1. Introduction

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.

2. Main algorithm

Algorithm 1 Block creation

1: **function** $BlockCreation()$
2: $Pick\ transactions\ from\ transactionPool$

The algorand protocol specs don't ensure a specific way or internal order in which transactions are added to the block. In fact, they don't force transactions to be added at all, being entirely possible for a node to propose an empty block. More research is needed on how block data is put together. Block metadata is well documented, but is added into it later in the block proposal stage.

Algorithm 2 Block proposal

function $BlockProposal(Block\ B,\ Accounts\ A)$
$ProposalMsgsSent = \{\}$ //Set of proposal messages sent by this node
for $a \in A$ **do**
$< sorthash, \pi, j > \leftarrow Sortition(a.sk, getSortitionSeed(round), t, role, sortition_w(a, round), sortitionW(round))$
if $j > 0$ **then**
$priority \leftarrow Min_{n \in [1,j]} Hash(sorthash||n)$
$UserPriorityHashTable[a] = priority$
$msg \leftarrow MSG(a.pk, Signeda.sk(priority, < sorthash, \pi >))$
$ProposalMsgsSent \cup = msg$
$GOSSIP\_MSG(a.pk, Signeda.sk(priority, < sorthash, \pi >))$
while $elapsed\ time < proposal\ time$ **do**
$incomingMsgs.push(listen())$
$msgs \leftarrow incomingMsgs[round, step = 0]$
$lowestHashMsg \leftarrow m \mid m.priority = Min_{m.priority\ \in\ incomingMsgs} \{verifySortition(m.sorthash, m.pi, m.pk)\ \&ValidateMsg(m)\}$
$FullBlockToPropose \leftarrow empty\_block(round)$
if $lowestHashMsg \in msgsSent$ **then**
$FullBlockToPropose \leftarrow B$
$FullBlockToPropose.seed \leftarrow ComputeSeed(ctx, r, B)$
$GOSSIP\_MSG(a.pk, Signeda.sk(B, < sorthash, \pi >))$
else
while $elapsed\ time < fullblock\ time$ **do**
$incomingMsgs.push(listen())$
$m \leftarrow incomingMsgs.back()$
if $ValidateMsg(m)$ & $lowestHashMsg.blockHash = H(m.FullBlock)$ **then**
$FullBlockToPropose \leftarrow m.FullBlock$
return $H(FullBlockToPropose)$

The block proposal stage (step 0 on specs) is the first stage of the consensus algorithm per se. Nodes loop through all their online accounts, running sortition to determine which accounts

will be proposing the assembled block (if any). It then proceeds to gossip the priority, sortition hash and proof of the designated accounts (signed with their leaf ephemeral keys according to the signature scheme described on specs). Afterwards, it waits for a designated time to receive other block proposals, validates them and keeps the one with the lowest observed hash (maximum priority). Once the time is reached, if the selected proposal came from this node, the seed is computed and added into the finished block's metadata. Then, the complete block is gossipped to the network. In the most common case where the proposal is not the one selected locally by the node, it waits for another pre-designated time to receive the full block from the network. In case of timeout in this last stage, it falls back to the empty block.

Algorithm 3 Soft Vote

function $SoftVote$(BlockHash)
  $hblock \leftarrow CountVotes(ctx, round, REDUCTION\_ONE, Tstep, tstep, lblock + lstep)$
  $empty\_hash \leftarrow H(Empty(round, H(ctx.last\_block)))$
  if $hblock = TIMEOUT$ **then**
    $CommitteeVote(ctx, round, REDUCTION\_TWO, tstep, empty\_hash)$
  else
    $CommitteeVote(ctx, round, REDUCTION\_TWO, tstep, hblock)$
  $hblock \leftarrow CountVotes(ctx, round, REDUCTION\_TWO, Tstep, tstep, lblock + lstep)$
  if $hblock = TIMEOUT$ **then return** $empty\_hash$ **else return** $hblock$

The soft vote stage (step 1, or labeled as reduction in the ASBAC paper) aims to reduce any ammount of potentially conflicting proposed blocks in the stage prior into a binary choice, either a block proposed by a user or an empty one. In the first sub-step, each commitee member gossips the proposed block, and then does a preliminary vote count while it waits for other nodes to catch up. In the second sub-step, committee members vote for the hash that received at least the ammount 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 the proposer was dishonest, then no single hblock may be popular enough to cross the threshold (and therefore this stage will return an empty_hash). Therefore, there will at most be one non-empty block that can be returned by soft vote for all honest users.

Algorithm 4 CertifyVote

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

    $< hblock, bConfirmed > \leftarrow BLOCK\_STEP(hblock, blockHash, step)$
    if $bConfirmed$ **then return** $hblock$
    $step++$

    $< hblock, bConfirmed > \leftarrow EMPTY\_STEP(hblock, step)$
    if $bConfirmed$ **then return** $hblock$
    $step++$

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

The certify vote stage (which potentially encompasses steps 2 through 255) is the core of the

BA* algorithm. The main loop executes a maximum of 254 times (from step 2 to step 255, labeled c̈ertänd d̈own̈respectively 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 main algorithm is comprised of three important procedures: 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. 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. Finally, 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 tho 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 can't 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.

Algorithm 5 BlockConfirmation

function $BlockConfirmation(hblock)$
  $r \leftarrow CountVotes(ctx, round, FINAL, Tfinal, tfinal, lstep)$
  if $r = hblock$ **then**
    return $< FINAL, BlockOfHash(hblock) >$
  else
    return $< TENTATIVE, BlockOfHash(hblock) >$

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.

3. Subroutines

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 ammount of votes in the predetermined temporal window, it finishes with a TIMEOUT. It's important to notice that every committee user's vote is processed at most once for the given context.

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.

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

Algorithm 6 CountVotes

function $CountVotes(ctx, round, FINAL, Tfinal, tfinal, lstep)$
  $counts \leftarrow \{\}$ // 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
      $< votes, value, sorthash > \leftarrow ProcessMsg(ctx, t, m)$
      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** $value$

Algorithm 7 CommitteeVote

function $CommitteeVote(ctx, round, FINAL, Tfinal, tfinal, lstep)$
  $role \leftarrow <\backslash committee", round, step>$
  $<sorthash, , j> \leftarrow Sortition(user.sk, ctx.seed, , role, ctx.weight[user.pk], ctx.W)$ // only committee members originate a message
  if $j > 0$ **then**
    $GOSSIP\_MSG(user.pk, Signed_{user.sk}(round, step, sorthash, \pi, H(ctx.last_block), value))$

Algorithm 8 ComputeSeed

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

seed. The $get_{SKu}(ctx, round)$ 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 $r-1-(rmodR)$, 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, $s < b$).

Algorithm 9 Sortition

function $Sortition(sk, seed, \tau, role, w, W)$
  $< hash, \pi > \leftarrow VRF(seed||role)$
  $p \leftarrow \frac{t}{W}$
  $j \leftarrow 0$
  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 $< hash, \pi, j >$

The Sortition procedure is one of the most important subroutines in the main algorithm, as it's 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 τ is the expected ammount 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.

Algorithm 10 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 $\frac{hash}{2^{hashlen}} \notin [\sum_{k=0}^{j} B(k; w, p), \sum_{k=0}^{j+1} B(k; w, p))$ **do**
    $j++$
  return j

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 ammount of times the user was chosen according to their respective observed stake.

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 (aka. 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 amd a flag to either continue or return from the main procedure with a certified vote.

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.

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

Algorithm 11 BLOCK_STEP

function $BLOCK\_STEP(hblock, step, blockHash)$
  $bConfirmed \leftarrow FALSE$
  $r \leftarrow hblock$
  $CommitteeVote(ctx, round, step, \tau_{step}, r)$
  $r \leftarrow CountVotes(ctx, round, step, Tstep, step, step)$
  if $r = TIMEOUT$ **then**
    $r \leftarrow blockHash$
  else if $r \neq emptyHash$ **then**
    for $step < s \leq step + 3$ **do**
      $CommitteeVote(ctx, round, s, step, r)$
    if $step = 1$ **then**
      $CommitteeVote(ctx, round, FINAL, \tau_{final}, r)$
    $bConfirmed \leftarrow TRUE$
  return $< r, bConfirmed >$

Algorithm 12 EMPTY_STEP

function $EMPTY\_STEP(hblock, step)$
  $bConfirmed \leftarrow FALSE$
  $r \leftarrow hblock$
  $CommitteeVote(ctx, round, step, \tau_{step}, r)$
  $r \leftarrow CountVotes(ctx, round, step, Tstep, step, step)$
  if $r = TIMEOUT$ **then**
    $r \leftarrow emptyHash$
  else if $r = emptyHash$ **then**
    for $step < s \leq step + 3$ **do**
      $CommitteeVote(ctx, round, s, step, r)$
    $bConfirmed \leftarrow TRUE$
  return $< r, bConfirmed >$

Algorithm 13 CommonCoinFlipVote

function $CommonCoinFlipVote(hblock, step, blockHash)$
  $r \leftarrow hblock$
  $CommitteeVote(ctx, round, step, \tau_{step}, r)$
  $r \leftarrow CountVotes(ctx, round, step, Tstep, step, step)$
  if $r = TIMEOUT$ **then**
    if $CommonCoin(ctx, round, step, Tstep, \tau_{step}, lambdaStep) = 0$ **then**
      $r \leftarrow blockHash$
    else
      $r \leftarrow emptyHash$

what the next vote will be for a given user, and introduces a tool to facilitate reaching a tentative consensus in the next stages.

Algorithm 14 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 ← H(sorthash||j)
            if h < minhash then minhash ← h
    return minhash mod 2
```

The CommonCoin procedure takes advantage of the randomness of sortition hashes, and attempts to get a consensus random bit (a çoin 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.