $\tau$  = 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 (aka. it's 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 11** BLOCK\_STEP

---

```

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

```

---

**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  $\langle r, bConfirmed \rangle$ , 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 12** EMPTY\_STEP

---

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

---

**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  $\langle r, bConfirmed \rangle$ , 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 13** CommonCoinFlipVote

---

```
function CommonCoinFlipVote(hblock, blockHash, step)
  r  $\leftarrow$  hblock
  CommitteeVote(ctx, round, step,  $\tau_{step}$ , r)
  r  $\leftarrow$  CountVotes(ctx, round, step,  $T_{step}$ ,  $\tau_{step}$ ,  $\lambda_{step}$ )
  if r = TIMEOUT then
    if CommonCoin(ctx, round, step,  $T_{step}$ ,  $\tau_{step}$ ,  $\lambda_{step}$ ) = 0 then
      r  $\leftarrow$  blockHash
    else
      r  $\leftarrow$  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 14** *CommonCoin*


---

```

function CommonCoin(ctx, round, step,  $\tau$ )
  minhash  $\leftarrow 2^{\text{hashlen}}$ 
  for  $m \in \text{incomingMsgs}[\text{round}, \text{step}]$  do
     $\langle \text{votes}, \text{value}, \text{sorthash} \rangle \leftarrow \text{ProcessMsg}(\text{ctx}, \tau, m)$ 
    for  $1 \leq j < \text{votes}$  do
       $h \leftarrow \text{Hash}(\text{sorthash} || j)$ 
      if  $h < \text{minhash}$  then minhash  $\leftarrow h$ 
  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
- $\tau$  = 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)