Karada: an objective and truly decentralised Proof of Stake protocol

Kuroi Hakuchō, kuroi.hakucho@tutanota.com

April 2023

"And the LORD said, Behold, the people *is* one, and they have all one language; and this they begin to do: and now nothing will be restrained from them, which they have imagined to do." *(Genesis 11:6 KJV)*

Abstract.

We present Karada, the first *objective* and *truly decentralised* Proof of Stake (PoS)[1] protocol. All modern PoS protocols suffer from what is referred to as weak subjectivity[2] [3], *i.e.*, a node synchronising with a network for the first time or after a long period of being disconnected, when presented with two or more self-consistent (*i.e.*, valid according to the given protocol's rules) blockchain versions, can't autonomously establish which version is the canonical one. Karada allows for a computationally inexpensive comparison of the canonicity of any two forks by comparing them in terms of the amount of stake used in the course of their production. Crucially, achieving consensus within Karada doesn't require electing a subset of nodes in the form of a block-validating committee, allowing an arbitrarily large set of nodes to share the task of validating transactions, yet without exponentially increasing communication complexity. This way, the advantages of PoS protocols can be achieved while still preserving objectivity comparable to, and decentralisation superior to, that of Proof of Work (PoW)[4].

Introduction.

The present document is the first part of a two-part whitepaper series, the second part being *Kokoro: an algorithmic wealth redistribution.*

A breakthrough solution to the Byzantine General's Problem[5], a cryptocurrency called Bitcoin[6], brought a wave of interest in the field of decentralised consensus. A slew of consensus mechanisms, most notably Proof of Work and Proof of Stake, became popular. PoS protocols have these merits over PoW ones that they are energy-efficient and don't introduce the centralisation of consensus through the economies of scale, which increasingly govern the PoW mining industry[7]. There are, however, some important concessions made to achieve this. In any PoS protocol, a relatively small subset of nodes participating in sharing the task of validating transactions needs to be chosen either by using some sort of a verifiable random function to select them or by having some nodes delegate their right to participate to a set of delegates. In consequence, each time a block is produced, either a single node or a small group thereof confirms, through adding the block to a chain of blocks, the validity of all the blocks which precede it (*i.e.*, the added block). Attempting to allow an unlimited number of nodes to be a part of the committee which obtains consensus on the canonical blockchain would result in a prohibitive communication complexity, which grows exponentially in respect to the number of nodes in the mentioned committee, *i.e.*, it would require $O(n^2)$ communication[8]. This, in turn, minimises the potential level of decentralisation, which comes with an increased risk of power collusion, bribery, *etc*.

Furthermore, another problem ensues. A sufficient number of nodes from a given committee, selected to co-produce some blocks at time *t*, can costlessly simulate multiple valid variants of the blocks, either during *t* or, perhaps more likely, at some time subsequent to *t* (*e.g.*, when the nodes are not invested in the system anymore). The smaller the committee, the more likely this is to take place. If it does indeed occur, a node (re)synchronising with the network, when faced with two or more differing (but self-consistent) blockchain extensions, needs to choose the canonical version by asking a trusted third party (an exchange, a block explorer, a friend, *etc.*) about it. There is no effective measuring mechanism that would allow the node to quantitatively compare the canonicity of two self-consistent blockchain extensions (as is possible in PoW). Therefore, the only way for nodes to come to the same conclusion on which version of the transactional history is correct would be to rely on trusted, centralised oracle-like institutions, which would then decide on the canonical chain. In face of a conflict, the network would split and the only available recourse would be off-chain politics. Even if the likelihood of this actually happening is relatively low, the consequences of such an event[9] would change the security assumptions involved in designing PoS protocols.

We present a PoS protocol that allows, each time a block is to be produced, an arbitrarily large set of nodes to *directly* (*i.e.*, without the need to select, either pseudorandomly or through stake delegation, a relatively small subset of nodes) take part in the consensus on the canonical chain of blocks which precede the said to-be-produced block, where communication complexity required is approximately constant, *i.e.*, *O(1)*. Furthermore, the protocol comes with a fork-choice rule which allows a (re)synchronising node to establish, in a computationally inexpensive way and under practically any plausible conditions, the canonical chain of blocks.

Karada is a permissionless, Byzantine Fault tolerant PoS protocol. Faced with a network partition, it favours consistency over availability. We will analyse the protocol with the assumption that it works in the synchronous network model. We assume that the protocol works within the framework of an account-based record-keeping model which incorporates world state and state transition trees.

1. Blockchain structure and staking

Karada blockchain is divided into epochs. Each epoch is in turn divided into two parts: *a staking part* and *a non-staking part*. The staking part of an epoch *e* is the set S_x, comprising the first 100 blocks from all the blocks which the specified epoch *e* comprises. The non-staking part of *e* is the set S_y, which comprises the 100 blocks subsequent to the first 100 blocks covered by the staking part of *e*. Block production of the 100th block in the non-staking part of *e* marks the end of the epoch *e*. Since the non-staking part of an epoch starts after the epoch's staking part, $S_x \cap S_y = \varphi$.

In order to start participating in consensus, a node needs to perform a *staking transaction*. To do that, a node *n*, which owns an address *a*, generates and signs with the use of a private key *pk* associated with the address *a* (that is, *pk* is associated with the public key from which *a* was derived) a transaction which confirms that the node wants to stake coins, *i.e.*, a staking transaction. Once the staking transaction is included in a block, the coins in *a* become frozen (unspendable) and the staking transaction becomes a part of the blockchain. A staking transaction freezes funds until an *unstaking transaction* unfreezes them. For example, *n* will be able to unstake and unfreeze the staked funds in *a* by generating and having included (subsequently to having made a staking transaction) in one of the blocks a transaction signed using *pk* that expresses the node's will to unstake the funds in question. Each single address can be used to make either a staking transaction or (if the address was used to make a valid staking transaction already) an unstaking transaction, which freeze and unfreeze funds in the address in question, respectively, once in every epoch.

Both staking and unstaking transactions can be included only in blocks that are produced within the staking part of an epoch (*i.e.*, in its first 100 blocks) and can't be included in blocks produced within its non-staking part (*i.e.*, in the 100 blocks which are subsequent to its staking part). For example, if a node *n* wanted to make a staking transaction in an epoch *e*, *n* could do so by including the transaction in one of the blocks from the set $S_d=\{b_1, b_2, \ldots, b_{100}\}$, where b_1 is the first block of *e* and b_{100} is the last block produced in the staking part of *e*, thus automatically freezing the funds in *a,* which belongs to *n* and is used to make the staking transaction. Furthermore, if *n* wanted to make an unstaking transaction afterwards, thus unfreezing the funds in *a*, *n* could do so at the earliest during the production of the next epoch, the one succeeding *e*, let us call it *e′*, by including the unstaking transaction in one of the blocks contained in the staking part of *e′.* Alternatively, *n* could do so in one of the blocks within a staking part of any epoch succeeding *e,* not just *e′*. As we have mentioned above, one address can't be staked and unstaked in the same epoch.

While a staking transaction freezes the staked funds immediately upon becoming a part of the blockchain (by being included in a block), an unstaking transaction, even if included in the blockchain, unstakes the funds (*i.e.*, returns them to their owner and makes them spendable) only starting from the first block of the epoch succeeding the epoch in which the unstaking transaction was included. For example, an unstaking transaction included in one of the blocks within the staking part of an epoch *e* will unfreeze the funds in the relevant frozen address with the effect from the first block of the epoch immediately succeeding *e, i.e., e'*, onwards.

Each staking transaction, besides expressing the will to stake some funds, includes a single asymmetric public key, which is a required part of such a transaction. Such a key is generated by a staking node and is called a *public participation key*. The asymmetric private key associated with a given public participation key is never disclosed in a staking transaction but kept secret by the staking node. The associated private key is called a *secret participation key*.

A public participation key ppk_1 included in a staking transaction *st* in an epoch *e* becomes a *valid* public participation key starting from the first block of the epoch immediately succeeding *e*, that is *e'*, and continues to be a valid public participation key until the funds staked in *st* become unfrozen, that is until the first block of the epoch immediately succeeding the epoch in which the transaction unstaking the funds frozen by *st* takes place. The *validity* of public participation keys is explained in subsection 2.1., where their mode of operation and the way they are used (when valid) within the protocol are described.

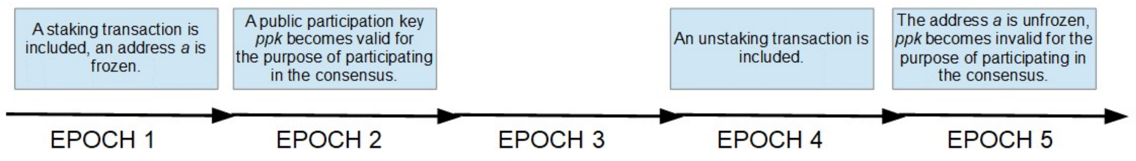

A staking transaction is included, an address *a* is frozen.
A public participation key *ppk* becomes valid for the purpose of participating in the consensus.
An unstaking transaction is included.
The address *a* is unfrozen, *ppk* becomes invalid for the purpose of participating in the consensus.
EPOCH 1
EPOCH 2
EPOCH 3
EPOCH 4
EPOCH 5

Illustration 1. In this example, in one of the blocks belonging to the staking part of epoch 1, a staking transaction made by a node *n* was included. The funds in an address *a,* which was used by *n* to make the transaction, became frozen immediately upon inclusion of the transaction. In the transaction, a public participation key *ppk* was revealed by *n*. Thus, *ppk* became valid and *n* was able to use it to participate in the consensus starting from epoch 2 onwards. However, *n* has decided to make an unstaking transaction during the production of epoch 4 and the unstaking transaction was included in one of the blocks belonging to the staking part of epoch 4. This transaction has unstaked and unfrozen the funds in *a* with the effect from the first block of epoch 5 onwards. It also meant that *n* won't able to use *ppk* to participate in the consensus with the effect from epoch 5 onwards.

As mentioned, staking and unstaking transactions can be included only in blockchain blocks produced within the staking part of an epoch. Each such block includes them in the form of data blocks of a separate hash tree. A hash tree containing staking/unstaking transactions is called a *staking transition tree*. A staking transition tree, whose root hash is included in a block *B* produced in an epoch *e*, comprises the staking/unstaking transactions included in *B* by the *B*'s producer. The staking/unstaking transactions have been generated, signed and broadcast in the network by the network participants which, through broadcasting the transactions, have signalled their will to, during the epoch *e*, either stake their funds or unstake their already staked funds; eventually, the transactions have reached the producer of *B*, and, given that each of them was generated and signed by its originator correctly, and not already a part of any of the preceding blocks in the epoch *e*, they were included in *B* by the *B*'s producer. The transactions, once included, take the effect (that is, actually stake or unstake funds) at the beginning of the subsequent epoch.

At the end of any given epoch's staking part, *i.e.*, in the first block produced in its non-staking part, the set of staking transactions which will be valid during the subsequent epoch's production is calculated and published. To arrive at the set of staking transactions which will be valid during the subsequent epoch, the set of staking transactions which *are* valid during the ongoing epoch's production is, in the first block of its non-staking part, transitioned through:

1.) Adding the staking transactions included in all the staking transition trees whose root hashes are contained in all the blocks produced in the staking part of the epoch whose production is ongoing.

2.) Subtracting the staking transactions subject to the unstaking transactions (that is, subtracting the staking transactions which are being unstaked through the said unstaking transactions) contained in all the blocks produced in the staking part of the epoch whose production is ongoing.

The transitioned staking state, that is the one in which some staking transactions were added and some subtracted, is converted into a hash tree, referred to as a *staking state tree*, whose root hash is published in the first block of the non-staking part of the epoch whose production is ongoing. This hash tree establishes the set of staking transactions which will be *valid* for the purposes of block production during the subsequent epoch.

2. Consensus on a single block

2.1. Steps to obtain consensus on a new block

For the network to function correctly, the nodes need to have their system clocks synchronised (*e.g.*, using an NTP server) with a bounded clock drift generally not exceeding a few seconds (we don't define the maximum clock drift allowed, however, the larger the drift, the smaller the odds of an affected node becoming a block producer).

Block production is scheduled in the network into unique one-minute long timeframes, where each timeframe is designated for the production of a single block. The timeframes are numbered; the very first timeframe, which will occur during the first minute subsequent to the network's starting time, will be designated as the 1st one, the second – as the 2nd one, *etc*. For example, let's assume that the very first block to be produced in the network, that is *B1*, is to be produced within the 60 seconds subsequent to the network's starting time, expressed in UTC as XXXX-XX-XXT12:10:01Z, that is within the timeframe from XXXX-XX-XXT12:10:01Z until XXXX-XX-XXT12:11:00Z. It means that the timeframe designated for the subsequent block's, *B2*'s, production is XXXX-XX-XXT12:11:01Z – XXXX-XX-XXT12:12:00Z. *B1* could be produced and broadcast in the network at, for example, 30th or 40th second of the timeframe prescribed for its broadcast, but the 60-second timeframe prescribed for the production and broadcast of *B2* would anyways be counted from the moment the whole one-minute timeframe for the *B1's* production and broadcast has ended. Thus, every single timeframe, known in advance, is a one-minute time interval designated for the production and broadcast of a block during that timeframe. Furthermore, each block which is produced and becomes a part of the blockchain is associated with a *height*, also referred to as a *blockchain height*. The blockchain height of a block always corresponds to the number of the timeframe during which the block was designated to be produced and broadcast. For example, if a block *B* was designated to be produced and broadcast during the 3100th timeframe, which has occurred during the 3100th minute subsequent to the moment the network started to operate, then that block's blockchain height is 3100 – irrelevant of the block's distance, expressed in blocks, from the genesis block (which is how the terms *blockchain height* or *block's height* are normally used in literature).

Each block in Karada blockchain must be signed by a pseudorandomly selected node, called a producer. Block production consists of generating a block and signing it using an appropriate secret participation key, known only to the block's producer. The method of block producer's selection and block production will be presented in the present subsection. Karada, unlike other protocols, doesn't achieve a tamper-proof consensus on the transactional contents of blocks (that is, on which block contains, among other things, which state transitions/world state) but on the ordering of block producers, *identified by public participation keys*. What this means is that each subsequently produced block can be produced (by a single producer) in many variants, each variant containing differing world state and state transitions (as well as other variable elements which will be described below), while all the variants nevertheless need to point at only one particular producer of the preceding block. The protocol is designed to achieve consensus simply on which producer produced a block and at which blockchain height (as understood above, *i.e.*, during which timeframe) and *not* on the transactional contents of blocks.

This seems to cause a problem because a single producer, once chosen to produce a block at a height *h*, could create an infinite number of block variants at *h*, each of which is signed using the same private participation key (thus valid) but with various transactional data included in differing variants of his block. However, given that the protocol allows its participants to obtain consensus on a single queue of block producers *Q*, where *Q* can be understood as a set of numbered timeframes, each associated with a particular public participation key which identifies the block producer that has produced a block during the given timeframe, only a chain of blocks produced by the block producers as they are ordered in *Q* can be considered valid. Let's assume that the network has reached consensus that the currently valid blockchain comprises the set of blocks $B=\{B_1, B_2, B_3, \dots, B_n\}$ (however, without reaching consensus on the specific contents of each block), where each block from the set *B* is associated with a particular producer, identified by a public participation key, in accordance with the way they are ordered in the queue *Q*. The ordered set of thus identified producers, which have produced blocks from *B* is $P=\{P_1, P_2, P_3, \dots, P_n\}$, where P_1 has produced the block B_1, P_2 the block B_2, and so on. Some of the producers from *P* may have produced their blocks in many variants, but each producer had to produce his block in *at least* one valid variant; therefore, for every valid version of the set *B*, there is one block produced at a particular height by a particular producer from the set of producers *P*, and there is no producer in *P* which hasn't produced one block in *B*.

Now, as will be explained later in the present subsection, each block header has to comprise the hash digest of the preceding block's header; and since the hash digest of a preceding block's header indirectly points at the hash digests of the headers of all the blocks which precede that block as well (as each of them also needs to point at the hash digest of the preceding block's header, thus forming a chain of hashes), the hash digest of the previous block's header included in a particular block's header points at the hash digests of the headers of all the blocks up to that particular block. For example, should a producer *P100* produce his block *B100* in only one variant, then, even if every block from *B1* to *B99* was produced in many variants, there would be only one set of blocks produced, in that order, by the producers *P1* to *P100*, which forms a valid blockchain. It would be this way since the single block variant of *B100* would determine the hashes of the block headers of *B1-B99* and, because of that, there would be only one valid version of the set of blocks *B1-B100* in which each block is produced by a proper producer.

In conclusion, <u>a block at a height *h* produced by a producer *P* can be produced in an infinite number of variants, each containing a differing world state, state transitions and other variable elements. However, any producer which will produce a block in just one variant at any height subsequent to the one of the *P*'s block, will serve as a tie-breaker, resolving the question as to which block variant at the height *h* (and, implicitly, at each height which precedes *h*) is correct.</u>

Height 100 (**1** variant produced):	Height *101* (**2** variants produced):	Height 102 (**1** variant produced):
Block *B100* *B100* producer: *P100* *B100* points to the previous block's (*B99*) producer: *P99* The set of state transitions included in B100: *St100={S100t₁, S100t₂, S100t₃, …, S100tₙ}*. The new state reached in B100: *S100* The set *St100* transitions the state of the preceding block, *i.e.*, *S99*.	**Block *B101*** *B101* producer: *P101* *B101* points to the previous block's (*B100*) producer: *P100* The set of state transitions included in B101: *St101={S101t₁, S101t₂, S101t₃, …, S101tₙ}*. The new state reached in B101: *S101* The set *St101* transitions the state of the preceding block, *i.e.*, *S100*.	**Block *B102*** *B102* producer: *P102* *B102* points to the previous block's (*B101*) producer: *P101* The set of state transitions included in B102: *St102={S102t₁, S102t₂, S102t₃, …, S102tₙ}*. The new state reached in B102: *S102* The set *St102* transitions the state of the preceding block, *i.e.*, *S101'*.
	Block *B101'* *B101'* producer: *P101* *B101'* points to the previous block's (*B100*) producer: *P100* The set of state transitions included in B101': *St101'={S101't₁, S101't₂, S101't₃, …, S101'tₙ}*. The new state reached in B101': *S101'* The set *St101'* transitions the state of the preceding block, *i.e.*, *S100*.	

Illustration 2. As we have specified, the protocol achieves an explicit consensus on the order of block producers. Each block must include a digital signature generated by its own producer and also point at the preceding block's producer. Since a block signed by one producer *P* can point only at one producer (*P-1*) of the block that precedes it, even if *P* produces his block in many variants, *P* isn't able to point at more than one previous block's producer (that is, *P-1*) in all of the variants produced by him. In the example above, the producer of *B100* (that is, the producer *P100*) has produced only one block variant (thus being an honest producer), comprising the state transition set *S100*, which transitions the state as included in the preceding block *B99,* that is the state *S99*. Afterwards, the malicious producer *P101* has produced and propagated two block variants at the blockchain height 101 – *B101* and *B101'*. Even though the malicious producer couldn't tamper with neither the digital signatures identifying him as the producer of *B101* and *B101'*, nor with what is the previous block's producer (*P100*) both block variants point at (so that, in summary, both *B101* and *B101'* had to be signed by the same producer *P101* and point at the same previous block's producer *P100*), he has generated differing sets of state transitions and, in consequence, differing new states reached, in the two block variants *B101* and *B101'*. This created temporarily diverged transactional histories in the network, despite there being only one order of block producers. However, an honest block producer, *P102*, was selected afterwards to produce a block at the height of 102. *P102* has chosen (from *B101* and *B101'*) only one of the two previous block's variants, that is *B101'*, as the veridical one and, in his block, transitioned the state from *B101'*, that is *S101'*. Thus, *B102* effectively became a tie-breaker by being produced in one variant only (which necessitated choosing only one preceding block's state to transition) and converged the transactional history.

Block producers are disincentivised from producing multiple block variants (*vide* section 3 below). As long as at least some of the block producers honestly extend just one version of the transactional history by producing a single block variant, a single queue of *x* ordered producers will always translate into a single *valid* transactional history of *x* blocks, where each blockchain height at which a block was produced corresponds to only one block variant which is valid according to the transactional history *as a whole* (*i.e.*, according to a transactional history in which every produced block is associated with at least one valid block variant and in which, as a whole, all the variants form a valid blockchain). Here, we have introduced the concept of an explicit consensus on the queuing of block producers and an implicit consensus on the transactional history. Furthermore, since any given block's producer can produce a block at a given height in just one variant only to produce another variant later on (as he still owns the private participation key which allows that), the fact that a block exists only in one variant at a particular moment doesn't mean that it will remain as such in the future. This means that the transaction finality is probabilistic, with each tie-breaking block produced on top of a block variant *B* increasing the probability of the transactions included in *B* being final.

Now we will describe how the consensus on a new block is achieved. As described in subsection 4.1. below, all consensus participants are necessarily aware of the same set of public participation keys valid for the duration of an epoch whose production is in progress. For an epoch *e*, a set of valid public participation keys $S=\{ppk_1, ppk_2, ppk_3, \ldots, ppk_n\}$ is thus established by each participant at the beginning of *e*. Each participant will, therefore, start participating in the block production during a particular epoch with the same set established.

Let's assume that *B1* is the first block to be produced during the epoch *e*. *B1* is to be produced at the blockchain height h_1, *i.e.*, the first height covered by *e*. Production of *B1* is scheduled to happen within a prescribed timeframe *ΔtB1*.

When *B1's* production timeframe *ΔtB1* ends, the network participants start the process which aims at selecting, during *ΔtB2*, the producer of a block at the subsequent height, *i.e.*, assuming that *B1* was produced during *ΔtB1*, a block *B2* at the height h_2. Each node which has a private participation key corresponding to a *valid* (for the duration of *e*) public participation key can take part in the process. In order to find *B2's* block producer the following steps are being taken during *ΔtB2*:

1.) Firstly, every participating node extracts from the last known block, that is from *B1*, a so-called *root block value*. What is a root block value and how it's extracted will be described at the end of the present subsection. A root block value is individual for each and every block produced by a single producer and always points both at the producer of the block it (the root block value in question) is included in and at the immediately preceding block's producer. This means that the root block value v_1 included in the block *B1* points at the *B1*'s producer *P1* and at the producer of the block immediately preceding *B1*; the root block value, v_2, included in *B2*, will point at the *B2*'s producer *P2* and at the producer of the block immediately preceding *B2*, that is at *P1*. At this step, by choosing only one root block value (if there is more than one producer which has produced a block at the preceding height), every participant selects a particular last block's (that is, produced at the blockchain height h_1) producer as valid and extracts the root block value from the block produced by the chosen producer. Because said value can't be later (during the subsequent steps) changed in any way and since all the subsequent steps depend on it, committing to a given root block value of the last block at this point means committing to a specific last block's producer.

2.) After choosing the root block value v_1 as the correct value of the last block, a node *n*, which we will use to exemplify the steps to be taken by all the nodes in the network, concatenates the chosen value v_1 with the numerically represented height of the *to-be-produced* block. The block *B2* is to be produced now at the blockchain height h_2; let's suppose that h_2 numerically is height 3002 (*i.e.*, it is the height which corresponds to the 3002nd timeframe, corresponding to the 3002nd minute after the network's starting time, analogously to how it was described at the beginning of the present subsection). In such a scenario, *n* concatenates v_1 with the height of *3002*, with the resulting string $s=v_1\#3002$ (for example, if v_1 were a number "123" then the resulting string *s* would be "1233002"; this is only an example, a real root block value would be much larger).

3.) The node *n* now creates a digitally signed message, we'll refer to it as *m*, where in said message:

-the string *s* is the data that is being signed (by using some digital signature scheme),

-the *n*'s private participation key *priv-pk* is the key that is used to sign the string *s* (where the corresponding currently valid public participation key *pub-pk* is included in the *n's* staking transaction *st*, which has frozen *n's* coins in one of the previous epochs).

The resulting *digital signature* itself will be referred to as *ds*.

The node *n* then broadcasts *m* in the network so that *m* could reach every other participant. Having composed and broadcast *m*, *n* has, through signing the message *m* using the *n's* private participation key *priv-pk,* committed to a particular root block value of the last block (*B1*), which *n* considers correct. *Priv-pk* is known only to *n* and only *n* can use it to sign messages which commit in this manner. However, the signed message *m* can be easily associated with the *n's* staking transaction *st* because the digital signature *ds* (which is a part of *m*) itself reveals the public participation key associated with the private participation key *priv-pk* used to sign *m*. That public key is *pub-pk*. And *pub-pk*, in order for the message *m* to be valid, needs to be a valid public participation key which is findable in one of the *currently* valid staking transactions (*i.e.*, valid for participation during the epoch *e*), the one in which *pub-pk* was included by *n* in the way we have explained in the previous section. In our example, this staking transaction is *st*.

4.) The message *m* is propagated in the network. Upon receipt of *m*, each participant associates the digital signature *ds* with the staking transaction *st* made by *n*. This can be easily done by scanning all the staking transactions valid during *e* (where all such transactions were contained in the staking state tree whose root hash was published in the epoch immediately preceding *e*, as described in the previous section) in search of the one that contains *pub-pk*. How each staking transaction needs to comprise a public participation key was described in the previous section. If *m* can be associated in this way (otherwise *m* would be considered invalid by the receiving participants), each participant which has received *m* checks how many coins were frozen in *st* (as we have explained in the previous section, a staking transaction freezes the coins that are being staked).

5.) Once said number of coins frozen in *st* is established, the number is associated with the message *m* by each participant which has received *m*. After associating *m* this way, each single participant who has received *m* hashes and rehashes the digital signature *ds* one time per one coin. For example, if we assume that 100 coins were frozen in *st*, then each participant which has received *m* will:

i.) firstly, hash *ds* once, obtaining the hash digest h_1,

ii.) secondly, hash the digest h_1 in order to obtain the digest h_2.

iii.) He will do it consecutively the number of times equivalent to the number of coins frozen in *st*, that is 100 times. As a result, the set of hash outputs $S_h=\{h_1, h_2, h_3, \ldots, h_{100}\}$ will be obtained by each participant which has received *m* (where h_3 is the digest obtained by hashing h_2, h_4 is the digest obtained by hashing h_3, *etc.*).

As we will see, the chances of becoming a block producer depend on the number of hash outputs that can be obtained during this step. This means that the more coins *n* would have frozen in *st*, the higher would be the *n's* chances of becoming the block producer of *B2*.

6.) Not only *n*, but each network participant which owns a valid public participation key establishes the root block value of the preceding block (*B1*), then generates and broadcasts a digitally signed message which is analogous to *m* (though a different root block value of the last block, which is a required part of such messages, can be chosen by any other node if there is more than one value known). This has to be done during the timeframe designated for *B2's* production, *i.e.*, during *ΔtB2*. Each participant sends his message and receives messages from other participants; then each one associates each of the received messages with an appropriate staking transaction and subsequently obtains a specific amount of hash outputs related to each message received, in accordance with the procedure described in the preceding item. Each hash output obtained in this way is referred to as a *ticket*.

7.) Each participant divides all of the tickets obtained by him into sets, where each single set contains tickets obtained using messages (received from the other participants) which, in their signed data part, include the same root block value (concatenated with the height of the currently-to-be-produced block, as in item 2) of the last block.

Let's suppose that *n* has received from some other participants some amount of messages which can be grouped into two *sets of messages*, where messages in each set contain in their signed data part one of the last block's root block values *A* or *B*. The node *n* divides all the tickets obtained on the basis of the messages received from the other participants into two disjoint sets, where the first set comprises the tickets obtained by serially hashing (as in the previous item) the digital signatures from the messages containing in their signed data part the root block value *A*; and *mutatis mutandis* for *B*. Each such separate set is called a *ticket set*.

8.) Each participant concatenates each ticket from a given ticket set with each other ticket from the same set, generating in this way pairs of concatenated tickets. Tickets from different ticket sets can't be concatenated with each other.

In our example, *n* has grouped the messages received from the other participants into two sets, one containing the messages with the root block value *A* and another containing the messages with the root block value *B*. The first set is $S_{MA}=\{m_1, m_2, m_3, \ldots, m_n\}$, and the second set is $S_{MB}=\{m_{1'}, m_{2'}, m_{3'}, \ldots, m_{n'}\}$. Let's suppose that, using the message m_1, in the way described in item 5 above, 20 tickets can be obtained, so that the set S_{m1} of the tickets obtained on the basis of m_1 is $S_{m1}=\{h_{m1\text{-}1}, h_{m1\text{-}2}, h_{m1\text{-}3}, \ldots, h_{m1\text{-}20}\}$, where $h_{m1\text{-}1}$ is the first digest obtained by hashing the digital signature from m_1, $h_{m1\text{-}2}$ is the digest obtained by hashing $h_{m1\text{-}1}$, and so on 20 times – let's assume that the valid staking transaction associated with m_1 has frozen 20 coins (so that 20 hash operations can be performed during the present step and 20 tickets obtained).

Now that *n* has obtained, from m_1, all the possible tickets, in order to find pairs of concatenated tickets, *n* first concatenates $h_{m1\text{-}1}$ and $h_{m1\text{-}2}$ so that the result is $h_{m1\text{-}1}\#h_{m1\text{-}2}$. This is the first pair of tickets, p_1. Afterwards, *n* concatenates $h_{m1\text{-}1}$ and $h_{m1\text{-}3}$, so that the result is $h_{m1\text{-}1}\#h_{m1\text{-}3}$. This is the second pair, p_2. The node *n* does that for all of the concatenated pairs which start with the ticket $h_{m1\text{-}1}$ (that is, until $h_{m1\text{-}1}\#h_{m1\text{-}20}$), then for all the pairs that start with $h_{m1\text{-}2}$ (*i.e.*, $h_{m1\text{-}2}\#h_{m1\text{-}1}$, $h_{m1\text{-}2}\#h_{m1\text{-}3}$, $h_{m1\text{-}2}\#h_{m1\text{-}4}$, ..., $h_{m1\text{-}2}\#h_{m1\text{-}20}$), thus obtaining a set of pairs of tickets, where each ticket concatenated with another was generated on the basis of the message m_1. Since there are 20 tickets in S_{m1}, there will be 380 pairs of tickets found within that set (20 tickets, each ticket can be concatenated with 19 other ones, so *20*19=380*).

After dealing in this manner with the set S_{m1}, the node n starts to concatenate each ticket obtained on the basis of m_1 (through hashing and rehashing the m_1*'s* digital signature) with each ticket obtained on the basis of every other message from the set S_{MA}. The first pair will be $h_{m1\text{-}1}\#h_{m2\text{-}1}$, (*i.e.*, the first ticket from S_{m1} concatenated with the first ticket from S_{m2}), then $h_{m1\text{-}1}\mathbin{+\!\!\!+}h_{m2\text{-}2}$, ..., and after concatenating $h_{m1\text{-}1}$ with each ticket from the set S_{m2}, n starts to concatenate $h_{m1\text{-}1}$ with every ticket from the set S_{m3}. The pairs obtained will be $h_{m1\text{-}1}\mathbin{+\!\!\!+}h_{m3\text{-}1}$, $h_{m1\text{-}1}\mathbin{+\!\!\!+}h_{m3\text{-}2}$, and so on. Once $h_{m1\text{-}1}$ is concatenated into a pair with every other ticket from the sets S_{m1} to S_{mn}, the procedure is performed again, although this time during each concatenation it is the ticket $h_{m1\text{-}2}$ that is the first in every pair concatenated. This is reiterated until every ticket is finally obtained on the basis of any given message from a given set of messages (here, all the messages containing the root block value *A*) is concatenated into a pair with every single other ticket obtained on the basis of the same message or on the basis of another message from the same set of messages. This process is done separately for each set of messages. If we assume that using the messages from S_{MA} 1200 tickets can be obtained, and, based on the messages from S_{MB}, 800 tickets, then based on S_{MA} 1438800 (as *1200*1199=1438800*) concatenated ticket pairs can be obtained and based on S_{MB} 639200 (as *800*799=639200*). No ticket obtained on the basis of a message from S_{MA} can be concatenated with a ticket obtained on the basis of a message from S_{MB}.

9.) Upon concatenation, each pair of concatenated tickets is hashed, resulting in a hash output (digest). For n, the hash digest resulting from hashing the first pair of concatenated tickets would be $h\colon h_{m1\text{-}1}\#h_{m1\text{-}2}\rightarrow d_1$, where d_1 is the digest resulting from the first hash operation; for the second pair, $h\colon h_{m1\text{-}1}\#h_{m1\text{-}3}\rightarrow d_2$ and so on for all the ticket pairs based on the messages from S_{MA}, thus obtaining 1438800 digests; then, analogously, for S_{MB} – 639200 digests. Each digest obtained in this manner is then measured in terms of how many leading digits from the set of digits from 1 to 8 it comprises. If a digit is either 1, 2, 3, 4, 5, 6, 7 or 8, it is considered suitable for this measurement; if a given digest is a string with more than x leading *suitable digits*, then the pair of concatenated tickets which was used as an input of the hash operation resulting in the digest is considered a *winning pair of tickets.* For example, if n finds that the hash function $h\colon h_{m1\text{-}x}\#h_{m1\text{-}y}\rightarrow d_n$, where d_n is a string with more than x leading suitable digits, then tickets $h_{m1\text{-}x}$ and $h_{m1\text{-}y}$ are a winning pair of tickets. The whole process described in the present section up to this point is aimed at finding such a winning pair of tickets.

How is x established? The required number of leading suitable (*i.e.*, from 1 to 8) digits changes for each new epoch. It is calculated for each given epoch in the first block of the non-staking part of the epoch that precedes it. For example, in the first block of the non-staking part of the epoch *e-1*, the root hash of the staking state tree with all the staking transactions that will be valid for the purpose of participation in block production during the succeeding epoch, that is *e* (*i.e.*, the epoch during which the block *B2* is about to be produced), was published; the number of coins frozen in all the staking transactions (included in the tree) which will be valid during *e* was calculated in said block. This calculation could have taken place at that point as, according to the previous section, no staking or unstaking transactions can be added or subtracted during a non-staking part of any epoch (and so the set of staking transactions valid during *e* can't be changed from that point onwards). Let's assume the number thus established to be 2000 frozen coins.

Since each single coin will allow for, during the timeframe designated for block production at any single blockchain height within *e*, generation of one ticket (as described in item 5 above), the fact that 2000 coins were frozen in the staking transactions valid during *e* means that maximum *2000*1999=3998000* pairs of concatenated tickets will be obtainable during block production at any single blockchain height within *e* (for this calculation, we never take into account the fact that, during the production of a block, there can be two or more ticket sets known, where the tickets from one set can't be concatenated with the ones from the other).

In Karada, we want a winning pair of tickets to be, on average, found after concatenating and subsequently hashing 25% of all possible (findable during the production of a block at a particular height and assuming there is only one ticket set known) ticket pairs. Returning to the example of the epoch *e*, we can calculate the desired amount of concatenated pairs of tickets which need to be, on average, hashed in order to find a winning pair of tickets during the production of a single block in *e* as *25%*3998000=999500*. This means that we want to, each time a block is being produced in *e*, make the network look for a digest with the number of leading digits, where each digit is a digit from 1 to 8, which can be found after hashing approximately 999500 concatenated pairs of tickets.

Such a digest would need to start with 20 leading suitable digits, each digit from 1 to 8. Let's consider that in any hexadecimal digest, the first hexadecimal digit has ½ chance of being a digit from the set of 1 to 8 (as there are 16 hexadecimal digits in total), the second hexadecimal digit has, independently, ½ chance of being such a digit and so on. Let *X* be the number of attempts consisting in hashing a concatenated pair of tickets. The expected value of *X*, if we want a digest to start with 20 leading suitable digits, can be calculated in the following way:

$$E(X) = \sum_{k=1}^{\infty} k \cdot p \cdot (1 - \mathrm{p})^{\,k-1}$$

$$= \sum_{k=1}^{\infty} k \cdot \left(\frac{1}{2}\cdot\frac{1}{2}\cdot\frac{1}{2}\cdot\frac{1}{2}\cdot\frac{1}{2}\cdot\frac{1}{2}\cdot\frac{1}{2}\cdot\frac{1}{2}\cdot\frac{1}{2}\cdot\frac{1}{2}\cdot\frac{1}{2}\cdot\frac{1}{2}\cdot\frac{1}{2}\cdot\frac{1}{2}\cdot\frac{1}{2}\cdot\frac{1}{2}\cdot\frac{1}{2}\cdot\frac{1}{2}\cdot\frac{1}{2}\cdot\frac{1}{2}\right) \cdot \left(1 - \left(\frac{1}{2}\cdot\frac{1}{2}\cdot\frac{1}{2}\cdot\frac{1}{2}\cdot\frac{1}{2}\cdot\frac{1}{2}\cdot\frac{1}{2}\cdot\frac{1}{2}\cdot\frac{1}{2}\cdot\frac{1}{2}\cdot\frac{1}{2}\cdot\frac{1}{2}\cdot\frac{1}{2}\cdot\frac{1}{2}\cdot\frac{1}{2}\cdot\frac{1}{2}\cdot\frac{1}{2}\cdot\frac{1}{2}\cdot\frac{1}{2}\cdot\frac{1}{2}\right)\right)^{k-1}$$

$$= 1048576 \approx 999500$$

This way, before the start of each epoch, the network would recalibrate its difficulty target for a winning pair of tickets by adjusting *x*, *i.e.*, the required number of leading suitable digits. Furthermore, an even more precise calculations could be used, where the last leading digit would be allowed to belong to some other set of digits, *e.g.*, 1-9 (which would slightly increase the probability of finding a proper hash digest) or 1-7 (which would slightly decrease the probability), so that the final probability would be as close to the target as possible. Since finding a winning pair of tickets allows for block production (*vide* item 11 below), by calculating in this way *x* for each new epoch, the network could (with some approximation) schedule: 1.) the share of network participants, weighed by coins staked, which need to generate messages belonging to the same set of messages (so pointing at the same last block's root block value and allowing for the generation of tickets which can be concatenated with each other into pairs) in order to find a winning pair allowing for block production and 2.) the number of winning pairs findable, and thus blocks produced, at a single height.

In order to concatenate and hash 25% of the total amount of ticket pairs which can be obtained during the consensus on a block at a single height, the tickets obtained on the basis of 50% of all coins validly staked in the network at that time would need to be concatenated into pairs and hashed. If we assume that more than half of the participants which have staked their coins are honest and non-faulty, the probability of finding a winning pair of tickets approaches (depending on by how much the number of participants exceeds 50%) 1. The number of winning pairs of tickets available (on average) at each height, should all the participants be honest and non-faulty, is 4. We believe that, in practice, the number of blocks produced per a height would be approximately within 2-3, which would be high enough to guarantee the network's liveness, but low enough to guarantee its consistency.

10.) In the preceding item, we have described how, in the first block of an epoch's non-staking part, a value *x* is obtained. Said value *x*, calculated in a given epoch, determines the number of leading digits from the set 1-8 required, during block production in the subsequent epoch, in a hash digest of a concatenated pair of tickets in order to qualify the pair as a winning pair of tickets. There are two other values, denoted as *y* and *z*, which are calculated in the first block of an epoch's non-staking part as well.

As mentioned in the previous item, the number of staked coins valid during a given epoch and, consequently, the maximum number of ticket concatenations obtainable at each height covered by the epoch, are calculated in the first block of the non-staking part of the epoch which precedes the epoch in question. According to the example from the previous item, in the first block of the non-staking part of the epoch *e-1*, the maximum amount of ticket concatenations achievable at each height of the succeeding epoch *e* was established as *2000*1999=3998000*.

In addition to *x*, two other values, denoted *z* and *y*, are calculated in the first block of an epoch's non-staking part as well. To establish said values *z* and *y*, the number of ticket concatenations findable, at each single height of the subsequent epoch (in our example, of the epoch *e*), using 51% of all the coins staked and valid during the subsequent epoch (here, *e*) is calculated. According to our example, as the total number of the staked coins is 2000, *51%*2000=1020*, so that *1020*1019=1039380* concatenations per a height of *e*.

Subsequently, the number thus obtained is divided by a power of 2 (*i.e.*, either 2, 4, 8, 16, *etc.*) such that the result of the division is a number as close to 100 as possible but larger than 100. In our example, the number 1039380 needs to be divided by 8192, *i.e.*, 2^{13}, which results in *1039380/8192=126,8...* (using 2 to the power of 12, *i.e.*, 4096, would yield a result too large – as *1039380/4096≈253,7*, while using 2 to the power of 14, *i.e.*, 16384, would yield a result too small - as *1039380/16384≈63,4*).

The resultant number, rounded to a natural number, is denoted as *z*; in our example, the value *z=127*.

The *exponent* used to calculate *z* is the value *y*; in our example, the values obtained in the first block of the non-staking part of the epoch *e-1* and *valid for block production in the subsequent epoch e were z=127 and y=13*.

The value *y*, calculated in the epoch *e-1*, indicates the number of leading digits which are either ones, twos, threes, fours, fives, sixes, sevens or eights, required in a hash digest of a pair of concatenated tickets found during the succeeding epoch *e*, in order for said pair to be considered *an approximate pair of tickets*. According to our example, if, at any height covered by the epoch *e*, during the concatenation and hashing operations described in items 8 and 9 above, a pair of concatenated tickets is found which hashes to a digest with *≥13* leading digits, where each of the leading digits is either a one, two, three, four, five, six, seven or eight, the pair is an approximate pair of tickets. For example, a hash digest starting with 4736216348472… would qualify. Since there are 16 hexadecimal digits, the first digit has a ½ chance of being a digit from the set 1-8, the second, has independently the same ½ chance, and so on. The chance of this happening, on average, for 13 leading digits is $(½)^{13}$*≈0.000122*. If there are 1039380 trials, each resulting in a hash digest comprising a pseudorandom string of hexadecimal digits and the chance of finding a proper digit is ~0.000122, then the usual number of proper digests found would be *1039380*0.000122=126,80436≈127,* which is the amount *z*.

Each blockchain block produced must, in addition to other elements, comprise the root hash of a hash tree containing, in its data blocks, *≥z* approximate pairs of tickets pointing at the same last block's root block value as the winning pair of tickets which allows for the block's production. Let's suppose that, in accordance with the exemplary scenario we have used throughout the present subsection, during the timeframe *ΔtB2* designated for block production at the height h_2, a network participant has found a winning pair of tickets which allows for the production (*vide* next item) of the block *B2* at the height h_2. Each of the winning tickets of the pair was generated (as in item 5 above) through serially rehashing a digital signature from a message pertinent for the consensus at the height h_2 and which points (in the message's signed data part) at the last block's root block value, which we'll refer to as *rbv*. In accordance with the requirement introduced by the value *z*, in order to be considered as a valid block, like any other block produced within the epoch *e*, *B2* would have to contain the root hash of a hash tree comprising at least *z* (*i.e.*, *≥127*) approximate pairs of tickets, where:

A.) Each approximate pair of tickets comprises two tickets, where each of the tickets was generated through serially rehashing a digital signature from a message pointing at the same last block's root block value as the message whose digital signature was rehashed in order to generate the other ticket of the pair.

B.) Said last block's root block value is the same as the root block value (in this case, *rbv*) pointed at in the messages whose digital signatures were rehashed in order to generate the winning pair of tickets which allows for the block's (in this case, *B2*'s) production.

C.) Each approximate pair of tickets is supplemented with data that allows for an easy validation of its correctness, that is:

- the messages, generated and signed as in item 3 above, whose digital signatures were serially rehashed in order to obtain both of the tickets of a pair,

- two Merkle proofs, proving the inclusion of the two staking transactions (together with the amounts of coins frozen and the public participation keys included in them) which have allowed for the generation of the above-mentioned messages, in the staking state tree whose root hash is contained in the preceding epoch (in this case, in *e-1*).

Using these data, the validity of each approximate pair of tickets will be able to be independently verified by every network participant analogously to how a winning pair of tickets is verified, which is described in the next item.

D.) The same approximate pair of tickets can't be included in a single hash tree twice.

In conclusion, *B2* needs to comprise the root hash of a hash tree with *≥127* valid approximate pairs of tickets, where each pair (indirectly) points at the same last block's root block value as the winning pair of tickets which have allowed for the production of *B2*. The producer of *B2* would be aware of the approximate pairs of tickets which fulfil the criteria mentioned above, as they would be revealed to the producer during the process of concatenating and hashing pairs of tickets, detailed in items 8 and 9 above.

Statement 1. Given that a block *B*, *to-be-produced* during an epoch *e*, needs to comprise a set *S* of *≥z* approximate pairs of tickets with *≥y* leading digits from the set 1-8 in each concatenated pair's hash digest, where:

A.) the values *z* and *y* were calculated in the preceding epoch *e-1* in the manner described above,

B.) every ticket belonging to a pair from *S* needs to be generated through rehashing a digital signature from a message pointing at the same last block's root block value as the messages which allow *B* to be produced,

B can be produced within the next timeframe, with a high degree of certainty, by an honest majority of *≥51%* of all the participants, weighed by their stake, eligible for block production during *e*. An attacker owning *≤45%* of all staked coins would be able to produce *B*, on average, after a relatively high amount of timeframes.

Examples. Each timeframe is scheduled as a one-minute time interval designated for block production at a particular blockchain height. As described under letter H.) of the next subsection, in case that no block was produced at a height *h* and the timeframe designated for block production at *h* has finished, the process aimed at choosing the block producer at the subsequent height *h+1* can start with the participants generating signed messages which include the same last block's root block value as the one used during the consensus at *h*, that is a root block value of a block produced at the height *h-1*, but this time concatenated (in accordance with item 2 of the present subsection) with a different blockchain height (*i.e.*, with *h+1* instead of *h*). In this case, the changed (incremented by one) blockchain height in the messages' signed content serves as a new source of pseudorandomness for the whole process of block producer selection at the subsequent height *h+1*. Consequently, each one-minute timeframe constitutes a separate attempt at finding winning/approximate pairs of tickets, where a failure of block production at a specific height (thus leaving the height without a block produced) doesn't affect the probability of producing a block at the subsequent height.

Therefore, each timeframe can be seen as a biased coin flip[10]. Let *Ms* denote a success event consisting in valid block production by an honest majority of the participants and *As* denote a failure event consisting in valid block production by an attacker owning a minority of staked coins.

We can demonstrate the truthfulness of the statement above with the help of some exemplary calculations. Firstly, we will calculate the probabilities *P(Ms)* and *P(As)* in conformity with the exemplary scenario of the production of *B2* described above. According to what was mentioned there, the total number of staked coins is 2000 and the number of concatenations available through the use of 51% of the total number of staked coins is *1020*1019=1039380*. The values *y* and *z* were calculated as, respectively, 13 and 127. Therefore, *Ms* constitutes the production of a block by a 51% (or more, however, we will use the minimum value of 51% for simplicity) majority and *As* constitutes analogous production by a 45% (or less, however, we will use the maximum value of 45% for simplicity) attacker, where for both parties a block produced must, among other things, contain *≥z* approximate pairs of tickets with *≥y* leading digits from 1 to 8 in each digest of an approximate pair of tickets.

Let *p* be the probability of finding a single approximate pair of tickets at a single blockchain height, which is the same for the honest majority and the attacker; since *y=13*, that is a hash digest of an approximate pair of tickets needs to start with at least *13* suitable leading digits, each from the set of 1-8, and since the first hexadecimal digit has ½ chance of being such a digit, the second hexadecimal digit has, independently, ½ chance and so on, the probability is $(½)^{13}≈0.000122$.

The number of trials, denoted *n*, where each trial consists in hashing a single pair of tickets, for the honest majority equals 1039380 per a blockchain height and, assuming the attacker owns 45% of the stake, *900*899=809100* for the attacker.

Finally, let *X* be the number of successful outcomes consisting in finding an approximate pair of tickets.

The probability of finding a particular number of approximate pairs of tickets can be calculated using the binomial probability formula:

$$P(X) = \binom{n}{X} \times p^X \times (1-p)^{n-X}$$

The binomial coefficient $\binom{n}{X}$ is defined by $\binom{n}{X} = \frac{n!}{X!(n-X)!}$ so the full formula with the coefficient is:

$$P(X) = \frac{n!}{X!\,(n-X)!} \times p^X \times (1-p)^{n-X}$$

Accordingly, the probability of the majority finding exactly one approximate pair of tickets, where *p=0.000122* and *n=1039380*, can be calculated as:

$$P(X=1) = \frac{1039380!}{1!\,(1039380-1)!} \times 0.000122^1 \times (1-0.000122)^{1039380-1} \approx 1.0700246E-53$$

In order for the event *Ms* to occur, *X* has to be equal to or greater than 127. To find the probability of *X≥127*, we need to find the sum of the probabilities for each value of *X* from 127 to 1039380. Let the probability for X=127 be x_{127}, the probability for *X=128* be x_{128}, and so on until $x_{1039380}$.

$$P(Ms) = P(X \geq 127) = \sum_{i=127}^{1039380} x_i \approx 0.5048762$$

During each timeframe, there is ~0.5048762 probability of the majority producing a block. The majority will therefore produce a valid block, approximately, during each second timeframe. Furthermore, even a small increase in the majority's stake would greatly increase the block production frequency.

Now, analogously, for the attacker (where *p* stays the same, but *n=809100*):

$$P(X = 1) = \frac{809100!}{1!\,(809100 - 1)!} \times 0.000122^{1} \times (1 - 0.000122)^{809100-1} \approx 1.3258812E - 41$$

$$P(As) = P(X \geq 127) = \sum_{i=127}^{809100} x_i \approx= 0.0035146465$$

As we have mentioned, each timeframe, associated with a particular blockchain height, can be understood as a biased coin flip and is associated with a particular, known in advance, 1-minute time interval. Let's assume that the attacker tries to fork the chain at a height *h*, while owning 45% of all the stake valid at *h*, in accordance with the above example.

In order to produce a valid block at the height *h*, the attacker needs to produce a block at *h* which comprises ≥*z* approximate pairs of tickets, where each of the pairs (indirectly) points at the same last block's root block value and hashes to a digest with ≥*y* leading digits from 1 to 8. His chance of doing so is ~0.0035146465, *i.e.*, one in ~285. Should he fail, his next attempt would be carried out by trying to produce a block at the subsequent height *h+1*. The attacker would necessarily start by generating messages pointing at the last valid block's root block value, as described in item 3 above. The signed part of each of these messages would contain the same last block's root block value (since no new valid block was produced by the attacker, the same block's root block value would be considered as the root block value of the last valid block), but this time concatenated with the subsequent numerically represented blockchain height *h+1* (as described in item 2 above). Again, the chance of producing a valid block at *h+1* by the attacker would be one in ~285. Should he fail, he would repeat the process of message generation, this time changing, in their signed data part, the numerically represented height from *h+1* to *h+2*. Consequently, on average, he would succeed in a single valid block's production, at each 285th height subsequent to *h*. As we will explain in subsection 4.1. below, such very long intervals between successful block productions make it impossible for an attacker owning *≤45%* to produce a persistent, valid fork.

We can also see that the results are similar with any numbers involved in the calculation. For example, let's suppose that there are 10000 coins staked in an epoch. The majority owns 51% of coins, *i.e.*, 5100, and the attacker – 45%, *i.e.*, 4500. For the attacker, *4500*4499=20245500* ticket concatenations are possible per a height, while for the majority *5100*5099=26004900*. The values *z* and *y* were calculated as, respectively, $z=26004900/(2^{17})= 26004900/131072=198,4\ldots\approx198$ and *y=17* (therefore, the probability of finding an approximate pair of tickets is ~0.00000763).

Calculations for *Ms*:

$$P(X = 1) = \frac{26004900!}{1!\,(26004900 - 1)!} \times 0.00000763^{1} \times (1 - 0.00000763)^{26004900-1} \approx 1.335605E - 84$$

$$P(Ms) = P(X \geq 198) = \sum_{i=198}^{26004900} x_i \approx 0.5212654$$

For *As*:

$$P(X = 1) = \frac{20245500!}{1!\,(20245500 - 1)!} \times 0.00000763^{1} \times (1 - 0.00000763)^{20245500-} \approx 1.26403E - 65$$

$$P(As) = P(X \geq 198) = \sum_{i=198}^{20245500} x_i \approx 4.33832\text{E} - 4 = 0.000433832$$

We can see that the probability of block production, per a timeframe, by the majority didn't change significantly when compared with the previous example, whereas for the attacker it has decreased to approximately one in ~2305.

As mentioned in the previous item, at each blockchain height, ~1-4 blocks (each with a different root block value than the other ones) can be produced, where each of the blocks produced at a given height points at a particular, not necessarily the same as the other ones, last block's root block value. However, since (according to the present item) production of a valid block is possible (in a consistent manner) only if the majority (weighed by their staked coins) of network participants support one and the same last block's root block value, at each blockchain height, only a block or blocks pointing at the same *single* last block's root block value can be validly produced. Assuming that the majority of participants, weighed by their staked coins, don't support (during a single timeframe) in their messages more than one last block's root block value – as explained in section 3, their staked funds get slashed for that – a *single* majority can't support two or more last block's root block values during a single timeframe. This way, each time a block or blocks is/are validly produced at a particular height, only one version of the preceding block (*i.e.*, one and the same last block's root block value) is validated in the block/all the blocks thus produced. That a majority of eligible participants have supported a certain last block's root block value during the consensus within a particular timeframe can be established, through verifying a relatively small amount of approximate pairs of tickets included in a block produced within said timeframe, without the need to know every single supporting message generated and propagated during the timeframe.

In order to incentivise block producers to include in their blocks as many approximate pairs of tickets as possible, the reward for a block's production (coinbase) would slightly increase with the number of said pairs included in the block.

11.) If, during the step described in item 9, a participant finds a winning pair of tickets (which, concatenated, hash to a digest with enough leading digits from 1 to 8) and the first (according to the order in which they were concatenated before being hashed) of the winning tickets from the winning pair was obtained, as in item 5 above, by serially hashing the digital signature from the message signed using the private participation key of the participant, then the participant in question can produce a block.

If a ticket *wt*, obtained on the basis of a message *m* (as in item 5 above), is found by the node *n* (during the concatenation and hashing operations described in item 9 above) to be one of the tickets from a winning pair, where:

A.) *n* has signed the message *m* using the private participation key *priv-key* and

B.) *wt* is the *first* ticket in the winning pair

– then *n* can now compose and produce the block *B2*. The node *n* does that by:

i.) Composing the set of transactions that *n* wants *B2* to contain. As we have described under illustration 2, a block producer can choose to extend, in the block he produces, the transactional history from any of the valid variants of the last block (*i.e.*, of the block whose root block value said block producer, in this case *n*, chose at the beginning of the process described in the present subsection). In our example, *n* chooses, if there are multiple valid variants of the last block (*i.e.*, of *B1*), one particular variant, and extends the transactional history from the variant chosen.

ii.) Adding to *B2* the winning pair of tickets *wt* and *wt'*, where *wt* was obtained (as in item 5) on the basis of the message *m*, the message having been signed (as in item 3) using the private participation key *priv-key*, which is known only to the node *n*. Additionally, *wt'* was obtained analogously on the basis of another message, *m'*, signed using another node's, we'll refer to it as *n'*, private participation key *priv-key'*.

iii.) Adding to *B2* the messages *m* and *m'*, signed using *priv-key* and *priv-key'*, respectively.

iv.) Adding to *B2* the data allowing for an easy identification of the public participation keys *pub-key* and *pub-key'*, which are associated with *priv-key* and *priv-key'*, respectively, in the valid staking transactions which contain them. According to the previous section, the network keeps track of which staking transaction are/will become valid during an ongoing/a subsequent epoch through the use of staking state trees. The node *n* adds two Merkle proofs, one proving the inclusion of *pub-key*, as a part of a valid staking transaction, in a data block of the relevant (that is, the one whose root hash is included in the preceding epoch) staking state tree and another, analogously, proving the inclusion of *pub-key'*. It would be easy for *n* to establish which staking transaction (made by *n'*) contains *pub-key'* by simply scanning all the valid staking transactions contained in the pertinent staking state tree, in order to find the one comprising *pub-*

key'. *Pub-key'* is known to *n* as it was revealed in the digital signature under the message *m'*, which *n* has received from *n'*.

v.) Adding to *B2* a coinbase transaction, rewarding *n* and *n'* equally for the block production. The node *n* has to divide the coinbase equally between itself and *n'*. This requirement makes it equally rational for the nodes in the network to both receive messages, which can be used to obtain tickets (so that a receiver of a given message could check if he can produce a block and earn half of the reward), and send them to other nodes (so that, should a message sent by one node allow for obtaining a winning ticket and block production by another node, half of the coinbase would still be earned by the sender of the message).

In our example, the address of the node *n'* to be rewarded has to correspond with *pub-key'* (so that only *n'* would be able to unlock the funds awarded) – as addresses are derived from public keys. Furthermore, coinbase transactions have a lock-up period of 1000 blocks, during which they can't be spent.

vi.) Adding to *B2* the hash of the last block's, that is *B1*'s, header. If *B1* was produced in many valid variants (analogously to how it was described under illustration 2), one variant should be chosen.

Finally, once the block contents (as described in i.)-vi.)) of *B2* have been generated, the block (as raw data) and the block's header are produced.

The raw data block comprises the elements described in i.)-vi.) above, where the element i.) contains: the root hash of the block's world state tree, the state transitions (the transactions between addresses, included in the block), the staking and unstaking transactions included in the block (in the staking transition tree, whose root hash is a part of the block since the block is produced during the staking part of an epoch; *vide* section 1) and the approximate pairs of tickets that allow for the block's production (as in the previous item).

The block's header is generated as well. It contains all the elements described in i.)-vi.), however, the state transitions, the staking/unstaking transactions and the approximate pairs of tickets are represented only by the root hashes of the respective hash trees which comprise them. The block header is signed by *n* using *priv-key* and broadcast in the network. This way, the signature under the block header will be verifiable without the need to download all the contents of the block.

The correctness of the elements described in i.)-vi.) can be easily verified. It can be checked whether the concatenated pair of the winning tickets *wt* and *wt'* hashes to an output with a suitable number of leading digits from 1 to 8, whether *wt* and *wt'* were obtained on the basis of the messages *m* and *m'*, respectively (and whether they were obtained correctly, that is whether the amounts of coins frozen in the staking transactions associated with *m* and *m'* allow for the number of hashing operations, made as described in item 5 above, necessary to obtain *wt* and *wt'*), whether *m* and *m'* are signed using the appropriate private participation keys *priv-key* and *priv-key'*, whether the public participation keys *pub-key* and *pub-key'*, associated with said private participation keys, are included in the appropriate valid staking transactions, whether the hash tree with approximate pairs of tickets contains enough (correct) pairs, *etc*.

12.) Now, once *B2*, in the form of a block header, is signed and broadcast within the timeframe *ΔtB2*, designated for the production of a block at the blockchain height h_2, it is propagated in the network. The raw data block is broadcast as well. Once the timeframe *ΔtB2* is over, the timeframe for the production of the next block at the height h_3, that is *B3*, starts. The timeframes for block production are known in advance, one per a blockchain height; the participants are able to keep track of the timeframes because, as we have already described in the present subsection, the system clocks of the network participants' devices are synchronised with a bounded clock drift. Now, that is during *ΔtB3,* in order to produce the subsequent block *B3*, all the steps described in items 1-11 above will be repeated analogously. Nodes will start by extracting the root block value from a previous block; given that there is no other valid block produced at the height h_2, they will extract a root block value from *B2*.

What is the root block value of *B2*? It is the hash digest obtained by concatenating the winning tickets from *B2*, that is *wt* and *wt'*, and subsequently hashing *wt#wt'*. Since the root block value of *B2* couldn't be known before *B2* itself was produced, the production of *B3* couldn't start before *B2* became known. This way, each new block becomes a solution to the pseudorandom lottery aimed at choosing the new block's producer and simultaneously generates the root block value needed to initialise the lottery which will choose the producer of the subsequent block; the chain of root block values, each pointing at the producer of the previous block and simultaneously inextricably linked with the one of the next block, becomes the basis of an ordered chain of block producers. We have described how this translates into a functional chain of transactions under illustration 2. Now the concept of a root block value, used throughout this subsection, becomes clear.

Each newly produced block header comprises:

A.) The previous block's root block value, extracted as in item 1 above and pointing at the chosen previous block's producer.

B.) Two messages, generated and signed as in item 3 above by two participants (one of them being the producer of the block in question), pointing at the same previous block's root block value.

C.) Two Merkle proofs, proving the inclusion of the two staking transactions (together with the amounts of coins frozen and the public participation keys included in them) which have allowed for the generation of the above-mentioned messages, in the staking state tree whose root hash is contained in the preceding epoch.

D.) Two winning tickets which, after being concatenated, hash to a digest with a suitable number of leading digits from 1 to 8 (as described in item 9 above). Each of the tickets must be obtainable by serially rehashing a digital signature from one of the messages mentioned in letter B.) above, where the digital signature can be rehashed a number of times not exceeding the number of coins frozen in the staking transaction (mentioned in letter C.) which has allowed for the generation of the message containing the digital signature that is being rehashed. The first winning ticket, according to the order in which they were concatenated before being hashed, needs to be obtainable on the basis of the digital signature generated by the block's producer (*i.e.*, using his private participation key).

E.) A coinbase transaction, where the addresses to be credited are derived from the public participation keys published in the staking transactions mentioned in letter C.).

F.) The hash of the previous block's block header.

G.) The root hashes of the following hash trees:

-A state transition tree (comprising the transactions between addresses occurring in the relevant block).

-A world state tree (comprising the world state of the relevant block).

-A staking transition tree (comprising the staking/unstaking transactions which are being published in the relevant block; this applies only if said block was produced within the staking part of an epoch, as described in section 1).

-A hash tree comprising a suitable number of approximate pairs of tickets, as described in item 10 above.

Furthermore, *if* the block header in question is the first within the non-staking part of an epoch, it needs to contain the root hash of a staking state tree (described in section 1) and, additionally, calculations of values *x*, *z* and *y* (as in items 9 and 10 above).

H.) The digital signature of the block header's producer, which signs all the data described in letters A.)-G.), where the signature needs to be made using the private participation key corresponding to the public participation key (included in one of the staking transactions described in letter C.) above) of the block header's producer.

The block header elements mentioned in letters A.)-E.) and H.) can't be altered by the producer without making the block header itself invalid. However, the elements mentioned in letters F.) and G.), as long as they conform to the protocol rules in general, can be chosen by the producer freely, without invalidating the header. This way, the producer is able to produce many block header variants, each with differing data in the elements mentioned in letters F.) and G.) (which pertain to the transactional history of the blockchain), while each variant has to contain the same data in the elements detailed in letters A.)-E.). Said data identifies the header's producer himself and, indirectly, all the producers which precede him (since any block header's root block value points at the preceding header's one, a tamper-proof chain of root block values is formed). The way this translates into a singular transactional history is described under illustration 2.

2.2. Overview of block production

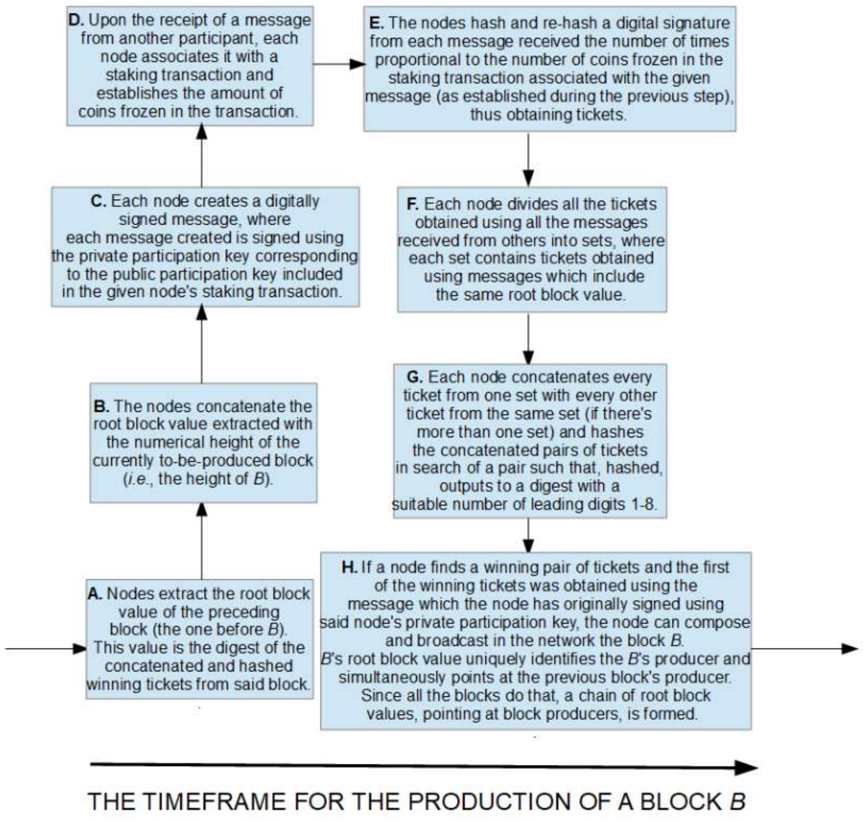

D. Upon the receipt of a message from another participant, each node associates it with a staking transaction and establishes the amount of coins frozen in the transaction.
E. The nodes hash and re-hash a digital signature from each message received the number of times proportional to the number of coins frozen in the staking transaction associated with the given message (as established during the previous step), thus obtaining tickets.
C. Each node creates a digitally signed message, where each message created is signed using the private participation key corresponding to the public participation key included in the given node's staking transaction.
F. Each node divides all the tickets obtained using all the messages received from others into sets, where each set contains tickets obtained using messages which include the same root block value.
B. The nodes concatenate the root block value extracted with the numerical height of the currently to-be-produced block (*i.e.*, the height of *B*).
G. Each node concatenates every ticket from one set with every other ticket from the same set (if there's more than one set) and hashes the concatenated pairs of tickets in search of a pair such that, hashed, outputs to a digest with a suitable number of leading digits 1-8.
A. Nodes extract the root block value of the preceding block (the one before *B*). This value is the digest of the concatenated and hashed winning tickets from said block.
H. If a node finds a winning pair of tickets and the first of the winning tickets was obtained using the message which the node has originally signed using said node's private participation key, the node can compose and broadcast in the network the block *B*. *B*'s root block value uniquely identifies the *B*'s producer and simultaneously points at the previous block's producer. Since all the blocks do that, a chain of root block values, pointing at block producers, is formed.
THE TIMEFRAME FOR THE PRODUCTION OF A BLOCK *B*

Illustration 3. Perhaps, in order to understand how Karada protocol works, it's worth going through each of the steps from items 1-11 of subsection 2.1. above while describing the purpose of a step from the protocol's perspective. This illustration can serve as an additional point of reference. We assume that the steps are performed by the nodes in the network once they have synchronised with the network and established the last valid block, in the manner described in subsection 4.2. below.

A. Once the last valid block is established, the first step for each node to perform consists in extracting said last block's root block value. Each step which follows will be done on the basis of this, and, as will be described in section 3, a single node can't commit to two differing last block's root block values without risking all its funds, locked due to the staking transaction which allows the node to take part in the consensus, being slashed[11]. It follows that each participant, by making a staking transaction, stakes some amount of coins as collateral binding him to commit to only one root block value of the last block during each timeframe designated for a specific height.

B. Next, as in item 2 of subsection 2.1. above, each node concatenates the established root block value, pointing at the producer of the last valid block, with the height of the currently to-be-produced block, which will be referred to as *h*.

C. Subsequently, each single node signs such a message (composed during the preceding step), consisting of two concatenated values, using the private participation key associated with the public participation key that the given node chose to include in the staking transaction which is used to take part in the consensus. This way, each node makes a quasi-statement, which is subsequently sent to other nodes, that it considers such-and-such particular root block value to be the correct root block value of the last block below the currently-to-be-produced height *h*, that is of the block at the height *h-1*. The digital signature under the message of a node easily identifies the funds of the node that are frozen in the node's valid staking transaction. Once a node generates and signs a message, it broadcasts the message into the network.

In an ideal situation, eventually, a message originating from any individual node reaches, during an ongoing timeframe, almost all the other nodes, with the effect that every node gets to know messages generated and signed by almost every other node. Now, each such message serves both as an irreversible statement made by its author on what is the last block's root block value (and so points at its producer) and as a basis to take part in the cryptographic quasi-lottery aimed at selecting the currently to-be-produced-block's producers.

D. As in item 4 of subsection 2.1., each node now associates (through identifying the public key associated with the digital signature which is a part of a particular message and finding, in the pertinent staking state tree, the same key included, as a public participation key, within a particular valid staking transaction) each message received from another node with the valid staking transaction made by the other node and links each message with a particular number of coins frozen in a relevant staking transaction.

E. As in item 5 of subsection 2.1., the more coins are frozen in a staking transaction, the more times the digital signature which is a part of the message linked with the transaction can be rehashed, with the effect that the number of hash outputs generated on the basis of a given message's digital signature matches the number of coins frozen. Each such digest is a ticket, which functions similarly to a ticket in a lottery and increases the chances of becoming a block producer for the node which has made the relevant staking transaction. The more coins a node has frozen in such a transaction, the more tickets will be obtained on the basis of any message said node generates during the period of validity of its staking transaction.

F. According to item 7 of subsection 2.1., the tickets are divided by the nodes into ticket sets, where each set contains only tickets obtained on the basis of messages pointing at the same last block's root block value.

G. According to items 8 and 9 of subsection 2.1., the tickets are concatenated into pairs and hashed. This corresponds to the actual cryptographic lottery taking place, where if any two nodes in the network are aware of the same set of messages, and by extension of the same set of tickets, they will always find the same winning pair(s) of tickets (if any), associated (through the messages based on which the tickets were generated) with the same last block's root block value(s) and the same public participation key(s) of the currently-being-produced-block's producer(s). In other words, each time the cryptographic lottery takes place, both the previous block's producer and the current block's producer are identified by the simple fact of a particular pair of tickets being found as the winning pair. Since it happens each time a block is produced, in effect producers are linked into an ordered chain.

H. According to item 11 of subsection 2.1., if a node finds out that a particular winning pair of tickets contains, as the first ticket of the pair, a ticket obtained on the basis of the message which was generated (on the basis of the node's valid staking transaction) and signed by that node, it means that the node can produce a block. The node composes the block, which includes the data that prove the result of the cryptographic lottery. The data included prove the fact that such-and-such pair of tickets are winning and that they were obtained on the basis of such-and-such messages, in turn generated on the basis of such-and-such staking transactions, all of which can be independently verified by everyone in the network. Then the node signs the block header using the node's private participation key and broadcasts both the signed header and the raw data block. Furthermore, the node needs to include in its raw data block a sufficient amount of approximate pairs of tickets, as mentioned in item 10 of subsection 2.1.

When a valid block is produced and broadcast, its root block value serves as a value required for the purposes of the subsequent cryptographic lottery, that is the one relevant to the block production at the subsequent height. In order to start all the steps described above for a new blockchain height, the participants need to first establish the root block value of the (last) block at the preceding height.

Furthermore, if, at a height *h*, no winning pair of tickets could be found during the cryptographic lottery, once the timeframe for block production at *h* has already ended, the participants can simply skip the block production at *h* and, during the timeframe designated for the subsequent production at the height *h+1*, use the root block value of the last known valid block, that is one produced at the height *h-1* (or even at *h-2*, should there be no known valid block at *h-1*), as the basis for the block production at *h+1*. In such a scenario, as the blockchain height, which is an integral part of the messages generated (as in item 3 of subsection 2.1.) for the consensus during a particular timeframe, changes, so does the outcome of the process, thus potentially allowing for finding winning/approximate pairs of tickets.

In summary, we can divide the process of reaching consensus in Karada protocol into 4 essential steps:

1.) During each epoch, the nodes commit or withdraw funds by staking and unstaking them, respectively. All the staking transactions valid at the end of an epoch *e* which weren't unstaked during *e* become the pool of staked funds valid during the subsequent epoch *e'*. Having established the set of valid staking transactions for *e'*, the nodes come to know the staking transactions which can be used to generate the messages which take part in the cryptographic lotteries, each lottery selecting a block producer (or more than one) at one particular height out of all the blockchain heights covered by *e'*. The nodes also know which timeframe is designated for the block production at each particular height.

2.) During a given timeframe, each node selects the last block's root block value, which points at the last block's producer, and commits to that choice in the message which allows the node to take part in the cryptographic lottery.

3.) Within the same timeframe, the nodes exchange their messages. Using them, they generate, concatenate into pairs and hash tickets, which constitutes the cryptographic lottery taking place. As only the tickets generated using messages pointing at the same last block's root block value can be concatenated into pairs, and due to the fact that the number of pairs (and so the chances of finding a winning/approximate pair of tickets) obtainable from a given set of tickets is exponential in respect to the number of tickets in that set, it's rational for each node to (during each timeframe) generate a message which points at the same root block value of the last block as the messages originating from the majority of the network.

4.) Normally, it is a message from the largest set of messages pointing at the same last block's root block value that has the most chances of becoming the basis for the generation of a ticket belonging to a winning/approximate pair of tickets. Once a block is produced within the timeframe designated, the block's root block value allows the lottery to take place during the subsequent timeframe.

In Karada, making a staking transaction is the investment, choosing (during each timeframe of the staking transaction's validity) the last block's root block value which is chosen by the majority as well increases the chances of becoming a block producer, and (through winning the cryptographic lottery) being able to produce a block and earn a coinbase reward is the profit reaped.

The protocol schedules block production into timeframes, where each timeframe is scheduled for a specific blockchain height at which a block is to be produced (during the timeframe in question). During each timeframe, the nodes generate messages, run (each node independently on its device) the cryptographic lottery on the basis of the tickets obtained from the messages, and eventually, one or more nodes is pseudorandomly selected to produce a block. What makes the nodes follow the timeframes prescribed and generate and broadcast the messages pertaining to a particular blockchain height during the corresponding timeframe?

1.) Let's suppose that during a timeframe tmf_h, scheduled for the block production at a height *h*, a node *n* should, in accordance with tmf_h, generate and broadcast a message which points at the root block value of the block at the preceding height, that is at *h-1*. However, *n* decides to wait with its message's generation and broadcast until the end of tmf_h and broadcasts its message *m* during the subsequent timeframe, tmf_{h+1}. Tmf_{h+1} is scheduled for the production of a block at the subsequent height, that is at *h+1*. We assume that (under normal conditions) at the time when tmf_{h+1} starts, a block B_h at the height *h* was already produced and broadcast during tmf_h. In such a scenario, during tmf_{h+1}, the network participants have already started to generate the messages, signed using their private participation keys, which point at the B_h's root block value. They need to generate said messages if they want to participate in the cryptographic lottery which pertains to the block production at *h+1*, *i.e.*, during tmf_{h+1}.

Let's suppose that *n* sends its message *m*, valid for the height *h*, during the timeframe tmf_{h+1}, scheduled for the block production at *h+1*. Another node, let's refer to it as *n′*, receives from *n* the message *m* and, after concatenation of a ticket obtained on the basis of *m* with another ticket obtained on the basis of its own message, valid for the same height *h*, finds a winning pair of tickets. Now *n′* can produce a block at the height *h* even though it's already the timeframe tmf_{h+1}, scheduled for the block production at the height *h+1*. This way, a block at the height *h*, let's refer to it as $B_{h'}$, is produced by *n′*.

However, $B_{h'}$ has slim chances of being embedded in the blockchain, as the alternative block B_h is already supported by the network through the participant's messages, which were broadcast during the ongoing timeframe tmf_{h+1}. Those messages, already broadcast, point at the root block value of B_h as the root block value of the last valid block, that is of a block at the height *h*. No participant which has already pointed at the root block value of B_h in the message that he broadcast will be able to broadcast another message, this time pointing at the root block value of $B_{h'}$, for the reason that, as will be described in section 3 below, broadcasting two conflicting messages pertaining to the same blockchain height results in losing staked funds through them being slashed. Because of that, $B_{h'}$ will almost certainly be abandoned (by its root block value not being pointed at in the network's messages), thus making late message generation, as *n* did by generating *m* during tmf_{h+1} instead of tmf_h, not rational.

Why would *n* want to wait with the generation and broadcast of its message in the first place? Consider the following situation. During tmf_{h-1}, that is during the timeframe preceding tmf_h, two blocks with differing root block values were produced, that is B_{h-1} and $B_{h-1'}$. Now, during tmf_h, the consensus participants need to choose the root block value of either B_{h-1} or $B_{h-1'}$ to include in their messages supporting some version of the last block. It seems to be rational for each individual participant, such as *n*, to wait with this decision (embodied by broadcasting a message) and receive messages from other participants in the meantime. Should *n* choose to do so, *n* would be able to measure the root block value of which block, B_{h-1} or $B_{h-1'}$, gets the most support in the network's messages during tmf_h (as will be described in subsection 4.2. below, the method of choosing which last valid block's root block value to support during any given timeframe is the same for each node; however, the outcome, in terms of the whole network, can't be estimated by any individual participant beforehand, as it can always be influenced by factors such as, *e.g.*, block propagation latency). Having measured this, *n* would commit in its message, pertaining to the height *h*, accordingly (so that the message would correspond to those of majority; it's rational as the larger the ticket set, exponentially the higher the chances of each ticket within the set becoming a winning ticket, as we have discussed in item 8 of subsection 2.1.). However, the longer *n* waits with its decision and message broadcast, the higher the probability that, even should the *n*'s message become the basis for a winning ticket at the height *h*, allowing *n* to earn a coinbase reward, the block at *h* thus produced by *n* wouldn't be embedded in the blockchain, for the same reasons as mentioned in the previous paragraph.

2.) Let's consider the reverse scenario. The node *n* broadcasts its message pertaining to the height *h* before *tmf*$_h$, that is during *tmf*$_{h-1}$. This way, *n* broadcasts a message relevant at *h* one timeframe earlier than the timeframe scheduled for the height *h*. The node *n* doesn't get any advantage this way. Furthermore, the earlier *n* broadcasts its message, the higher the chances that the root block value of a block at *h-1* (should there, eventually, be two or more blocks produced at this height to choose from) which will be supported, in their messages, by a majority of the network during the subsequent timeframe *tmf*$_h$ is not known yet. Should *n* point in its message at the root block value of a block at the height *h-1* different than the root block value that the majority will point at in their messages later on (that is, during *tmf*$_h$), *n* will just lose an opportunity to efficiently take part in the cryptographic lottery during *tmf*$_h$ (as no second message which pertains to *h* could be broadcast by *n*, due to the fact that this would cause the broadcast of conflicting messages and, as will be described in the subsequent section, slashing the frozen funds of *n*).

This way, adhering to the timeframes for block production is rational. In general, the likelihood of finding a block with the required number of suitable leading digits in its root block value is exponential in respect to the number of participants, weighed by their stake, pointing at the same last block's root block value in their messages. It is the synchronisation of their operation in time (in the above-mentioned way) that allows the participants to make their messages point at the same last block's root block value.

3. Slashing

In Karada, each of the participants commits some amount of funds as collateral (why staking transactions function as collateral is explained in the present section) and, during each timeframe scheduled for block production: 1) decides on *only one* last block's root block value and 2) lets this decision, through signing and broadcasting a message associable with his staking transaction, be known to the entire network (otherwise the participant in question couldn't take part in the lottery aimed at choosing the current block's producer). The cryptographic lottery which takes place on the basis of such messages is simply a series of concatenations and hashing operations which pseudorandomly select the current block's producer and simultaneously confirm the producer of the last block, all of this being governed by timeframes linked with blockchain heights.

This process relies on the assumption that, during the timeframe designated for the production of a block at a given height *h*, a single node *n* generates (in the way described in item 3 of subsection 2.1.) no more than one message signed using the *n*'s private participation key and pointing at the last block's root block value. Why, however, would the node *n*, were *n* to be aware of two last block's versions at the preceding height (that is *h-1*), the first version being *B* with the root block value *rbv* and the second *B'* with the root block value *rbv'*, generate only one message *m* with either *rbv* or *rbv'* (concatenated with the height *h*) in the signed content of the message *m*? From the *n*'s perspective, it seems rational to generate two messages, *m* and *m'*, where *m* contains *rbv* and *m'* contains *rbv'*. This way, *n* would be able to take part in two cryptographic lotteries during the same timeframe (designated for the block production at *h*), one lottery being based on *rbv* and the other one on *rbv'*. Such action would increase the *n*'s chances of becoming a block producer, since *n* would participate in two lotteries simultaneously, instead of just one.

For *n*, there is nothing at stake which would make *n* choose to generate only one message with one last block's root block value (should there be more than one to choose from) per a timeframe. The node *n* can generate and broadcast as many such messages as there are last block's root block value versions, thus increasing the *n*'s chances of becoming a producer. Should every node adopt this strategy, unresolvable forks would form as multiple last block's root block values would be equally supported in the network.

This problem is resolved by a slashing mechanism, which disincentivises nodes from such behaviour. Each time a node *n* generates two (or more) messages *m* and *m'*, both pertaining to a single blockchain height *h* (through comprising *h* in their signed data part), and both signed using the same *n*'s private participation key *priv-key* and associated with the same *n*'s staking transaction *st*, and broadcasts *m* and *m'* into the network, the messages are propagated and eventually reach almost every node. Two messages like this, <u>signed using the same private participation key and generated for the consensus at the same blockchain</u>

height (as the blockchain height to which a given message pertains needs to be a part of the message), but with differing root block values of a last valid block included in them, are called *conflicting messages*.

If the pair of conflicting messages *m* and *m′*, generated by the node *n*, reaches another node, which we'll refer to as *n′*, it is then retained by *n′*. Now, should *n′* become a block producer while the funds in the staking transaction *st* (made by the node *n*) are still frozen, *n′* could penalise *n* by slashing its funds. The block producer *n′* could include in the staking transition tree associated with the block being produced by him a *slashing transaction*, comprising both messages (that is *m* and *m′*), which would constitute a proof that *n* has generated conflicting messages pertinent to the same height *h*. The signed contents of *m* and *m′* would prove that both messages relate to the consensus at the same blockchain height but point at differing root block values of the last valid block. The digital signatures in *m* and *m′* would prove that both messages were signed by the same participant *n*, since only *n* knows the private participation key *priv-key*, which allows to sign messages generated on the basis of the staking transaction *st*.

Such a slashing transaction, included in a block, would have two effects:

1.) It would have the same unstaking effect as if the owner of the staked funds, that is *n*, has unstaked them, which means that the funds frozen in *st* would become unfrozen in the first block of the succeeding epoch. However, instead of being sent back to the address of *n*, they would be sent to the address belonging to *n′*, as the penalising node would be allowed to forfeit and take into possession all the funds frozen by the penalised one.

2.) An unstaking transaction, if any, already made during the ongoing epoch by *n* and attempting to unstake the funds frozen in *st*, would become nullified and replaced by the slashing transaction; furthermore, no unstaking transaction could be made by *n* afterwards (*i.e.*, after the slashing transaction).

Since a slashing transaction needs to be included in a staking transition tree (and the root hashes of these, according to section 1, are part only of blocks produced during the staking part of an epoch), it can't be made during the non-staking part of an epoch. Should a node generate conflicting messages during the non-staking part of an epoch, the relevant slashing transaction could be made only by the producer of the first block of the subsequent epoch. For example, should *n* generate conflicting messages during the non-staking part of an epoch *e*, a block producer *n′*, selected to produce the first block of the subsequent epoch *e′*, would be able to include in the staking transition tree whose root hash is published in his block the slashing transaction penalising *n* analogously to how it was described above. Such a slashing transaction, included in a staking transition tree by the node *n′*, would have the following effects:

A.) If the penalised node *n* didn't make an unstaking transaction (attempting to unstake the funds frozen in *st*) during the just-finished epoch *e*, then the slashing transaction would be included in the staking transition tree associated with the block produced by *n′* and would have the same effect as if the owner of the staked funds, that is *n*, has unstaked his funds. The funds frozen in *st* would become unfrozen in the first block of the epoch succeeding the epoch *e′*. However, instead of being sent back to the address of *n*, they would be unstaked by being sent to the address belonging to *n′*, the penalising node. No unstaking transaction could be made by the penalised node *n* during *e′*.

B.) If the penalised node *n* did make an unstaking transaction (attempting to unstake the funds frozen in *st*) during the just-finished epoch *e*, then that transaction would become nullified and replaced by the slashing transaction, which would mean that the affected funds would be sent to the address of the penalising node *n′* instead of the one belonging to *n* (where both the slashing transaction and the transaction crediting the penalising node's address would take place in the same block).

This way, slashing transactions made within a given epoch wouldn't conflict with the set of staking transactions included in the staking state tree whose root hash is published (in accordance with section 1) in the first block of the non-staking part of the epoch, as no slashing transaction (which needs to be included in a staking transition tree) could be included in any block produced after publishing the root hash of the staking state tree and until the end of the epoch in question (*i.e.*, during its non-staking part).

The first block producer to include, according to the rules mentioned above, two conflicting messages in his block gets to forfeit the funds frozen by the originator of the conflicting messages. It can happen for as long as the funds are frozen in the relevant staking transaction. If a participant doesn't want to get his funds slashed, he should always follow the one height – one message – one last block's root block value chosen rule.

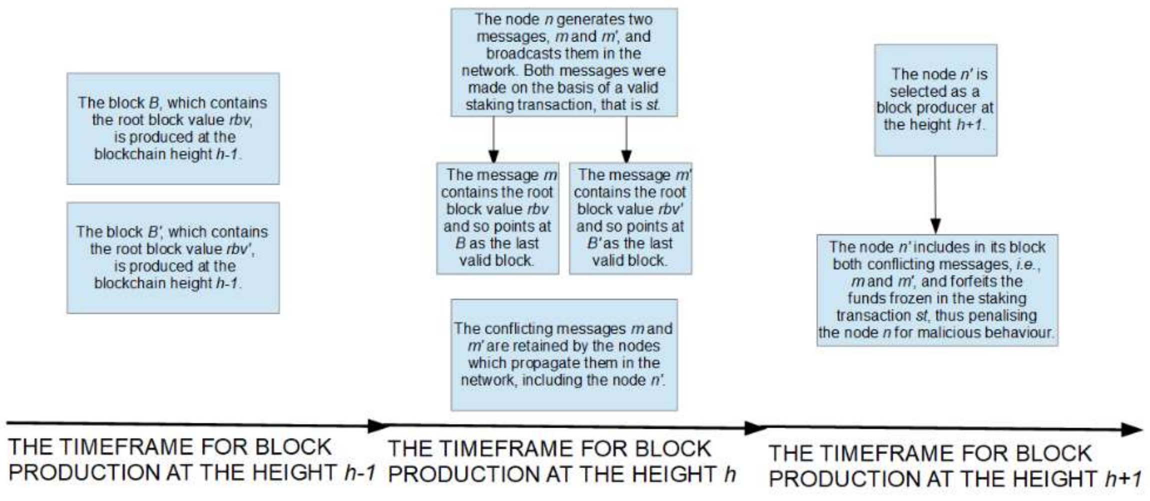

The block *B*, which contains the root block value *rbv*, is produced at the blockchain height *h-1*.
The block *B'*, which contains the root block value *rbv'*, is produced at the blockchain height *h-1*.
The node *n* generates two messages, *m* and *m'*, and broadcasts them in the network. Both messages were made on the basis of a valid staking transaction, that is *st*.
The message *m* contains the root block value *rbv* and so points at *B* as the last valid block.
The message *m'* contains the root block value *rbv'* and so points at *B'* as the last valid block.
The conflicting messages *m* and *m'* are retained by the nodes which propagate them in the network, including the node *n'*.
The node *n'* is selected as a block producer at the height *h+1*.
The node *n'* includes in its block both conflicting messages, *i.e.*, *m* and *m'*, and forfeits the funds frozen in the staking transaction *st*, thus penalising the node *n* for malicious behaviour.
THE TIMEFRAME FOR BLOCK PRODUCTION AT THE HEIGHT *h-1*
THE TIMEFRAME FOR BLOCK PRODUCTION AT THE HEIGHT *h*
THE TIMEFRAME FOR BLOCK PRODUCTION AT THE HEIGHT *h+1*

Illustration 4. According to this scenario, which happened during the staking part of an epoch, the node *n*, during the timeframe designated for block production at the height *h*, broadcast two conflicting messages. One of the messages contained the root block value of the last block *B*, and the other contained the root block value of *B'*. Another node, *n'*, was selected as a block producer during the subsequent timeframe and included the pair of conflicting messages in the staking transition tree associated with its block, proving the malicious behaviour of *n*. The node *n'* thus penalised *n*, which entitled *n'* to forfeit the funds frozen in the valid staking transaction made by *n* by having them unstaked, in the first block of the succeeding epoch, and simultaneously credited to the address owned by *n'*.

Another (other than broadcasting conflicting messages) undesired behaviour is producing two or more block variants by a single producer, where each of the variants contains a different transactional history, as we have described under illustration 2. We can disincentivise nodes from behaving in this way by using an analogous slashing mechanism. Let's suppose that a node *n* has produced, at a height *h*, two differing block variants, *B* and *B'*, where each of the variants comprises a different transactional history. The signed block headers of *B* and *B'* were broadcast into the network by *n*. The *conflicting block headers* of *B* and *B'* were retained by the nodes which have propagated them, one of these nodes being *n'*. As we have mentioned in item 11 of subsection 2.1., coinbase transactions are subject to a lock-up period of 1000 blocks. Should *n'* become a block producer of any of the 1000 blocks subsequent to *h*, *n'* would be able to include in its block, provided none of the preceding producers did this already, the two conflicting block headers of *B* and *B'*. Since both block headers comprise the same winning pair of tickets and are signed using the same private participation key, but contain differing root hashes of world state/state transition/staking transition/other trees (or other data), they are conflicting. By doing the above, *n'* would be able to forfeit the funds locked in the relevant coinbase transaction (which would normally go to *n*).

Finally, broadcasting a message pertaining to consensus at a specific height and pointing at a particular last block's root block value and, in addition, producing a block header at that height which points (in the messages included therein) at another last block's root block value could be penalised in an analogous way. Should a node *n* broadcast a block header *B*, which points at the previous block's root block value *rbv*, and a message *m*, which points at another root block value *rbv'*, where both *B* and *m* pertain to the same height *h*, the coinbase transaction included in *B* could be slashed in any of the 1000 blocks produced subsequently to the production of *B*.

It is perhaps worth mentioning that consensus participants are not penalised for simply being offline (which, as going offline can happen unintentionally, could deter nodes from staking), as is the case in many PoS protocols. Only an intentional malicious action can result in losing funds. This, combined with a short stake withdrawal delay (only a few hours) and a low minimum amount of coins/computational resources needed to take part in staking, would ensure high staking rate and incentivise participants to participate directly, instead of delegating the task to staking pools.

4. Consensus on the chain of blocks

4.1. Establishing the last validly produced epoch

As we have explained in section 1, the Karada blockchain is partitioned into epochs, each epoch made of 200 blocks (100 in its staking part and 100 in its non-staking part). In any given epoch, the 101st block contains the root hash of a staking state tree, which, in its data blocks, includes the staking transactions valid for block production in the subsequent epoch. For any node, synchronisation with the network starts with determining the last validly produced epoch. In order to establish, during a node's *n* (first) synchronisation with the network, the last validly produced epoch, *n* performs the following steps:

1.) Having downloaded the genesis block, which includes the staking transactions valid for block production in the very first epoch *e1* (exceptionally, the stake valid for the first epoch is included in its first block, *i.e.*, the genesis block; for every other epoch, the valid stake is determined by the staking state tree whose root hash is contained in the 101st block of the epoch which precedes it, as described at the end of section 1), *n* comes to know the set S_1 of staking transactions, valid for block production in *e1* (and, implicitly, the number of coins staked, corresponding with S_1). The node *n* obtains 3 values associated with *e1*: *x*, *z* and *y*. The value *x* is calculated analogously to how it was described in item 9 of subsection 2.1. Using the number of coins staked in S_1 and, consequently, the maximum number of ticket concatenations possible during each timeframe of *e1*, the value *x* is established as the minimum number of leading digits from 1 to 8 in a hash digest of a pair of concatenated tickets which qualifies said pair (during the first epoch) as a winning pair. The value *z* is found analogously to how such value was obtained in item 10 of subsection 2.1. The number of ticket concatenations achievable during each timeframe of *e1* with the use of 51% of coins from S_1 allows to determine *z*, which is the minimum number of approximate pairs of tickets included in any given block of *e1* which is needed to qualify said block as validly produced. The value *y* is then inferred, as in item 10 of subsection 2.1., from *z* and denotes the minimum number of leading digits 1-8 in a digest of a pair of tickets which qualifies said pair as an approximate pair of tickets (for the purpose of block production within *e1*).

2.) The node *n* asks each of its peers to provide *n* with any chain (200-block-header long) of block headers known to the peer, which was produced during the epoch *e1*. In addition to any given block header received from a peer in response to this request, *n* asks the peer for all the approximate pairs of tickets which are included in the hash tree with approximate pairs of tickets associated with the block whose header was received (as in item 10 of subsection 2.1.). The inclusion of a particular approximate pair of tickets in the hash tree with approximate pairs of tickets can be verified through a Merkle proof leading from the data block containing the pair to the tree's root hash (which is a part of the header of the blockchain block in question). Having determined the values *x*, *z* and *y*, valid for the purpose of verification of any block header produced in *e1* and provided to it by its peers, the node *n* verifies each received block header of a block produced in *e1* in terms of whether or not:

i.) generally, the block header was produced in conformity with the protocol rules,

ii.) specifically, the winning pair of tickets which allowed for the block header's production is valid and has $\geq x$ leading digits 1-8 in its hash digest; furthermore, the hash tree with approximate pairs of tickets (whose root hash is included in the header) has $\geq z$ valid pairs, each with $\geq y$ leading digits from 1-8 in its digest.

If each block header from a given chain of 200 block headers produced within the epoch *e1* adheres to the requirements described in i.) and ii.) above, the node *n* verifies how many timeframes it took to produce the chain. Each block is associated with a timeframe; as described in item 2 of subsection 2.1., concatenating a numerically represented timeframe with the last block's root block value is one of the first steps involved in every block's production. With each passing timeframe, this numerical value is incremented by one; the node *n* subtracts the number of the timeframe associated with the 1st block header of the verified chain from the number of the timeframe associated with the 200th block header of said chain; the result is the number of timeframes it took to produce the chain. If the result is lower than or equal to 5000, the chain is accepted; if the result is higher than 5000, the chain is rejected by *n* as invalid.

In item 10.) of subsection 2.1., we have presented some calculations related to the possibility of block production by an attacker which owns *≤45%* of valid stake and concluded that, although such an attacker could produce blocks, it would be possible only intermittently and with large amounts of timeframes between the blocks produced. The *45%* safety threshold was chosen by us on the basis of an extremely high improbability (as we will argue here) of simulating a self-consistent, persistent fork using *≤45%* of valid stake. The reasoning presented in item 10.) of subsection 2.1. was as follows: in order to simulate a valid (*i.a.*, comprising *≥z* approximate pairs of tickets, where each pair hashes to a digest with *≥y* leading digits from 1 to 8) block, an attacker needs to attempt this at least hundreds of times, where each attempt requires incrementing the number of the to-be-simulated-block's timeframe by one.

As we will see below, the crucial data needed to securely determine the canonical version of an epoch is the root hash of the staking state tree from the 101st block of the immediately preceding epoch. An attacker which would like to change the root hash of a staking state tree in the 101st block header of a given epoch *and* be able to present (to other nodes) this epoch as completely produced (*i.e.*, comprising 200 valid blocks, presented in the form of block headers) would need to simulate at least the epoch's block headers from the 101st to the 200th, that is, 100 headers. According to what was explained in the preceding paragraph, at least tens of thousands of timeframes would need to be *used* (by incrementing the number of the timeframe associated with the block preceding the first block header simulated by the attacker tens of thousands of times) to allow the attacker to generate a valid 100-block-header long blockchain extension; however, according to the requirement introduced above, only an epoch which was produced within up to 5000 timeframes is acceptable in the network; therefore, we believe that this kind of an attack can be safely ruled out as implausible. On the other hand, as calculated in item 10 of subsection 2.1., an honest majority of *≥51%* (weighed by stake) participants can produce a valid block, on average, at least every second timeframe (therefore, producing a 200-block header long epoch within 5000 timeframes would never be a problem for such a majority). Another option for an attacker to alter the root hash of the staking state tree in a given epoch would be to buy/steal or acquire in some other way the private participation keys used to sign the block headers originally and *honestly* produced by an honest majority, at least from the 101st to the 200th of said epoch. As each block header points at the previous block header's hash digest, only owning the private participation keys used to sign all the headers from the 101st to the 200th of said epoch would allow the attacker to simulate the chain of said headers which would alter their contents (most importantly, the root hash of the staking state tree in the 101st header) *while* the outcome would still be an unbroken chain of valid block headers, each pointing at the digest of the preceding header. Should even a single private participation key be missing, the attacker wouldn't be able to alter the stake valid for the subsequent epoch; therefore, we believe that this kind of an attack can be ruled out as implausible as well.

As we have discussed under illustration 2, an unbroken chain of root block values can potentially be associated with some blocks produced in multiple differing variants (of their transactional contents); however, even a single tie-breaking (*i.e.*, honestly produced in one variant) block resolves the question as to which variants of the blocks which came before it are valid. Consequently, if even a single block after a given epoch's 101st block is produced honestly in one variant, there will be only one valid version of the crucial 101st block of the epoch available (and of the root hash of the staking state tree included therein). <u>In conclusion, we can safely assume that as long as no attacker owns *>45%* of the stake valid for block</u>

production during a particular epoch, any valid and complete (*i.e.*, comprising 200 blocks/block headers) version of this epoch will contain in its 101st block header the same root hash of a staking state tree (and, in consequence, the same set of staking transactions will be valid for block production in the subsequent epoch) as any other such (*i.e.*, valid and complete) version.

3.) Having established, as in the preceding item, the block header of the 101st block of the canonical version of *e1*, the root hash of the staking state tree from the block header and the number of coins corresponding to all the staking transactions contained in the staking state tree become known. Additionally, the values *x*, *z* and *y*, valid for block production in the succeeding epoch *e2*, were calculated in said block header. Knowing all of this, the node *n* determines the valid version of *e2* analogously to how (as in the preceding item) the valid version of *e1* was determined (with the only difference being that *e1*, as the very first epoch in existence, contained the stake and the values *x*, *z* and *y*, valid for its verification, in the genesis block, not in the 101st block of the preceding epoch – as there is none).

During the determination of the canonical version of *e2*, each approximate pair of tickets included in a verified block of *e2* must correspond to two appropriate staking transactions contained in the staking state tree whose root hash is included in the 101st block header of the canonical version (which the node *n* has already established) of the preceding epoch *e1* (*vide* the last paragraph of the present item). Having determined the canonical version of *e2*, the root hash of the staking state tree from the 101st block header of *e2* and the corresponding amount of coins staked, as well as the values *x*, *z* and *y* (this time valid for block production in the next epoch, *i.e.*, *e3*) are used to establish the canonical version of *e3*. The process is reiterated for each subsequent epoch.

Should it occur to a node that a particular version of an epoch *e*, determined to be canonical, starts with a block header *B* which doesn't correspond to what was established as the last block header (referred to as *B-1*) of the canonical version of the preceding epoch *e-1*, so that the root block value of *B* isn't based on the root block value of *B-1* (or, perhaps, the hash digest of *B-1*, published in *B*, doesn't correspond to *B-1*), the node would need to retrieve (from its peers) an alternative last block header(s) of *e-1* (or alternative variant(s) – understood as defined under illustration 2- of last block header(s) of *e-1*) so that, in the end, the block headers of *e-1* and *e* would form an uninterrupted chain adhering to the protocol rules (*e.g.*, each block header, including *B*, would point at the previous block header's root block value and hash digest).

All that is needed to verify an epoch are the relevant block headers and the associated approximate pairs of tickets (whose number is typically a little above 100 per block). This method is relatively computationally inexpensive. In order to be able to perform an epoch verification as above, a (re)synchronising node needs to be provided with (by its peers) relevant block headers of each successively verified epoch and the approximate pairs of tickets associated with said block headers. As approximate pairs of tickets aren't contained in block headers (only a root hash of a hash tree with approximate pairs of tickets is included in a header), some of the nodes in the network (in addition to full nodes, which would host a copy of the entire blockchain) would need to store the approximate pairs of tickets associated with the block headers which they, upon request, provide to their peers for the purpose of facilitating said peers' network (re)synchronisations. As we have mentioned in letter C.) of item 10.) of subsection 2.1., each data block of a hash tree with approximate pairs of tickets, in addition to an approximate pair of tickets, contains two Merkle proofs, proving the inclusion of the two staking transactions (in the staking state tree whose root hash is included in the preceding epoch) which have allowed for the generation of the messages on the basis of which the pair of tickets was obtained. Therefore, in order to verify (*i.e.*, confirm that they were generated on the basis of a valid stake) approximate pairs of tickets from the blocks of a particular epoch, there is no need to know all the staking transactions included in the staking state tree whose root hash is contained in the preceding epoch; it can be done using relevant Merkle proofs, leading from the pertinent staking transactions to the root hash of said preceding epoch's staking state tree.

4.) If, after determining that a given epoch, *e.g.*, *e245*, was completely and validly produced and no subsequent, completely and validly produced, epoch (in accordance with our example, that would be *e246*) can be found, it means that the last epoch established (in our example, *e245*) is the last validly produced epoch. The staking state tree whose root hash is published in the 101st block header of said last epoch contains the staking transactions valid for block production in the epoch which is being produced (*i.e.*, its blocks are being produced in the network and their production has not yet finished; in our example, it's *e246*).

The abovementioned steps can be performed, analogously, when resynchronising with the network; a resynchronising node would then start the process from the last known to it validly produced epoch.

What would happen, however, if an attacker would manage to take possession of *>45%* of the stake valid for block production in a given epoch *e* and, subsequently, simulate a self-consistent version of *e*? Let's suppose that the canonical (*i.e.*, originally produced by an honest majority) version of the epoch *e,* denoted as v_h, was produced in the network; after some time has passed since the production of v_h, the attacker, having appropriated a sufficient amount (weighed by corresponding stake) of private participation keys valid during *e*, simulated a different (and self-consistent, *i.a.*, produced within 5000 timeframes) version of the epoch *e*, referred to as v_m. How is a node *n*, while (re)synchronising with the network, to measure, in terms of their canonicity, the two versions of the epoch *e*, *i.e.*, the maliciously simulated v_m and the honestly produced v_h, and choose which one is to be followed? Faced with said two self-consistent versions of *e*:

A.) The node *n* extracts from each single block belonging to v_h a hundred approximate pairs of tickets. The pairs extracted from a given block *B* (from v_h) are the *top hundred* approximate pairs of tickets, out of all the pairs in *B*, whose hash digests start with the *most* leading digits 1-8. This process is reiterated for every block of v_h from the 1st one up to the 200th. The same operation is performed, analogously, for v_m. In effect, two sets, each comprising two hundred subsets (where each subset, in turn, includes a hundred approximate pairs of tickets), referred to as S_h and S_m, are obtained. Each subset from S_h includes a hundred approximate pairs of tickets from a particular block from the 1st to the 200th block of v_h, whose hash digests start with the most, as compared with the other approximate pairs from the same block, leading digits 1-8; the same is true, *mutatis mutandis*, for S_m. In total, S_h comprises twenty thousand approximate pairs of tickets from v_h and S_m includes twenty thousand such pairs from v_m.

B.) Each of the two sets is measured in terms of the number of leading digits 1-8 in the digests of the approximate pairs of tickets belonging to the set. If the digests of the approximate pairs of tickets belonging to S_h contain, in total, more leading digits 1-8 than is the case for S_m, then the node *n* chooses v_h as the canonical version of the epoch *e*; and *vice versa*.

Statement 2. As long as an attacker doesn't possess private participation keys associated with an amount of stake, valid for block production within an epoch *e*, which is higher than *m-(19%*m)*, where *m* stands for the amount of stake originally used (on average) for block production within *e* by honest and non-faulty participants in relation to the total amount of stake valid during *e*, the attacker won't be able to simulate such a version of *e* that could fool a (re)synchronising node into accepting said version when said node would verify it according to the method described in letters A.) and B.) above.

Examples. We can demonstrate the truthfulness of this statement with the help of some exemplary calculations. Let's suppose that the stake valid for block production during an epoch *e* equals 5000 coins, out of which 80%, *i.e.*, 4000 coins, were originally used to produce blocks in *e* by an honest and non-faulty majority of participating nodes *M*. After some time has passed since the epoch *e* was thus produced, an attacker *A* managed to accumulate (*e.g.*, through buying or stealing them), private participation keys representing *80%-(19%*80%)=64,8%* of the stake valid for block production in *e*, *i.e.*, 3240 coins. The attacker wants to simulate a version of the epoch *e* which would be accepted by a (re)synchronising node as the canonical one (while the node would compare it, in accordance with the method described above, to the version produced by *M*).

At each height of *e*, *A* can obtain *3240*3239=10494360* ticket pairs and *M* can obtain *4000*3999=15996000* ticket pairs (and, subsequently, their hash digests). A number of trials, each consisting in hashing a concatenated pair of tickets during block production at a single height, is denoted *n*, where *n=10494360* for *A* and *n=15996000* for *M*. Let *p* be the probability of finding a digest of a pair of tickets with *≥18* leading hexadecimal digits 1-8, *i.e.* (as a digit has ½ chance of being a digit 1-8), *p=(½)*^{18}*=0.000003814697265625*. Let *X* denote a number of successes, where a success consists in finding a pair of tickets which hashes to a digest with *≥18* leading digits 1-8. The expected number of successes (at a single blockchain height) for *M* can be calculated using the formula $E(X) = P(X) \times n.$

$$E(X) = 0.000003814697265625 \times 15996000 \approx 61$$

Using this result as a reference point, we will calculate the probability of *M* finding, during block production at a single height, *≥61* ticket pairs which hash to a digest with *≥18* leading digits 1-8. To calculate it, we will use the binomial probability formula $P(X) = \frac{n!}{X!(n-X)!} \times p^X \times (1-p)^{n-X}$. The probability of finding exactly one such ticket pair is:

$$P(X=1) = \frac{15996000!}{1!\,(15996000-1)!} \times 0.000003814697265625^1$$
$$\times (1 - 0.000003814697265625)^{15996000-} \approx 1.92672E - 25$$

To find the probability of *X≥61*, we need to find the sum of the probabilities for each value of *X* from 61 to 15996000. Let the probability for X=61 be x_{61}, the probability for *X=62* be x_{62}, and so on until $x_{15996000}$.

$$P(X \geq 61) = \sum_{i=61}^{15996000} x_i \approx 0.518043$$

Since an epoch comprises 200 blocks, we can calculate that *M* would, on average, produce an epoch comprising *200*0.518043=103,6086≈104* blocks containing *≥61* approximate pairs of tickets which hash to a digest with *≥18* leading digits 1-8.

Analogously, we will calculate the probability of *A* finding, at a single height, the same number of ticket pairs which hash to a digest with *≥18* leading digits 1-8 as we have calculated as expected for *M*, *i.e.*, *≥61*:

$$(X=1) = \frac{10494360!}{1!\,(10494360-1)!} \times 0.000003814697265625^1$$
$$\times (1 - 0.000003814697265625)^{10494360-1} \approx 1.645726E - 16$$

$$(X \geq 61) = \sum_{i=61}^{10494360} x_i \approx 0.001223401$$

The relation between the probability of finding a pair of tickets which hashes to a digest with any particular number of leading digits 1-8 and the probability of finding a pair which hashes to a digest with any other particular number of leading digits 1-8 is the same for both *A* and *M* (meaning that finding one pair which hashes to a digest with, *e.g.*, 18 leading digits 1-8 would typically co-occur with finding the same amount of pairs hashing to a digest with, *e.g.*, 14 leading digits 1-8, or any other number, for both *A* and *M*). Therefore, if we can find out whether *A* or *M* could generate (during, respectively, malicious block simulation and honest block production within an epoch) more blocks/block headers with *≥61* approximate pairs of tickets which hash to a digest with *≥18* leading digits 1-8, we can infer whether *A* or *M* could generate an epoch comprising approximate pairs of tickets hashing to digests with more leading digits 1-8 in total. Therefore, we can assume that, in order to fool a (re)synchronising node into accepting his simulated epoch version instead of the one produced by *M* (when the node would compare the two epoch versions according to the method described in letters A.) and B.) in the present subsection), the attacker *A* would need to simulate an epoch version with at least 104 blocks (out of 200), each containing *≥61* approximate pairs of tickets which hash to a digest with *≥18* leading digits 1-8 (as this is what *M* would, on average, manage to produce).

As we have established above, the probability of *A* finding a single block with *≥61* approximate pairs of tickets, where each such pair hashes to a digest with *≥18* leading digits 1-8, is approximately 0.001223401.

Since, as described in item 2.) above, only an epoch version produced within 5000 timeframes would be accepted in the network as valid, the number of attempts *n*, each consisting in simulating a block/block header, available for *A* during a single process of epoch simulation, is 5000. The probability *p* of a success, consisting in producing a single block with *≥61* approximate pairs of tickets, where each such pair hashes to a digest with *≥18* leading digits 1-8, is ~0.001223401. The probability of exactly 1 success is:

$$(X = 1) = \frac{5000!}{1!\,(5000-1)!} \times 0.001223401^{1} \times (1 - 0.001223401)^{5000-} \quad \approx 0.01345434628$$

What is the probability of *A* simulating a chain of *≥104* blocks/block headers, each with *≥61* approximate pairs of tickets hashing to a digest with *≥18* leading digits 1-8? To find the probability of *X≥104*, we need to find the sum of the probabilities for each value of *X* from 104 to 5000. Let the probability for X=104 be x_{104}, the probability for *X=105* be x_{105}, and so on until x_{5000}.

$$(X \geq 104) = \sum_{i=1}^{5000} x_i \approx 5.513053296\mathrm{E}-88$$

The attacker *A*, with the probability of success at a single attempt ~5.513053296E-88, in order to succeed, would need to (if it were possible) attempt simulating an epoch more times than there are atoms in the known universe.

The goal of *A* could be, however, not to generate a wholly simulated (by *A*) version of the epoch *e* which would fool (re)synchronising nodes into accepting it as the most canonical but rather to generate only a partially simulated (by *A*) version of *e* which could both fool (re)synchronising nodes into accepting said version as the most canonical one and, at the same time, alter the root hash of the staking state tree from the 101st block header of *e* in order to change the stake valid for block production in the subsequent epoch. Creating the most canonical version of *e* which still has the same (as the version produced by *M*) root hash of the staking state tree in its 101st block header would mean simulating another epoch version which points at the same (as the epoch version originally produced by *M*) set of staking transactions valid for block production in the subsequent epoch, *i.e.*, in *e+1*. In such a situation, even if a (re)synchronising node would be fooled into accepting the attacker's version of *e*, the same stake, pointed at in both the honestly produced by *M* version of the epoch *e* and the one simulated by *A*, would be valid for the purpose of an analogous comparison of canonicity of all the valid versions of the subsequent epoch *e+1*. Changing the crucial 101st block header requires simulating at least the block headers from the 101st to the 200th (of *e*) while, simultaneously, generating the most canonical version of *e*. In accordance with the calculations presented above, the majority *M*, on average, would produce *104/2=52* blocks/block headers with *≥61* approximate pairs of tickets hashing to a digest with *≥18* leading digits 1-8 during block production of the last 100 blocks/block headers of *e*. What are the attacker's chances of, while trying to produce a fork of the originally produced by *M* epoch version which starts from the 101st block header and persists up to the 200th header, simulating 100 block headers with *≥61* approximate pairs of tickets hashing to a digest with *≥18* leading digits 1-8 within 4900 timeframes (as at least 100 timeframes from the 5000 allowed were already used to produce, by *M*, the first 100 blocks of the forked epoch version)?

$$(X = 1) = \frac{4900!}{1!\,(4900-1)!} \times 0.001223401^{1} \times (1 - 0.001223401)^{4900-1} \approx 0.01490228395$$

$$(X \geq 52) = \sum_{i=52}^{4900} x_i \approx 7.821130052\mathrm{E}-31$$

As we can see, it's impossible. Therefore, since even a more canonical (than the version produced by *M*) version of *e* simulated by *A* would need to point at the same set of staking transactions valid for block production in the subsequent epoch *e+1* as the version produced by *M*, the situation (given that the attacker would continue to possess approximately the same share of a valid stake, *i.e.*, also for block production during *e+1*) would simply reoccur. Again, any simulated version of *e+1* would have to point at the same (as the version of *e+1* originally produced by an honest majority) stake valid for the next epoch, *i.e.*, *e+2*, *etc*. Furthermore, a (re)synchronising node would need to, as described in item 3) above, ask its peers for some number of alternative last block headers of *e*, produced originally by the majority *M*, which are different from the last headers in the (more canonical) version of *e* partially simulated by *A* (but don't form an uninterrupted chain with the first block headers of the most canonical version of the epoch *e+1*, as the first block headers of the most canonical version of an epoch are always produced by an honest majority).

We can formulate two crucial conclusions: firstly, according to the calculations in item 10.) of subsection 2.1., as long as ≥51% of nodes, measured by their stake, are honest and non-faulty, block production occurs in a constant and predictable manner; secondly, as long as no attacker owns an amount of stake larger than 81% of the amount used by the honest and non-faulty nodes participating in consensus during any given epoch of the blockchain, every node (re)synchronising with the network is able to autonomously choose the canonical version of the blockchain even in the presence of multiple self-consistent versions of it. Since we have in mind the whole stake valid during any given epoch and belonging to all honest and non-faulty consensus participants (and not to a subset of elected validators), the assumption that no attacker would ever be able to simulate a blockchain version which would outperform the *true* canonical version in terms of its canonicity is very weak (assuming a properly distributed ownership of stake). As we will argue below, this holds true even in the presence of attacks aimed at disrupting the normal traffic of the network. Therefore, under the mentioned assumption, the protocol comes with objectivity.

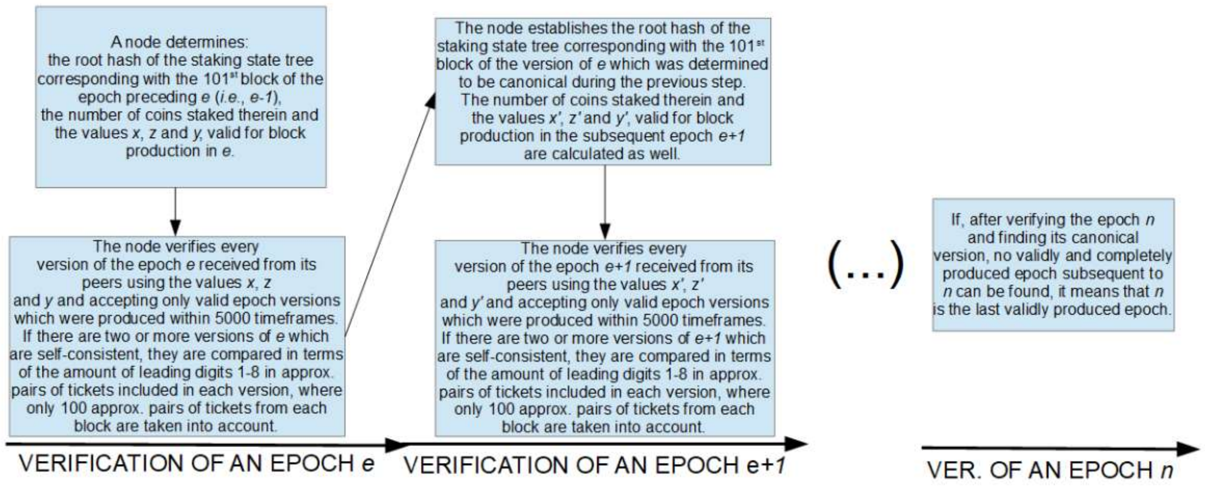

A node determines: the root hash of the staking state tree corresponding with the 101st block of the epoch preceding e (i.e., e-1), the number of coins staked therein and the values x, z and y, valid for block production in e.
The node verifies every version of the epoch e received from its peers using the values x, z and y and accepting only valid epoch versions which were produced within 5000 timeframes. If there are two or more versions of e which are self-consistent, they are compared in terms of the amount of leading digits 1-8 in approx. pairs of tickets included in each version, where only 100 approx. pairs of tickets from each block are taken into account.
The node establishes the root hash of the staking state tree corresponding with the 101st block of the version of e which was determined to be canonical during the previous step. The number of coins staked therein and the values x', z' and y', valid for block production in the subsequent epoch e+1 are calculated as well.
The node verifies every version of the epoch e+1 received from its peers using the values x', z' and y' and accepting only valid epoch versions which were produced within 5000 timeframes. If there are two or more versions of e+1 which are self-consistent, they are compared in terms of the amount of leading digits 1-8 in approx. pairs of tickets included in each version, where only 100 approx. pairs of tickets from each block are taken into account.
(...)
If, after verifying the epoch n and finding its canonical version, no validly and completely produced epoch subsequent to n can be found, it means that n is the last validly produced epoch.
VERIFICATION OF AN EPOCH e
VERIFICATION OF AN EPOCH e+1
VER. OF AN EPOCH n

Illustration 5. The method for establishing the last validly produced epoch is illustrated.

Below, we will discuss the potential impact of attacks attempting to disrupt the normal traffic of the Karada network, which fall into two general categories: network partitionings and Internet shutdowns.

In order for the network to produce blocks in a consistent manner, approx. *≥51%* of nodes (weighed by stake) need to be connected. Should a partitioning event occur, splitting the network into multiple partitions, only if approximately *≥51%* of nodes (weighed by stake) were to stay connected within any given partition, would block production be able to take place in a regular manner. Assuming a large number and broad geographical distribution of nodes, it's practically impossible to achieve the following two goals at the same time: partitioning the network, where individual partitions are capable of block production (and so are large enough) *and* making each partition not comprise even a single node able to communicate with a node from another partition. And if there are at least some common nodes in multiple partitions, they can send blocks/block headers/messages/other data to each other, and, by extension, between partitions, essentially making the whole function like a singular network. Therefore, it is extremely unlikely that a partitioning event could pose a threat to consensus.

Alternatively, an attacker could, through imposing an Internet shutdown, prohibit some participants from participating in consensus during a particular epoch so that, *e.g.*, only the minimum necessary amount of stake, *i.e.*, approx. 51%, would be used to produce the epoch. Afterwards, the attacker could attempt to simulate a version of the epoch which would outcompete the originally produced, by an honest majority, epoch version in terms of its canonicity and, crucially, alter the root hash of the staking state tree included in the 101st block header of the originally produced version. In order to succeed, he would need to control a relatively small amount of the stake valid for block production in the epoch, since (by prohibiting some participants from participating in the epoch's production) he has lowered the amount of stake required to simulate a desired amount of blocks/block headers within the most canonical version of the epoch. In order to prevent this scenario from taking place, the following rule could be introduced.

The amount of leading digits 1-8 in all the root block values of the blocks produced within a given epoch is generally dependent on the maximum amount of ticket concatenations that could be obtained at a single height during the epoch (which is calculated for each epoch as in item 9 of subsection 2.1.), which, in turn, is the number of coins in the stake valid for block production within the epoch squared. We assume that there are no great variations in the percentage of total valid stake which is used for block production from epoch to epoch. Therefore, as long as the network functions normally, the relation between the number of leading digits 1-8 in root block values of produced blocks and the number of staked coins valid for their production should stay approximately constant.

If the total amount of leading digits 1-8 in the root block values of the first 195 blocks produced in a particular epoch *e* decreases, in relation to the maximum amount of ticket concatenations that could be obtained per blockchain height in the epoch *e*, by more than 10% in comparison with the average calculated for the last 100 epochs, then the set of staking transactions valid for block production in *e* remains valid (with no staking transactions being added or subtracted) for block production in each epoch subsequent to *e* until:

i.) a valid epoch with a sufficient number of leading digits 1-8 in its first 195 blocks' root block values is produced

ii.) or 49 epochs are produced without the abovementioned happening.

Starting from the epoch subsequent to *e*, *i.e.*, *e+1*, and until the production of an epoch characterised by either i.) or ii.) occurring, no staking/unstaking/penalising transactions can be included in staking transition trees. The set of staking transactions valid for block production in *e* continues to be valid (and remains unchanged) from *e+1* onwards. When either i.) or ii.) takes place, the network returns to its normal manner of operation, *i.e.*, staking/unstaking/penalising transactions can be included in staking transition trees. Let's suppose that an epoch *en* was produced subsequently to *e*, where *en* is characterised by either i.) or ii.) taking place (*i.e.*, either *en* has a sufficient amount of leading digits 1-8 in its first 195 blocks' root block values or it's the 49th epoch subsequent to *e*); henceforth, starting from the epoch subsequent to *en*, *i.e.*, *en+1*, staking/unstaking/penalising transactions can be included in staking transition trees. Furthermore, the stake valid for block production in *en+1* is the set of staking transactions included in the staking state tree whose root hash is contained in the 101st block of *e*. Such a set of epochs as the set of epochs from *e* to *en* (*i.e.*, a set of epochs for the production of which the same set of staking transactions continues to be valid) is called an *extended epoch*. In summary, the stake valid during an extended epoch can't change until said extended epoch ends.

Let's suppose that, some time after the epochs from *e* to *en* were produced, a (re)synchronising node *n* is faced with two self-consistent versions of the epoch *e*, referred to as V_A and V_B, and needs to compare them in the way described in letters A.) and B.) in the present subsection. V_A was honestly produced during an Internet shutdown, leading to a decrease of stake used in its production, and so has an insufficient amount of leading digits 1-8 in the root block values of its first 195 block headers, therefore being the first epoch of an extended epoch. All the epochs in the extended epoch, starting from V_A, are verified by *n*, while the same stake is valid for the purpose of their verification. Afterwards, the node *n* chooses from the extended epoch either the first epoch with a sufficient amount of leading digits 1-8 in its first 195 block headers' root block values (which is therefore the last epoch of the extended epoch) or, if no epoch like this can be found within 49 epochs subsequent to V_A, the epoch with the most leading digits 1-8 in its first 195 block headers' root block values out of all the epochs belonging to the extended epoch in question. The epoch chosen is called a *comparable epoch*. Thus, the node *n* choses the comparable epoch, referred to as *ec*.

Faced with two self-consistent epoch versions V_A and V_B, the node *n* compares them as described in letters A.) and B.) in the present subsection. However, instead of using V_A for this comparison with V_B, *n* uses the *comparable epoch* which is a continuation of V_A, *i.e.*, *ec* (to compare with V_B). Assuming that no attacker can impose an Internet shutdown (which artificially decreases the amount of stake used for consensus by shutting down a part of the network) affecting the network for *>49* epochs, within an extended epoch, an epoch would have to be produced when the network's normal traffic isn't disrupted and, therefore, using a typical amount of stake and with a sufficient number of leading digits 1-8 in the root block values of its first 195 block headers. Returning to our example, any version of the epoch *e*, including V_B, would be compared by a (re)synchronising node, in terms of its canonicity and according to the procedure described in letters A.) and B.) in the present subsection, with *ec*, which we assume was produced using an approx. typical amount of valid stake. As we have described under statement 2 above, an attacker's main goal, should the attacker want to disrupt consensus within the network, would be to alter the 101st block header of any given epoch, which determines the stake valid for block production in the subsequent epoch. As long as he is unable to do that, no persistent fork can occur. Returning to our example, in order to achieve this goal, an attacker would need to produce such a version of the epoch *e* which simultaneously alters the 101st block header of the originally produced (by an honest majority) version of *e* and is more canonical than *ec* when compared as in letters A.) and B.) in the present subsection. An evaluation of his chances of doing so could be done analogously to the calculations presented below statement 2. Furthermore, a comparison (as in letters A.) and B.) in the present subsection) of two or more extended epochs starting with the same epoch could be performed analogously.

4.2. Establishing the last valid block and taking part in consensus

Network nodes can be divided into two basic categories: those that, having staked some amount of coins in a currently valid (for block production) staking transaction, want to participate in consensus by trying to produce a block (in the manner described in section 2) and those that, while not participating in consensus, want to verify particular transactions/addresses/other data published on the blockchain (which are of interest to them).

A node *n*, which *wants* to participate in consensus, having established the last validly produced epoch *e* in the way explained in the previous subsection, in order to take part in the process (as described in section 2) aimed at block production within the subsequent epoch, *i.e.*, *e+1*, performs the following steps:

1.) The node *n*, using the node's system time as a reference point, determines the current timeframe. As we have described at the beginning of subsection 2.1., nodes have their system clocks synchronised *(e.g.*, using an NTP server). As each timeframe is simply a 60-second increment of time (from the moment the network started to operate), a participant can come to a conclusion on which timeframe is the *current* one by counting the number of such increments from the network's starting time to the present moment. From now on, *n* constantly keeps track of the number of current timeframe as it changes with the passage of time.

2.) The node *n*, having established the last validly produced epoch *e* in the way described in the previous subsection, determines the root hash of the staking state tree included in the 101st block header of *e* and the values *x*, *z* and *y* (the methods of calculating *x*, *z* and *y* are described in items 9.) and 10.) of subsection 2.1.), which are valid for block production within the subsequent epoch *e+1*.

3.) The node *n* asks its peers to provide it with:

A.) All the staking transactions included in the data blocks of the staking state tree whose root hash is included in the 101st block header of the last validly produced epoch, *i.e.*, of *e*. These staking transactions represent the stake valid for block production during *e+1* and *n* will require them in order to be able to validate messages, produced by other participants, analogously to how it was described in item 3.) of subsection 2.1., and received from them in the course of block production within *e+1*.

B.) The world state (*i.e.*, all the addresses and their balances) associated with the 101st block (according to the previous subsection, assumed to be immutable) of the last validly produced epoch *e*.

C.) A chain of blocks as raw data which:

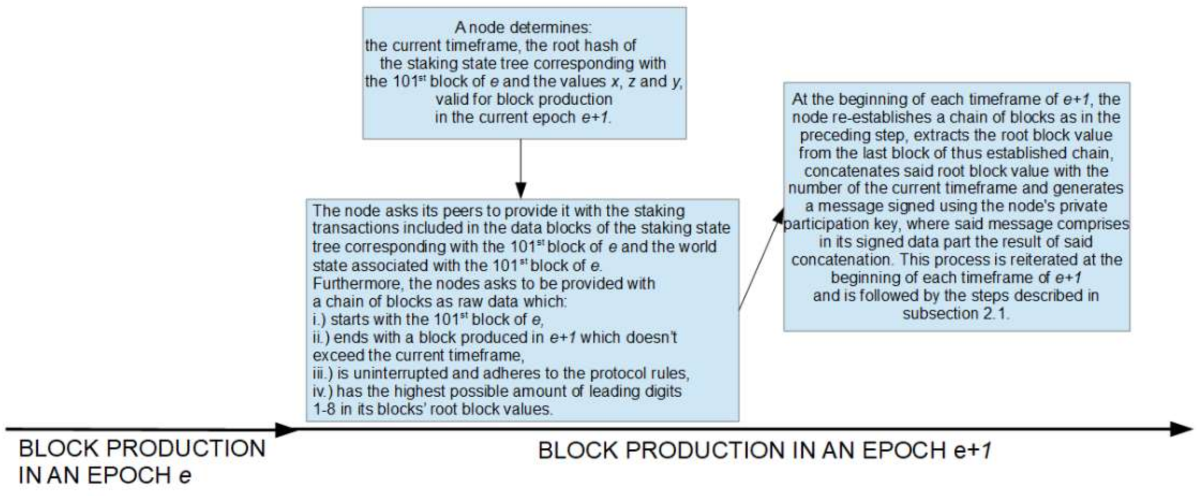
i.) Starts with the 101st block of *e* and ends with a block produced in *e+1* which doesn't exceed the current timeframe, determined as in item 1.) above. Furthermore, the chain of blocks needs to form an uninterrupted chain adhering to the protocol rules (where each block, *i.a.*, points at the previous block's root block value and its header's hash digest and, *importantly*, is associated with a known corresponding block header, generated and signed correctly by its producer) so that the blocks from the 101st of *e* until the last (not exceeding the current timeframe) of *e+1* are valid as a whole blockchain extension *and* the world state, obtained according to letter B.), transitioned by the transactions included in every block of the chain of blocks and, subsequently, converted into a hash tree, corresponds with the root hash of the world state tree in the last block from said chain of blocks.

ii.) Has the highest amount of leading digits 1-8 in its blocks' root block values from all the possible extensions adhering to i.).

Full nodes (and nodes participating in consensus which have already downloaded the data mentioned in letters A.) – C.) above) store the staking transactions as in letter A.), the world state as in letter B.) and the blocks as raw data as in letter C.). Having downloaded the world state and staking transactions from the 101st block of a particular epoch, a node wouldn't need to repeat an analogous process for as long as the node would take part in consensus; knowing all the relevant transitions which have taken place, the node would simply appropriately transition the downloaded world state/staking transactions with each newly produced epoch. Furthermore, once not needed for the purposes of the process described here, irrelevant world state, blocks as raw data and staking transactions could be pruned.

4.) If a chain of blocks fulfilling the criteria mentioned in letter C.) of the preceding item is found by *n*, the last block from the chain, referred to as *B*, is deemed to be the last valid block. The node *n* extracts from *B* its root block value, referred to as *rbv*. Subsequently, at the start of the first forthcoming timeframe, *n* concatenates *rbv* with the number of said timeframe, analogously to how it was described in item 2 of subsection 2.1., and generates a message (with the result of the concatenation in its signed data part) signed using the valid private participation key of *n*, analogously to how it was explained in item 3 of subsection 2.1. This message points at *B* as the last valid block. All the subsequent steps to be taken by *n* for the purpose of taking part in consensus are as described in subsection 2.1. Should *n*, as a result of taking these steps, be chosen as a block producer (as in item 11 of subsection 2.1.), *n* would be able to produce a block (and the corresponding block header) which would point at *rbv* as the root block value of the preceding block (and would include the hash digest of said preceding block's header).

At the beginning of each timeframe, nodes taking part in consensus would try to re-establish what is the last valid block by asking their peers for any known to them chains of blocks fulfilling the criteria as in letter C.) of the previous item. Since the nodes would reiterate the process during each timeframe, the valid blockchain extension from the 101st block of the last validly produced epoch to a block which doesn't exceed the current timeframe with the most leading digits 1-8 in its blocks' root block values would become the common reference point in the network, used for choosing the last block to support through generating a message (as in item 3 of subsection 2.1.). Therefore, nodes (in general) would support the same last block during any given timeframe (whether they would all support the same *variant* of the last block – as there can be more than one block variant, understood as defined under illustration 2 – is irrelevant for the process as long as the block supported is the last block of a particular blockchain extension fulfilling the criteria as in letter C.) of the preceding item). In effect, the more nodes exchange messages during an epoch, each message supporting a last block during a particular timeframe, the more leading digits 1-8 are obtained in the root block values of the blocks produced in the epoch and, additionally, the more approximate pairs of tickets are included in said blocks. Both of these facts entail some crucial consequences for the effectiveness of the methods and calculations presented in the previous subsection. Furthermore, taking into account the arguments and calculations from the previous subsection and assuming that the 101st block of any single validly produced epoch can be considered immutable, since a blockchain extension forming an epoch needs to be produced by an honest majority (as weighed by stake) of participants, where each of them presumably have checked the correctness of each block forming the epoch (as compared with all the preceding blocks from the 101st block of the previous epoch onwards) before supporting it through broadcasting a suitable message, the resulting most canonical version of the blockchain will always be self-consistent.



A node determines: the current timeframe, the root hash of the staking state tree corresponding with the 101st block of *e* and the values *x*, *z* and *y*, valid for block production in the current epoch *e+1*.

The node asks its peers to provide it with the staking transactions included in the data blocks of the staking state tree corresponding with the 101st block of *e* and the world state associated with the 101st block of *e*.
Furthermore, the nodes asks to be provided with a chain of blocks as raw data which:
i.) starts with the 101st block of *e*,
ii.) ends with a block produced in *e+1* which doesn't exceed the current timeframe,
iii.) is uninterrupted and adheres to the protocol rules,
iv.) has the highest possible amount of leading digits 1-8 in its blocks' root block values.

At the beginning of each timeframe of *e+1*, the node re-establishes a chain of blocks as in the preceding step, extracts the root block value from the last block of thus established chain, concatenates said root block value with the number of the current timeframe and generates a message signed using the node's private participation key, where said message comprises in its signed data part the result of said concatenation. This process is reiterated at the beginning of each timeframe of *e+1* and is followed by the steps described in subsection 2.1.

BLOCK PRODUCTION IN AN EPOCH *e*
BLOCK PRODUCTION IN AN EPOCH *e+1*

Illustration 6. Here, the method for establishing the last valid block and taking part in consensus is illustrated.

Should all the blocks (*i.e.*, 200) be produced in the epoch *e+1*, the node *n* would start to participate in consensus in the succeeding epoch, *i.e.*, *e+2*, analogously to how *n* has participated in consensus during *e+1*.

A node *n*, which *doesn't want* to participate in consensus, having established the last validly produced epoch *e* in the way explained in the previous subsection, in order to validate particular transactions/addresses/other data published on the blockchain within the subsequent epoch, *i.e.*, *e+1*, performs the following steps:

A.) Determines the current timeframe as in item 1.) above.

B.) Determines the root hash of the staking state tree included in the 101st block header of *e* and the values *x*, *z* and *y*, which are valid for block production within the epoch *e+1.*

C.) Asks its peers for a chain of valid block headers which starts with the 101st block header of *e*, ends with a block header produced in *e+1* which doesn't exceed the current timeframe and comprises the highest possible amount of leading digits 1-8 in its block headers' root block values. The chain needs to adhere to the protocol rules and form a valid blockchain extension. This step is reiterated at the beginning of each timeframe of *e+1*.

D.) Since, as we have explained under illustration 2, transactional finality in the Karada network is probabilistic, increasing with the production of each tie-breaking block, the node *n* would want to wait until some number of valid block headers were produced subsequently to a block header *B* from a chain of block headers established as in letter C.) before accepting the transactions included in *B* as final. Afterwards, the inclusion of a particular transaction/address/other data in the block (as raw data) which corresponds to the block header *B* can be verified by *n* through a Merkle proof (which can be provided by a full node) leading from the data block of the appropriate hash tree (containing the transaction/address/other data of interest to *n*) to the root hash of the hash tree (*e.g.*, the world state tree) included in *B*.

5. Scaling the network

As we have discussed in the preceding section, a node which is not interested in participating in consensus and only wants to synchronise with the network in order to validate particular transactions/addresses which are of interest to it, can do so in a way which doesn't require excessive hardware resources or high amount of Internet bandwidth. A high-end smartphone with a broadband Internet connection should be sufficient.

However, a node which, after synchronising with the network, wants to take part in consensus on the basis of having made a currently valid staking transaction, needs to perform, during each timeframe, many operations which are costly in terms of computational power/memory/bandwidth needed:

1.) Such a node needs to, besides broadcasting a message pointing at the last block's root block value, receive analogous messages from other participants. The amount of memory/bandwidth needed for this is proportional in respect to the size of the network as measured by the number of participating nodes.

2.) The node needs to, after receiving the mentioned messages, verify each of them in terms of whether or not it was generated on the basis of a valid staking transaction, verify a digital signature from each message and obtain tickets using all the valid messages. The amount of computational power/memory needed for this is proportional in respect to the size of the network as measured by the number of participating nodes for the first two tasks and proportional in respect to the amount of coins staked in the system for the third task.

3.) The node needs to concatenate the tickets obtained into pairs and hash the concatenated pairs in search of winning (and approximate) pairs of tickets. This step is particular in that the amount of computational power/memory needed is exponential in respect to the number of coins staked in the system.

In order to prevent straining the mentioned resources of the participating nodes and, irrelevant of the network's growth, enable participation with the use of a consumer-grade computer/Internet connection, the following protocol optimisation could be implemented.

During each timeframe, participants generate and broadcast messages as described in item 3 of subsection 2.1. A part of each such message contains a digital signature generated using an appropriate private participation key. After their generation, the messages are divided into sets (according to item 7 of subsection 2.1.) and tickets are obtained, concatenated into pairs and hashed.

According to the optimisation proposed here, an additional restriction would be introduced. Two tickets could be concatenated into a pair and hashed, as in items 8 and 9 of subsection 2.1., only if both of the tickets were obtained, as in item 5 of subsection 2.1., by serially rehashing two digital signatures, where the hash digest of the first signature starts with the same hexadecimal digit as the hash digest of the second signature.

Let's suppose that there are two tickets, *t* and *t'*, where *t* was obtained by serially rehashing the digital signature from a message *m* and *t'*, analogously, from a message *m'*. The digital signature part of the message *m* is referred to as *ds* and of *m'* – *ds'*. The digest *d* of the hash operation *h: ds→d* is established for *ds* and the digest *d'* of *h: ds'→d'* for *ds'*. Now, according to the proposed scaling solution, *t* and *t'* could be, as in the steps described in items 8 and 9 of subsection 2.1., concatenated into a pair and, subsequently, hashed only if the digests (*d* and *d'*) obtained by hashing the digital signatures (*ds* and *ds'*) from the two messages (*m* and *m′*) based on which the tickets (*t* and *t′*) were obtained, start with the same hexadecimal digit.

According to our example, if *d=X…* and *d′=X…*, so that both digests start with the same leading hexadecimal digit *X*, then *t* and *t′* could be concatenated into a pair and hashed during the steps described in items 8 and 9 of subsection 2.1; otherwise they (*t* and *t′*) couldn't.

Additionally, the following adjustments would need to be applied to the protocol:

1.) Should a block be produced on the basis of the pair of winning tickets *t* and *t′*, the block would be valid only given that the digests *d* and *d′* would conform to the rule introduced here, that is the leading hexadecimal digit of both *d* and *d′* would need to be the same. The same rule would apply to approximate pairs of tickets.

2.) The application of the requirement that the digests of hashed digital signatures start with the same leading hexadecimal digit decreases the probability of finding a winning or an approximate pair of tickets by a factor of 16 (as, on average, the same number of digital signature digests starts with each particular leading digit and as there are 16 hexadecimal digits in total) for each leading digit which needs to be the same. A proportional decrease in the number of leading digits (from 1 to 8) which is sought for in the digests of (hashed) concatenated pairs of tickets in order to find winning and approximate pairs of tickets, which is (*i.e.*, the numbers of leading digits from 1 to 8) referred to as, respectively, value *x* in item 9 of subsection 2.1. (relevant for winning pairs of tickets) and value *y* in item 10 of subsection 2.1. (relevant for approximate pairs of tickets), needs to be applied in order to preserve the network's liveness. For example, should there be, according to the present optimisation, a requirement that the digests of hashed digital signatures start with the same (one) leading hexadecimal digit, both the values *x* and *y* would need to be decreased by four (*i.e.*, requiring four leading digits, each from 1 to 8, less than normally). The reason for this is that, under the conditions introduced by the present optimisation and in this scenario, the total maximum number of ticket concatenations obtainable during each particular timeframe, denoted as *mTOTAL*, drops by the factor of 16 compared to normal conditions, being *mTOTAL=tTOTAL*(tTOTAL/16)* instead of *mTOTAL=tTOTAL*tTOTAL*, where *tTOTAL* refers to the total number of tickets available during the timeframe. This drops the probability of finding a winning/approximate pair by the factor of 16, so said probability must now be increased by the same factor in order to stay the same. As the values *x* and *y* decrease, with each leading digit that must fall within the set of digits from 1 to 8, said probability by the factor of 2, decreasing the number of digits dictated by the values *x* and *y* by 4 (leading digits) would bring back the balance.

3.) The rationale behind the proposed scaling solution is to divide the tickets which can be, during a particular timeframe, concatenated into pairs and hashed in search of winning/approximate pairs of tickets, into smaller sets. This is achieved by dividing the messages within a single message set (as understood in item 7 from subsection 2.1.) into smaller subsets, where, in each particular subset, a digest of each (hashed) message's digital signature starts with the same leading digit. A rational node which owns a large portion of coins in the system could increase the number of concatenations of its own tickets by simply making one staking transaction which freezes the large portion of coins. Since the leading digit of a digest resulting from hashing a single digital signature is simply one and the same, and because the amount of concatenations within one set of tickets is exponential in respect to the number of tickets in that set, a rational node could increase the number of concatenations allowed for the node's tickets per a timeframe by simply allocating a large amount of coins in a single staking transaction. This way, during each timeframe of the staking transaction's validity, a proportionally large amount of tickets generated on the basis of a single node's message could be concatenated with each other, as by definition they would all be obtained on the basis of a digital signature whose hash digest starts with the same digit. To alleviate this and make the probability of finding both a winning pair of tickets and an approximate pair of tickets unaffected by this strategy, a *relatively low* cap would need to be put on the amount of coins allowed to be frozen in a single staking transaction. This way, all the security guarantees described in this paper would remain unaffected.

Were this optimisation to be implemented, then:

1.) During each timeframe, each participating node would need to receive only the messages, broadcast by the other participants, in which hashed digital signatures (*i.e.*, digests) start with the same hexadecimal digit. This would decrease the amount of memory/bandwidth needed (as less messages would be received by a single participant). Each node would need to inform its peers, at the beginning of each message exchange, messages whose digital signature digests start with which digit it wants to receive.

2.) In turn, the amount of the verifications of messages (in terms of whether or not they were generated on the basis of a valid staking transaction) and digital signatures would decrease proportionally, decreasing the amount of computational power/memory needed. The same applies to obtaining tickets from valid messages.

3.) The number of concatenation and hashing operations performed, and so the amount of computational power/memory required, in search of winning/approximate pairs of tickets would decrease exponentially in respect to the decrease in the number of tickets involved.

Finally, the number of leading hexadecimal digits which, due to implementation of the proposed optimisation, would need to be identical in the hash digests of hashed digital signatures could be adjusted as needed with the growth of the network (the higher the required number, the more optimised the protocol would be from the perspective discussed in the present section) so that the communication complexity required would stay approximately constant.

Some elements of the protocol, however, can't be easily optimised. Every participant participating in consensus during an epoch *e* needs to know all the staking transactions included in the staking state tree whose root hash is contained in the preceding epoch *e-1* (in order to be able to verify each message, generated as described in item 3 of subsection 2.1., received during block production within *e*). Furthermore, he needs to verify each new block of *e* of which he becomes aware, in order to be able to evaluate its correctness before supporting it through generating a message; this, as explained in subsection 4.2., necessitates downloading some number of preceding blocks as raw data and the world state associated with the first block from said number of blocks (all of which, however, can be pruned when not needed anymore).

Conclusion

We have proposed an *objective* and *truly decentralised* Proof of Stake consensus protocol. It allows an arbitrarily large set of nodes to, each time a block is to be produced, take part in the consensus on the canonical chain of blocks which precede the to-be-produced block, where communication complexity required is approximately constant, *i.e.*, *O(1)*. Furthermore, the protocol comes with an effective fork-choice under any plausible conditions. Taken together, these features allow to achieve a *true* decentralisation and a sovereign-grade censorship-resistance[12].

References

[1] Sunny King, Scott Nadal. "PPcoin: Peer-to-Peer Crypto-Currency with Proof-of-Stake." <https://peercoin.net/whitepapers/peercoin-paper.pdf>. August 2012.

[2] Vitalik Buterin. "Proof of Stake: How I Learned to Love Weak Subjectivity." <https://blog.ethereum.org/2014/11/25/proof-stake-learned-love-weak-subjectivity/>. November 2014.

[3] Adam Szepieniec. "Proof-of-Work is objective, Proof-of-Stake is not." <https://bitcoinmagazine.com/technical/proof-of-work-is-objective-proof-of-stake-is-not>. June 2022.

[4] https://en.bitcoin.it/wiki/Proof_of_work

[5] Leslie Lamport, Robert Shostak, Marshall Pease. "The Byzantine Generals Problem." <https://lamport.azurewebsites.net/pubs/byz.pdf>. July 1982.

[6] Satoshi Nakamoto. “A Peer-to-Peer Electronic Cash System.” <https://bitcoin.org/bitcoin.pdf>. October 2008.

[7] Alireza Beikverdi, JooSeok Song. “Trend of centralization in Bitcoin's distributed network.” <http://dx.doi.org/10.1109/SNPD.2015.7176229>. June 2015.

[8] Kenta Iwasaki, Heyang Zhou. “Wavelet: A decentralized, asynchronous, general-purpose proof-of-stake ledger that scales against powerful, adaptive adversaries.” <https://wavelet.perlin.net/whitepaper.pdf>. May 2019.

[9] Nassim Nicholas Taleb. “Fooled by Randomness.” October 2001.

[10] https://en.wikipedia.org/wiki/Bernoulli_trial

[11] Yang Xiao, Ning Zhang, Wenjing Lou, Y. Thomas Hou. “A Survey of Distributed Consensus Protocols for Blockchain Networks.” <https://arxiv.org/abs/1904.04098>. April 2019.

[12] Larry Sukernik. “Sovereign-grade and Platform-grade Censorship Resistance.” <https://sukernik.medium.com/sovereign-grade-and-platform-grade-censorship-resistance-c1fb7b6b492a>. February 2018.

F8E6DFE70C619992DB39AE118C51048AB189603B76D88D946E1507A381A5D5197D7FCC4FA7985DA5CA4BBA07F6907420D03490F3ADF23D6C5CE95A182EA24B0B