Spectrum: Cross-chain interoperability at scale

Spectrum Labs

March 2023

1 Introduction

Following the success of Bitcoin, many blockchain-based cryptocurrencies have been developed and deployed. To meet different requirements in various scenarios, a great number of heterogeneous blockchains have emerged. However, most of the presented blockchain platforms are developed independently, therefore, they are designed for their own use cases and are incompatible with each other. Hence, interoperability between blockchains has become one of the key issues which prevents blockchain technology from wide adoption.

With fair blockchain interoperability users can potentially conduct transactions across different blockchain networks smoothly and without any intermediaries. This guarantees a reduction in the fragmentation of the crypto ecosystem and opens up new horizons and business models. Implementation of the blockchain interoperability protocol is challenging since different blockchains have different security solutions, consensus algorithms and programming languages. An inaccurate solution can potentially increase the possibility of attacks and create management challenges across different connected networks.

The classic cross-chain interoperability solution is a trusted oracle that registers some event on one blockchain and performs the required action on the other. Centralized oracles provide fast and cheap transactions but lack a key feature – decentralization. The liquidity of the protocol built on this approach is custodial which is a centralized approach similar to CeFi when users deposit their funds to an exchange's wallet.

Another common approach involves intermediate network consisting of a fixed number of hand-picked oracles to facilitate data transfer among multiple blockchains. The consensus mechanism in such protocols is usually proof-of-authority or proof-of-stake, hence, the wide range of potential validators are eliminated due to verification procedures or high collateral and network moderation typically carried out by several dozen of rarely alternating nodes. Moreover, a common practice is to store funds transferred between blockchains on some kind of threshold wallets, which are generated by the participants of the intermediate network. This results in all funds being controlled by a fixed group of oracle operators. Such a system is also not truly decentralized.

Regarding the application scenarios, one of the most popular in the existing blockchain interoperability proposals is an atomic token swap. However, atomic token swapping protocols [1] are not self-inclusive enough to complete the tasks of cross-chain decentralized applications with more complex activities than just token exchanges. The reason is that the atomic swapping process does not have the ability to destroy a certain amount of assets in the source blockchain and re-create the same amount on the target blockchain. Moreover, this process always requires a counterparty who is willing to exchange tokens [2].

True blockchain interoperability requires the users and developers have the ability to access information from one blockchain inside another without any additional efforts from a third party. This is a complicated task, thus, before achieving a successfully interoperable multi-blockchain system, many challenges must be overcome, such as scalability when applying to a large-scale scenario [3].

The motivation of this paper is to describe the Spectrum protocol, which provides an open, truly decentralized, secure and scalable cross-chain interoperability solution. The Spectrum protocol is intended for both end-users and developers, who will be able to implement their applications on top of the protocol to widespread the use of blockchain technology in various business areas.

2 Related Work

Blockchain interoperability is promising but still faces various design challenges. There have been many systematic researches regarding this issue and many famous authors have discussed chain interoperability in general. Blockchain interoperability in the literature is usually classified into categories. Buterin [4] suggested centralized, sidechains/relays, and hash locking. Belchior et al. [5] classified it into cryptocurrency-directed approaches, blockchain engines, and blockchain connectors, Wang [6] proposed to group it into chain-based interoperability, bridge-based interoperability, and dApp-based interoperability.

2.1 Existing Interoperability Solutions

In this paper, we want to emphasize the benefits of the decentralization in the chain interoperability mechanism, so we will not include the systematical-level study of all existing approaches and will briefly discuss the classification proposed by Wang.

2.1.1 Chain-based Interoperability

Chain-based interoperability is aimed at public blockchains and uses atomic swaps as its main mechanism to exchange information between different chains. Following the classification, there are three main approaches to implementing chain-based interoperability: hash locking, trusted notary scheme and sidechain.

Hash Locking is an intermediary method that allows to validate or execute blockchain transactions. Hashed Time Lock Contracts (HTLCs) were originally developed as an alternative to centralized switching and can be thought of as a distributed commitment [7] able to fend off Byzantine adversary. It uses a hash time-locked system to lock the transaction [8] which is similar to the concept of the cross-chain atomic swap.

From the technical point of view, the hash locking approach has some significant drawbacks, for example, it must lock some assets during its opening phase for an established transaction channel, thereby creating a race condition and, moreover, the possibility of losing assets if a timeout occurs.

Trusted Notary Scheme is usually considered as the simplest way to achieve cross-chain interoperability. The blockchain notary schemes can provide the functionalities of timed proof of existence, whose proof can be used as further proof of ownership [9]. It doesn't require any additional changes in the underlying blockchains and uses a trusted notary to verify the correctness and integrity of information transferred. A notary can be a stand-alone authority or a group of trusted parties that monitor order books of the connected chains and initiating transactions upon the occurrence of some valid events or requests.

Well-known solutions using this technology are, for example, Herdius [10] and Bifrost [11]. In practice, the most appropriate way to achieve interoperability using a notary scheme is to combine it with other methods, as it is done in the Interledger [12] which combines it with a sidechain.

Sidechain is the most promising approach in this category. Sidechain can add new functionalities, namely, security and privacy to the existing blockchains, making possible a tokens synchronization and additional data transfer between chains [13]. The essential feature of the sidechain is that it's design always takes into consideration the structure and the consensus of each connected blockchain, but none of the mainchains are aware of the presence of a sidechain. This is achieved by utilizing a two-way peg scheme [14] which uses a relay routine for a bidirectional hooking. An important consequence of this approach is that sidechains can be designed in a decentralized manner and have their own consensus protocols.

Using a two-way pegs introduces a level of centralization, however, there are solutions which uses a federated two-way pegs where single authority is replaced by a group of trusted individuals selected in a trustworthy manner.

State-of-the-art sidechain platforms are Loom [15], Liquid [16] and Proof-of-Authority (PoA) networks [17]. There also exists a lot of ongoing projects since this technology is innovative and in demand by the blockchain industry.

Summing it up, a practical way to apply chain-based interoperability methods to current mainstream blockchain systems is to combine them together. Most existing solutions are designed primarily to exchange assets, however blockchain technology is much wider in its applications, and it's better to focus on transaction interoperation between different chains in practical implementations and effectively use all these promising approaches.

2.1.2 Bridge-based Interoperability

Bridge-based interoperability aims to create a connection component between homogeneous and heterogeneous blockchains. Solutions in this field are more complex and typically support the extension of smart contracts which allows developers to design and deploy their own logic thereby expanding the interoperability applications. Bridge-based interoperability can be implemented in two main forms: trusted relay and blockchain engine.

Trusted Relay is a very native approach where trusted parties share transactions between different blockchains. Relay schemes replicate block information of the source blockchain via verifiable smart contracts within a target blockchain to allow the target blockchain to verify the existence of data on the source blockchain without requiring trust in a centralized entity [4]. There are many developing relay schemes: BTC Relay [18], PeaceRelay [19], etc. State-of-the art projects are: Hyperledger Cactus [20], Testimonium [21] and Tesseract [22]. All these solutions support complex use case and are highly usable and reliable, however, still not fully decentralized [6].

Blockchain Engine also provides a relay among the connected blockchains. It is based on a shared infrastructure which support different layers and services including network, consensus, incentive, etc. Requirements of multi-layer supports is essential, thus, most existing blockchain engine-based solutions are still in the stage of proof of concept or under active development. Most significant projects are: Polkadot [23], Cosmos [24], WanChain [25], and ARK [26].

All bridge-based solutions provide convenience for end-users since they don't need to know what happens in the bridge. In general, trusted relays are much more simple and adopted to handle chain interoperability, however, they usually utilize mechanisms similar to the notary schemes which also leads to a certain degree of centralization.

2.1.3 dApp-based Interoperability

Presence of well functioning decentralized applications (dApps) is significant in the blockchain ecosystem, so dApps should be interoperable as well and this is the goal of dApp-based interoperability. Each dApp cannot ensure semantic interoperability, and it's essential to develop the minimum semantic that must be supported by each application to achieve interoperability among dApps. dApp-based blockchain interoperability protocols in the literature are typically classified as: blockchain of blockchains, blockchain adapters and blockchain agnostic protocols.

Blockchain of Blockchains is a platform that allows developers to construct cross-chain dApps where each blockchain functions as an independent one. It is similar to the sidechain idea but differs in implementation. Sidechains are typically aimed at atomic swaps among the homogeneous blockchains where all actions should be coordinated by the mainchain. Blockchain of blockchains solutions typically requires a second layer of blockchain (mainchain) to record the activities that happen on each subchain which can be heterogeneous [6]. There are several projects where blockchain of blockchains concept is applied for different scenarios: Overledger [27], HyperService [28], SMChain [29] and etc.

Blockchain Adapter handles the interoperability by providing an interface for the end-users to runtime selection, smart contracts, etc. Most significant project in this category are PleBeuS [30] and smart contracts *move* protocol [31].

Blockchain Agnostic Protocol: refers to a single platform allowing multiple blockchains to co-exist, enabling cross-chain or cross-blockchain communication between arbitrarily distributed ledgers. Blockchain agnosticism provides its end-users various options to pick their optimal blockchain and provide the capabilities for cross-chain operations. Several agnostic-based technologies have been described in the literature: ILPv4 [32], Gravity [33], SuSy [34] and etc. All these solutions are flexible and has great potential, although most of them are focused on the general design of the prototype and do not grant backward compatibility.

Although dApp-based blockchain interoperability is very promising, most of the solutions in this category are either in early stages of development or lack a practical implementation with criteria to evaluate their effectiveness and efficiency.

2.1.4 Discussion

All of the interoperability approaches described above have their strengths and weaknesses. However, the chain-based interoperability approaches, especially sidechains, are well-established and benefits from extensive research and improvements in design. Sidechains have two important pros that will help to increase the widespread adoption of blockchain technology in various business areas:

- Having their own consensus mechanisms, sidechains can process transactions efficiently and reduce transaction fees for users.
- Taking into consideration the structure and the consensus of each connected blockchain sidechains allow dApps to expand their ecosystem.

The main cons of the existing sidechain protocols is a *centralization* and *poor security guaranties* of the consensus. The disadvantages of centralization are obvious:

- A system is not sustainable when it depends on a single party.
- If the trustee goes down, unfinished swaps can appear frozen halfway.
- A malicious trustee can censor transactions.
- A malicious trustee can perform a man-in-the-middle attack by sending an inaccurate data.

Almost the same deficiencies exist for a semi-centralized protocols, where only a few dozen individuals act as validators. Such "decentralization" is very conditional as it is difficult to meet the requirements to become a validator, furthermore, malicious validators can easily cooperate to successfully attack.

Thus, we come to the conclusion that the scalable practical implementation of the truly decentralized system with a provably-secure consensus protocol is the main step towards wide practical usage of sidechains and bringing their benefits into cross-chain interoperability.

3 Goals

To overcome the outlined problems of the existing protocols the resulted Spectrum protocol must satisfy the following properties:

1. **Decentralization.** The system should be highly decentralized.
2. **Interoperability.** The system should be able to support a large number of heterogeneous blockchains.
3. **Openness.** The system should allow anyone to participate in consensus permissionlessly. Protocol should be fully open-source and all participants will be encouraged by the incentives system.
4. **Consensus Scalability.** The system should be able to operate normally while maintaining sufficiently large consensus groups consisting of hundreds of active validators on each connected blockchain.
5. **Operational Scalability.** The system should scale linearly with the number of supported blockchains.
6. **Security.** The system should be able to withstand Sybil attacks.
7. **Sustainability.** The system should be able to tolerate faults of particular connected blockchains.
8. **Upgradability.** The system should allow to add new blockchains into list of supported over time.

To achieve our goals we will combine the best practices from the approaches that are already in use in the chain-based interoperability solutions. To eliminate the existing bottlenecks, we will supplement them with own-developed improvements which we will emphasize and describe in details in the following sections.

4 System Model

In this section we will describe the main components and general assumptions which is essential to conceptualize and construct the Spectrum protocol.

4.1 Security Model Preliminaries

We consider a semi-synchronous setting where protocol participants have a somewhat accurate common notion of the elapsed time and the network has an upper bound on the message delay, which is not known to the participants and is used as a security parameter.

We also assume that our model operates in the dynamic availability setting [35], where an arbitrary (but upper-bounded) number of the consensus members may not be fully operational, e.g., due to network problems, reboots or software updates that affect some of their local resources including their network interface and clock.

Time and Slots. We consider a setting where time is divided into discrete units called slots. Each slot sl_r is indexed by an integer $r \in \{1, 2, ..\}$, and the ledger associates one time slot with at most one block. All actions of protocol participants necessary for its correct execution are also associated with specific slots. The largest units of time in the protocol are epochs, each consisting of R slots.

Synchrony. A common assumption in known blockchains with a semi-synchronous setting is that all participants are equipped with roughly synchronized clocks and have access to the global clock setup for the synchronization. An existing synchronization techniques are inapplicable to the standard model used for the analysis of Nakamoto-style consensus protocols, thus, there are no strong security guarantees in such a model in the case where the agreement on the current slot is replaced by the assumption of potentially unsynchronized local clocks that proceed at roughly the same speed.

We adopt a provably secure approach to global clock synchronization in the dynamic participation setting [36]. This approach assumes that members of the initial consensus group have access to local clocks and any discrepancies between parties' local time are insignificant in comparison with the slot duration. This is still a typical approach, however, the key feature is an imperfect version of the clock functionality used as a global setup. It allows parties to advance to a next epoch even before every honest member has finished with his current epoch. Once in an epoch, participants synchronize their clocks based on public blockchain data. Therefore, this mechanism ensures that all parties, both active and those who later join the protocol, can synchronize with other participants and will remain synchronized as long as they faithfully follow the protocol.

Random Oracle. We assume that an ideal random oracle is available to each member of the consensus. Random oracle models a function $\mathcal{H} : \{0, 1\}^* \to \{0, 1\}^l$, which samples a uniformly random string from the $\{0, 1\}^l$ in response to some query, while any repeated queries are answered consistently.

Security Configuration. We consider an untrustworthy network environment that allows for adversarial-controlled message delays and immediate adaptive corruption. Namely, we allow the adversary A to selectively delay any messages sent by an honest party for up to Δ^{net} slots and corrupt parties without delay.

The Spectrum protocol is executed by a set of nodes N, where each node $n \in N$:

- Is associated with a unique wallet holding a stake of tokens s_n.
- Is able to generate key-pairs (PK, SK) without trusted public key infrastructure.
- Is able to sign messages $sign : (SK, m) \to \sigma$.
- Is able to verify signatures $verify : (\sigma, PK, m) \to 0|1$.
- Has access to a random oracle $\mathcal{H}$.

We assume that at any time t a subset $V \subseteq N$ of nodes can be controlled by an adversary and are considered faulty. Byzantine nodes can divert from the protocol and collude to attack the system while the remaining honest nodes follow the protocol.

4.2 External Systems

We also assume multiple independent distributed systems $S_1, \ldots, S_K$ with underlying ledgers $L_1, \ldots, L_K$ as defined in [37]. For each ledger $L_k, k \in K$ there is a process P_k that can influence the state evolution of the underlying ledger L_k by committing a transaction

TX_k into it. We extend the model defined in [37] by assuming that all ledgers allow for execution of simple predicates upon validation of transactions: $verify : C \rightarrow 0|1$, where C is a *context* that contains description of state the transaction interacts with. There is also a function $desc : TX_k \rightarrow DESC^{TX_k}$ that maps transaction TX_k to some *description*, e.g. specifying the transaction value, recipient address, etc. For each S_k there is a corresponding functionality unit $\mathcal{F}^k_{\text{ConnSys}}$ that allows any node equipped with the unit to interact with S_k. Each node $n \in N$ is equipped with at least one such functionality unit and at most K functionality units.

4.3 Transaction Ledger

We adopt the definition of transaction ledger from [38]. A protocol Π implements a robust transaction ledger, provided that Π is divided into blocks that determine the order in which transactions are incorporated into the ledger. Each block in this model is assigned to a specific time slot and the ledger must satisfy the following properties:

1. *Persistence.* Once a node of the system proclaims a certain transaction TX as stable, the remaining nodes, if queried, will either report TX in the same position in the ledger or will not report as stable any transaction in conflict to TX. Here the notion of stability is a predicate that is parameterized by a security parameter K_{f}, specifically, a transaction is declared stable if and only if it is in a block that is more than K_{f} blocks deep in the ledger.

2. *Liveness.* If all honest nodes in the system attempt to include a certain transaction then, after time expires corresponding to U_{c} slots (called the transaction confirmation time), all nodes, if queried and responding honestly, will report the transaction as stable.

5 System Design

This section presents Spectrum protocol design starting from a naive approach based on Practical Byzantine Fault Tolerance (PBFT) [39] and gradually addressing the challenges. Our protocol is largely inspired by Ouroboros protocols family [40], [35], [36], therefore, we will use some of their core ideas and concepts.

5.1 Strawman Design: PBFTNetwork

For simplicity we begin with a notarization protocol based on PBFT and then iteratively refine it into the Spectrum protocol.

PBFTNetwork assumes that a fixed consensus group of $n = 3f + 1$ nodes has been pre-selected upfront and at most f of these nodes are Byzantine. The PBFT protocol is designed in such a way that there is no need to trust each individual notary, but only two-thirds of the set. This approach has proved its reliability in practice and has been widely used in various blockchain protocols for many years.

At any given moment of time, one of the nodes is the leader who observes the events on the connected blockchains, batch them and initiate a notarization round within the consensus group. All validators verify the proposed batch by checking for relevant updates on the connected chains. Upon successful verification each node signs the batch with a secret key and sends the signature to the leader.

Liveness and safety of the PBFTNetwork is guaranteed under the simplifying assumptions already mentioned above that at most f nodes are Byzantine. However, the assumption of a fixed trusted committee is unrealistic for open decentralized systems. Moreover, as PBFT consensus members authenticate each other via non-transferable symmetric-key MACs, each consensus member has to communicate with others directly, what results in the $O(n^2)$ communication complexity. Quadratic communication complexity imposes a hard limit on the scalability of the system. Such a design is not suitable for building a multichain system, since the workload of each validator grows linearly with each added chain.

In the subsequent sections, we address these limitations in four steps:

1. **Opening the Consensus Group.** We introduce a lottery-based mechanism to *select the consensus group dynamically.*

2. **Replacing MACs by Digital Signatures.** We replace MACs by digital signatures to make authentication transferable and thus opening the door for *sparser communication patterns* that can help reduce communication complexity.

3. **Scalable Collective Signature Aggregation.** We utilize Byzantine-tolerant aggregation protocol that allows for *quick aggregation of cryptographic signatures* and reduces communication complexity to $O(\log n)$.

4. **Eliminating Validator Bottleneck.** We assign each consensus participant to one or more distinct committees depending on the set of chains he is willing to support to *improve system scalability.*

5.2 Opening the Consensus Group

Spectrum is an open-membership protocol, so PBFTNetwork's assumption on a closed consensus group is not valid. Sybil attacks can break any protocol with security thresholds and an appropriate dynamic selection of the consensus group becomes crucial for preserving network's liveness and safety. Election of consensus group members should be performed in a random and trustless way to ensure that a sufficient fraction (at most f out of $3f+1$) of members are honest.

Similar selection mechanics is required in most blockchain protocols. Bitcoin [41] and many its successors are using Proof-of-Work (PoW) consensus, which, in essence, is a robust mechanism that facilitates randomized selection of a leader who is eligible to produce a new block. Later, PoW approach was adapted into a Proof-of-Membership mechanism [42]. This mechanism allows once in a while to select a new consensus group which then executes the PBFT consensus protocol.

A primary consideration regarding PoW-based consensus mechanisms is the amount of energy required to operate such systems. A natural alternative to PoW is a mechanism based on the concept of Proof-of-Stake (PoS) [43]. Rather than investing computational resources in order to participate in the leader selection process, participants of a PoS system instead run a process that randomly selects one of them proportionally to the stake. Pure PoS mechanism to solve the PBFT problem was firstly used in [44] to select both consensus group members and PBFT rounds leaders and to introduce randomness into this process, a verifiable Random Function (VRF) has been applied.

5.2.1 Verifiable Random Function

A Verifiable Random Function (VRF) [45] is a reliable way to introduce randomness into a protocol. By definition, a function $\mathcal{F}$ can be attributed to the VRF family if the following methods are defined for the $\mathcal{F}$:

– Gen: $Gen(1^l) \to (PK, SK)$, where PK is the public key and SK is the secret key.

– Prove: $Eval(x, SK) \to \pi$, where x is an input and $\pi := \Pi(x, SK)$ is the proof, associated with x and mixed with a random value, sampled from $\{0,1\}^{l_{\text{VRF}}}$.

– Verify: $Verify(x, \pi, PK) \to 0|1$, where the output is 1 if and only if $\pi \equiv \Pi(x, SK)$.

The most secure implementations of VRF nowadays are Elliptic Curve Verifiable Random Functions (ECVRFs). Basically, ECVRF is a cryptographic-based VRF that satisfies the uniqueness, collision resistance, and full pseudorandomness properties [46]. The security of ECVRF follows from the decisional Diffie-Hellman assumption in the random oracle model, thus ECVRF is a good source of randomness for a blockchain protocol. Using ECVRF is also cheap and fast, since single ECVRF evaluation is approximately 100 microseconds on x86-64 for a specific curves used in hash functions. Moreover, there is a great UC-extension for batch verification proposed by [47] which make it even faster by reducing the number of evaluations.

5.2.2 Lottery

Our lottery mechanism is based on ECVRF as a source of randomness and is generally inspired by Ouroboros Praos [40] and Algorand [44]. The lottery mechanism in general allows the protocol assign a specific *role* to a participant, while the validity of the participant's role can be verified using only publicly available data.

The main assigning logic is as follows:

1. Participant calculates a certain threshold value T according to predefined rules and using only publicly available data for the calculation.

2. Participant evaluates VRF function and calculates a random number y using the VRF's proof π.

3. If $y < T$ then the participant is considered valid for the respective role.

To be more precise, let's clarify that in our setting a threshold value T is calculated according to the formula $T = 2^{l_{\mathrm{VRF}}} \cdot \phi_f(\alpha, f)$ where $\alpha = s / \sum_{i=1}^{M} s_i$ is a relative stake. Consequently, the probability of winning is calculated as $p(\alpha, f) = 1 - (1 - f)^{\alpha}$. Thus, the winning probability depends on the participant's relative stake and is adjusted by the free parameter f. This is where the PoS concept comes into play: the bigger the stake, the higher the chance of winning the lottery.

The lottery mechanism is fast, secure, and adaptive, since the involved pre-defined parameters can be changed via the voting process. Moreover, the same primitives can be used to achieve different goals and we will utilize the lottery mechanism in several aspects of our protocol.

Consensus Group Lottery. In the current section, we are considering a lottery mechanism application for *dynamic consensus group selection*. The Spectrum protocol initially is running by the manually selected opening consensus group $\{PK_i\}_{i=1}^{M}$ of the predefined size M. Stakeholders interact with each other and with locally installed ideal functionalities $\mathcal{F}_{\mathrm{VRF}}$ and $\mathcal{F}_{\mathrm{LB}}$ over a sequence of $L = E \cdot R$ slots $S = \{sl_1, \ldots, sl_L\}$ consisting of E epochs with R slots each.

Let's clarify what the mentioned above pre-defined primitives are needed for. The ideal Verifiable Random Function functionality $\mathcal{F}_{\mathrm{VRF}}$ we use here is similar to the extended VRF functionality introduced by Christian Badertscher et al. [47]:

Ideal Leaky Beacon functionality $\mathcal{F}_{\mathrm{LB}}$ is used to sample an epoch random seed from the blockchain and is defined as follows:

Functionality $\mathcal{F}_{\mathrm{LB}}(e_n, C_{\mathrm{loc}})$

```
1: // New epoch random seed is sampled once per epoch.
2: // C_loc is the local chain of the validator.
3: if $e_n < 2$ then
4:     return false
```

```
end if
for each $B_k \in C_{\text{loc}} \mid (B_k.\mathsf{get}(e) \leq e_{n-1}) \ \wedge (\forall B_k.\mathsf{get}(sl) \in R \cdot (n-1) \cdot 2/3)$ do
    // Every block B_k in the C_loc was produced by i'-th leader
    // during j'-th slot, i.e. k = (i', j').
    $\pi^{\text{sl}} \leftarrow B.\mathsf{get}(\pi^{\text{sl}})$.
    Extract the random value $r^{\text{sl}} \leftarrow \pi^{\text{sl}}$.
    $y^{\text{rand}} \leftarrow \mathcal{H}(r^{\text{sl}}||\mathsf{RAND})$.
    $\eta_n \leftarrow \mathcal{H}(\eta_{n-1}||e_n||y^{\text{rand}})$.
end for
return $\eta_n$
```

An extended formal analysis of the security guaranties of the $\mathcal{F}_{\text{LB}}$ can be found in the original Ouroboros Praos paper [40].

Consensus group is constantly rotated each epoch $e_n > 2$. Any verified protocol participant PK_i can try to become a temporal member of the consensus group. Participant is verified if his verification key tuple v_i^{ver} is published in the blockchain during the epoch e_{j-2} in the special $\mathsf{VerificationRegTx}(v_i^{\text{ver}})$. The consensus group lottery flow is as follows:

1. At the end of the epoch $e_n > 2$ every verified participant PK_i requests a new epoch seed η_n from the $\mathcal{F}_{\text{LB}}$.
2. New consensus lottery threshold $T^{\text{cons}} = \phi_{f^{\text{cons}}}(\alpha_i^{n-2})$ is calculated by every PK_i using stake distribution (to get the relative stake α_i^{n-2}) from the blockchain state at the last block of the epoch e_{n-2}. Free parameter f^{cons} of the associated function ϕ is $f^{\text{cons}} = M_n/N_n$, where M_n is a pre-defined number of new consensus group members to select at epoch e_n and N_n is the total number of verified stakeholders.
3. When every PK_i evaluates $\mathcal{F}_{\text{VRF}}$ with input $x^{\text{cons}} = \eta_n||e_n$ and calculates the associated random number $y_{i,n}^{\text{cons}}$ from the received proof $\pi_{i,n}^{\text{e}}$, i.e. $y_{i,n}^{\text{cons}} = \mathcal{H}(r_{i,n}^{\text{e}}||\mathsf{CONS})$, where $r_{i,n}^{\text{e}}$ is a random number extracted from the proof and CONS is an arbitrary pre-defined constant.
4. To reveal the result of the consensus group lottery PK_i compares value $y_{i,n}^{\text{cons}}$ with the threshold $T_{i,n}^{\text{cons}}$. If $y_{i,n}^{\text{cons}} < T_{i,n}^{\text{cons}}$ then the participant is a legal member of new consensus group which will be active in the epoch e_{n+2}.
5. Finally, to declare his right to participate in the new consensus group, participant PK_i includes an associated proof $\pi_{i,n}^{\text{e}}$ into the $\mathsf{ConsLotteryResTx}(e_n, v_i^{\text{vrf}}, \pi_{i,n}^{\text{e}})$ and adds it into the main chain.

Note, that the members of the consensus group should be known ahead of time for the synchronization. Therefore, in order to participate in the e_n consensus lottery already verified participant must publish $\mathsf{VerificationUpdTx}$ message with his verification tuple at the epoch e_{n-2}. Public disclosure of the future consensus group doesn't give much advantage to an adversary since there are hundreds of consensus members in every epoch and denial of service attacks are difficult to succeed. At the same time any grinding attacks are limited because an adversary can't arbitrarily control η_n values.

The main task of the validators set elected via the consensus group lottery is to observe and notarize events using a digital signature aggregation mechanism which we will introduce in the next sections.

5.3 Replacing MACs by Digital Signatures

The main issue with MACs is that any node capable of validating MAC is also capable of generating new messages with valid MACs as the secret key used for MAC generation is also necessary for validation. Digital signatures, on the other hand, use asymmetric protocols for signature generation and signature verification powered by public-key cryptography. A valid secure digital signature for the message can only be generated with the knowledge of the secret key (non-forgery requirement), and verified with the corresponding public key (correctness requirement), and the secret key never leaves the signer's node. The authenticity of the message from the network node can be verified by any party knowing the node public key. Moreover, given the full history of communication, the malicious actor is still not able to forge the new message with valid signature of the node. This gives a way finer control over the set of permissions and provides a strong authentication method.

Spectrum utilizes the specific subset of signatures based on so-called sigma-protocols. The benefits of these protocols are numerous, including the possibility of proving complex logical statements inside the scheme, provable zero-knowledge, and use of standardized and well-established crypto-primitives, namely conventional cryptographic hash functions and standard elliptic curves with hard discrete logarithm problem. This means the high level of support in the existing chains without modification of the core opcodes or writing supplementary on-chain routines.

5.4 Scalable Collective Signature Aggregation

In this section we describe our approach to the following problem. The naive approach to writing the consensus values on the blockchain in a verifiable way would be simply write the resulting values together with the signatures from every node which successfully participated in the consensus protocol. Spectrum consensus groups can contain thousands of nodes. If one takes Schnorr signature scheme [48] with 256-bit keys, every signature is 64 bytes long. That means thousands of kilobytes of data needed to be written on the blockchain and consuming valuable storage space, not speaking on the computational efforts from the blockchain validating node to actually verify all these signatures. Therefore, in these circumstances, the signature aggregation method is mandatory.

The aggregation allows one to write a single shorter signature instead of the list of signatures while preserving similar security level. There are few signature aggregation schemes for the sigma-protocol based signatures, such as CoSi [49] and MuSig [50]. These protocols perform extremely well if all the keys of the predefined set of co-signers are included in the resulting signature generation. In this case instead of having thousands of separate signatures one has only one of the size of single Schnorr signature. But this is not the case with many realistic situations with large consensus groups (such as Spectrum). It would be too optimistic to assume that all the nodes are always online, and every single node is following the protocol honestly to every letter. One needs the mechanism to process these failures. Whereas CoSi proposes the method to process such failures, it comes at cost of significant increase in the size of the resulting signature. Our scheme relies on the similar ideas, however we tend to provide better scaling with faulty nodes and more compact constructions than the original CoSi.

In short, we construct a compact aggregated signature scheme with potential node failures based on standard cryptographic primitives. It must have constant small size in the absence of failures and provide reasonably small space and computational overheads in the presence of failures. The signing protocol must be performed in a distributed fashion providing defence from the malicious co-signers.

5.4.1 General Overview

We start with the MuSig scheme and modify it to the meet the criteria listed above. We assume the Discrete Logarithm group to be the subgroup of the elliptic curve as usual. That is, elliptic curve is defined over finite field, we consider subgroup of its points with coordinates in this field of prime order with fixed generator g and identity element being the point at infinity if the curve is written in the form $y^2 = f(x)$, f is the third degree polynomial. Nothing prevents one from using another group with hard discrete logarithm problem. We use multiplicative notation for the group operation, and the group elements except for generator are written in capital letters. The secret keys are the integers modulo group order, we will denote them by lowercase letters. H is the cryptographic hash function. When we write $H(A, B)$, we assume that there is a deterministic way of serializing the tuple (A, B), and this serialization is used as an argument for H. The public key corresponding to the private key x is the group element $X = g^x$.

Any interactive sigma-protocol consists of three stages in strict order: commitment (when one or more group elements are sent from prover to verifier), challenge (when the random number is sent from verifier to prover), response (when one or more numbers calculated from the previous stages and the secret key are sent from prover to verifier). This triple constitutes the Proof-of-Knowledge of the secret key. To turn the interactive protocol into a non-interactive one, Fiat-Shamir heuristic is used, where the challenge is replaced by the hash value of all the preceding public data.

The takeaways from this setting, which are important for the understanding of our construction are the following:

- In case of n nodes one must have n commitments to aggregate and the list should not be changed till the end of the protocol.
- As the commitments from different nodes come at potentially different time, there can be an attack on this stage. Say, one node does not pick the commitment based on random, but rather calculates it based on the commitments received from the other nodes. This kind of attack is known as k-list attack, as to forge the upcoming signatures the malicious node solves the k-list problem, which is quite possible with a sufficient amount of data. To exclude this possibility one needs all the nodes to "commit to Schnorr commitment" beforehand. One can use hash function with no homomorphic properties for that purpose.
- All the steps are strictly sequential. Hence, every stage must complete with the full aggregation of individual contributions. There does not seem to be a simple way to perform it fully asynchronously.
- Instead of the last step (response) it is sufficient to provide the proof of knowledge for the response. This brings no additional value to the conventional signatures, but it helps with the processing of the node failures during the execution. Namely, the consensus group may demonstrate that somebody in the group knew the discrete logarithms of the commitments not accounted for in the response stage. Therefore, the group as a whole could compute the full response if the failure had not occurred.
- There must be a way to count the failures above, such that the signature verifier could decide whether it tolerates this number or not.

5.4.2 Aggregation Rounds and Structures

Here we list the overall structure of aggregation to give a grasp on the overall process. The detailed explanation is presented below:

> **Round 1: Pre-Commitment.** Collect Commitments for Schnorr commitments. Structure: list of hashes of elliptic curve points. Distribute all the hashes after the aggregation.

Round 2: Commitment. Collect and aggregate Schnorr commitments. Structures: list of signatures (proofs of discrete logarithms for the commitments) together with Schnorr commitments. Distribute among all the nodes. Upon receiving every node verifies that the hashes of the points are those provided on round 1, and verifies the proofs of discrete logarithms. The commitments with the checks passed are aggregated to get the overall commitment. It is used to compute the challenge and the individual responses in the sigma–protocol.

Round 3: Response. Collect and aggregate the responses. Structure: list of individual responses. Upon receiving every individual response is verified. The responses which passed the verification are added together. If the response is invalid or missing, the corresponding discrete logarithm proof from round 2 is appended to the output.

Output. Aggregate signature (Y, z) together with the set

$$\{(Y_i, DlogProof(Y_i)\},$$

where i runs over the set of nodes which have not provided valid responses.

5.4.3 Signature Generation

The signature generation algorithm is as follows:

1. Each signer computes $a_i \leftarrow H(H(X_1, X_2, \ldots, X_n); X_i)$ and the aggregate public key $\tilde{X} \leftarrow \prod_i X_i^{a_i}$.
2. Each signer generates a pair $Y_i = g^{y_i}$ to commit to, commitment $t_i \leftarrow H(Y_i)$ and the signature σ_i of some predefined message with secret key y_i.
3. The commitments t_i are aggregated in the list L_1.
4. After every participating co–signer received L_1, the tuples (Y_i, σ_i) are aggregated in the list L_2.
5. Upon receiving the tuple (Y_i, σ_i), verify $t_i = H(Y_i)$, and verify that σ_i is a valid signature corresponding to Y_i. The failed records are excluded from L_2, the next steps and communication round.
6. Every node computes the aggregate commitment $Y = \prod_i Y_i$ using all the valid records in L_2.
7. Every node computes the challenge $c \leftarrow H(\tilde{X}, Y, m)$ and the responses $z_i \leftarrow y_i + c a_i x_i$.
8. The responses z_i are aggregated into list L_3.
9. Initialize $z \leftarrow 0$ and empty set $R \leftarrow \{\}$.
10. Upon receiving the response z_i, verify that $g^{z_i} = Y_i X_i^{a_i c}$.
11. If this is the case, set $z \leftarrow z + z_i$. Otherwise, insert corresponding entry from L_2 in R as (i, Y_i, σ_i).
12. Output the triple (Y, z, R).

5.4.4 Signature Verification

The signature verification is carried out as follows:

1. Compute $a_i \leftarrow H(X_1, X_2, \ldots, X_n; X_i)$.
2. Compute $\tilde{X} \leftarrow \prod_i X_i^{a_i}$.
3. Compute $X' = \prod_{i \notin R.0} X_i^{a_i}$.
4. Compute $Y' = \prod_{i \in R.0} Y_i$.
5. Compute $c \leftarrow H(\tilde{X}, Y, m)$.
6. Verify $g^z = X'^c Y Y'^{-1}$.
7. Verify all of $\sigma_i \in R.2$ with respect to $Y_i \in R.1$.
8. Compare $(n-k)$ (where k is the size of R) with the required threshold value.

5.4.5 Instantiation of Signature Aggregation

We instantiate our signature aggregation protocol on top of Handel [51], a Byzantine-tolerant aggregation protocol that allows for the quick aggregation of cryptographic signatures over a WAN. Handel has polylogarithmic time, communication and processing complexity.

Our signature aggregation protocol involves aggregation of three lists: $L1$, $L2$ and $L3$. As long as Handel requires that the partial aggregation function satisfies both commutativity and associativity conditions we have to replace lists with sets. We instantiate each of three aggregation rounds on top of Handel round. Because of parallel nature of Handel we have to run multicasting between chained rounds of aggregation in order to consistently aggregate. The resulted construction consists of three Handel rounds and two multicasting rounds in between.

5.5 Eliminating Validator Bottleneck

So far, each member of the consensus group had to track changes on all connected chains in order to participate in consensus properly. However, this approach reduces the number of possible consensus participants and limits the scalability of the system. Therefore, for the optimal design of our consensus protocol, we will use the following observations:

Observation 1: Events coming from independent systems S_k are not serialized.

Observation 2: Outbound transactions on independent systems S_k can be independently signed.

Utilizing those properties, we now introduce committee sharding. We modify the protocol in a way such that at each epoch e_n, K distinct committees consisting of nodes equipped with functionality unit $\mathcal{F}^k_{\text{ConnSys}}$ relevant to a specific connected system S_k are selected via the consensus group lottery. All primitives used in the lottery are equal for different committees, however, lotteries are independent.

We denote one such committee shard as V_n^k, which uniquely maps to S_k. Then, complete mapping of committees to chains at epoch e_n can be represented as a set of tuples committee-chain $\{(V_n^k, S_k)\}_{k=1}^K$. Throughout epoch e_n all events and on-chain transactions in S_k are handled exclusively by V_n^k. Nodes in V_n^k maintain a robust local ledger $L^{\text{loc},k}$ with notarized reports consisting of events observed in S_k.

5.5.1 Leader Lottery

Once all validators sets $\{V_n^k\}_{k=1}^K$ for epoch e_n are elected via the consensus group lottery, the lottery process does not stop, but this time, in order to initialize the notarization of the report, the leader of every committee should be determined.

The leader lottery flow for every separate V_n^k during epoch e_n is as follows:

1. At the end of the epoch e_{n-1} every consensus group member $PK_i \in V_n^k$ requests a new epoch seed η_n from the $\mathcal{F}_{\text{LB}}$.

2. Every k-th committee member calculates the leader lottery threshold value $T_{i,n}^{\text{lead},k} = \phi_{f_k^{\text{lead}}}(\alpha_i^{n-2})$. Stakeholders distribution is calculated according to the blockchain state at the last block of the epoch e_{n-2}. The parameter f_k^{lead} is the pre-defined value that determines how many slots will have at least one selected leader for the committee V_n^k.

3. When, for every slot $sl_j \in e_n$ every committee member PK_i evaluates $\mathcal{F}_{\text{VRF}}$ with input $x_j^{\text{lead}} = \eta_n || sl_j$ and calculates the associated random number $y_{i,j}^{\text{lead}}$ from the received proof $\pi_{i,j}^{\text{sl}}$, i.e. ${y_{i,j}^k}^{\text{lead}} = \mathcal{H}(r_j^{\text{sl}} || \mathsf{LEAD} || S_k^{\text{id}})$, where r_j^{sl} is a random number extracted from the proof and LEAD is an arbitrary pre-defined constant.

4. To reveal the result of the leader lottery PK_i compares value ${y_{i,j}^k}^{\text{lead}}$ with the threshold $T_{i,n}^{\text{lead},k}$ and if ${y_{i,j}^k}^{\text{lead}} < T_{i,n}^{\text{lead},k}$ then the participant is a j-th slot leader.

5. Finally, PK_i initiates a notarization round for slot sl_j with the associated proof $\pi_{i,j}^{\text{sl}}$ included in his initialization message.

Regarding the security it is important to note that slot leaders don't become publicly known in advance. An attacker can't see who is a slot leader until he initializes report notarization, thus an attacker can't know who specifically to attack in order to try to subvert a certain slot. All he can try to do is to make as many forks as possible to estimate the most advantageous, but according to the analysis [40] this advantage doesn't change the security properties of the entire protocol.

5.5.2 Syncing Shards

Each committee V_n^k forms the notarized reports of events and adds them into its local ledger $L^{\text{loc},k}$. All these reports should be periodically synced and added to a block of the main super ledger L^+ in order for the system to be able to compute a cross-chain state transition. To facilitate this process, reports should be broadcast to other committees. The main actors at this stage are:

1. *Local leader*: local committee leader.

2. *Relayer*: any protocol participant that broadcasts notarized reports to the local leader and to other committees' members. Every local leader can be a relayer at the same time.

3. *General leader*: one of the local leaders who added a block consisted of collected notarized reports and other internal transactions to the L^+.

There is no separate lottery for the general leadership and any local leader is able to publish his block to L^+, thus, he can choose from two main strategies:

1. *Wait*: malicious strategy where local leader waits for broadcasts from other committees members and doesn't broadcast his own report to eliminate competitors for adding a block.

2. *Broadcast and wait*: fair strategy where local leader immediately broadcasts his report, waits for broadcasts from other committees' and then competes honestly for adding a block.

There should be a motivation for an individual local leader to choose the fair strategy instead of keeping his report for too long and there also should be a motivation for every committee member to act as a relayer. This is achieved through the design of the incentive system.

There are three types of the incentive for the Spectrum protocol participants: $\{R_b, R_d, R_m\}$, where R_b is a guaranteed reward for adding a notarized report to the block, R_d is given for broadcasting a report to the general leader and R_m is given personally to the general leader who will finally add the block. Delivery reward R_d is given if and only if a delivery was made within a predetermined period of time Δt_d.

Reward amounts are initially configured in such a ratio that if $R_d = 0$ there is no prior strategy for local leaders, they will either wait for other reports or broadcast their reports with equal probability. At the same time, all other committee members are motivated to act as a relayers to receive an extra reward, since the notarized report can be firstly generated by any member of the committee. All the rewards except R_m are shared equally between all committees members whose signatures are included in the finally added block.

As a result, the syncing shards flow looks as follows:

1. After notarization, a committee member holding the notarized report which contains the local notarization time, sends it to his local leader and to other known committees members.

2. All committees members who receive notarized reports from other committees also send them to the local leader.

3. The local leader collects the received notarized reports.

4. When waiting time approaches Δt_d, the local leader forms and broadcasts a block consisting of all external collected reports and reports from the local $L^{\text{loc},k}$ that have not yet been added to L^+.

5. After block is reliably settled in the L^+, all associated participants can claim their rewards.

We also introduce another type of authority incentive that decreases chances of unfair and inactive participants in the consensus group lottery. When calculating the lottery threshold all stakes are weighed depending on the actions of their holders in the previous epoch, i.e. $s_i = A_\text{m} \cdot s_i^\text{real}$, where A_m is the authority multiplier. If some authority was a member of the previous committee and participated in the adding of at least 2/3 of the blocks produced in the considered period of time (same which is used to sample new epoch seed), then his actual stake s_i^real is multiplied by $A_\text{m} = 1$. Multiplier A_m decreases linearly to 0, which is the case where member was passive during the entire epoch.

With this mechanism, we solve the following problems:

- Members are motivated to be focused on cooperation with other committees so that their participation is reflected in each block added in the L^+.

- Inactive and dishonest members are automatically excluded from the next epoch committee.

- Participants are motivated to stay active throughout the entire epoch so that their chances of being selected in the committee don't decrease due to an authority multiplier $A_\text{m} < 1$, otherwise, in order to even the odds with new lottery participants, they will either have to increase their real stake or skip the lottery until the next one.

5.5.3 Key Evolving Signature Scheme

All blocks added into L^+ must be signed with a committee leader's signature. In regular digital signature schemes, an adversary who compromises the signing key of a user can generate signatures for any messages, including messages that were generated in the past. Usage of the Key Evolving Signature (KES) scheme provide the forward security [52] that is necessary for handling the adaptive corruption setting.

A function $\mathcal{F}$ can be attributed to the KES family if the following methods are defined:

– Gen: $Gen(1^l) \rightarrow (PK, SK)$, where PK is the public key and SK is the initial the secret key.

– Update: $Update(SK) \rightarrow SK'$, where SK' is associated with new time period.

– Sign: $Sign(SK, m) \rightarrow \sigma$, where σ contains the actual time period.

– Verify: $Verify(\sigma, PK, m) \rightarrow 0|1$.

Accordingly, KES allows any protocol participant to verify that a given signature was generated with the legal signing key for a particular slot. The security guarantees are achieved by evolving the secret key after each signature in a way that the actual secret key was used to sign the previous message cannot be recovered.

One of the most efficient realizations is the MMM scheme [53]. This scheme uses Merkle trees in the KES methods, resulting in good performance in terms of updating time and signature size. Using this scheme, 2^l secret keys can be securely restored, while size of the signature is kept constant and depends on only pre-defined security parameter l.

5.5.4 Forks and Integrity

Protocol flow implies that there can be a several local leaders in every connected S_k committee, which leads to forks. This type of fork is a normal part of the protocol lifecycle, however, total possible number of the normal forks in our protocol is greater than in other blockchains, since any of the local leaders can append their blocks to L^+. The chance of occurring a malicious forks produced by an adversary is minimized due to the lottery and the incentive mechanism design. In addition, the task for an adversary becomes more difficult by virtue of the interaction between the protocol participants during the syncing shards process.

For the above reasons, the main rules for resolving forks are simple and are followed by members of all committees when validating a proposed blocks:

1. *Densest chain*: this rule mandates that if two chains C and C' start diverging at some time τ according to the reported beacon's slots then prefer the chain which is denser in a sufficiently long interval after that time. Full algorithm of this novel chain selection rule can be found in the original paper [35].

2. *Max stake*: if the densest chain rule doesn't resolve a slot battle, then the valid chain chooses according to the real stake size of the battled chains, the maximum stake is the winner.

We will note here, that the densest chain rule is crucial for a global clock synchronization. It offers a useful guarantee than the joining party will end up with some blockchain that, although arbitrarily long, is at worst forking from a chain held by an honest and already synchronized party by a bounded number of blocks (equal to the security parameter K_{f}) with overwhelming probability [36].

However, a large number of forks still significantly affect properties that maintain the integrity of the L^+:

1. *Latency*: the number of elapsed slots required for a transaction to appear in a block on the L^+.
2. *Finality*: the number of elapsed slots required for a transaction to become settled and immutable.

The latency of the protocol is good enough due to the short duration of the slots, while the finality, as a result of the functional features of our protocol, depends on the connected S_k integrity properties.

Most ledgers do not guarantee instant finality of transaction, that means that any (or all) transactions may not be applied to the corresponding S_k ledgers in the end. Different blockchains has different finality parameters, and the Spectrum finality time corresponding to adding K_{f} blocks should be greater than all of them. Thus, a reliable confirmation time should be set with a margin and, therefore, using the number of slots Δsl that have passed in the Spectrum network, developers should be able to receive information about the number of blocks that have passed in any connected blockchain during this period of time. The duration of block creation in each S_k is different, but the average values are preserved for a certain period of time $\Delta T >> d_s$, where d_s is the duration of Spectrum's slot. Thus, after each ΔT time interval, Spectrum network will update the set of constants: $\{(d_k, K_{\mathrm{f}}^k)\}_{k=1}^K$, where d_k is a block duration in the S_k and K_{f}^k is the default reliable number of confirmations in the S_k.

Using the data above, each Spectrum's Δsl can be associated with the delta of blocks that have passed in any connected blockchain: $\{\lfloor \Delta sl \cdot d_s \,/\, d_k) \rfloor\}_{k=1}^K$. When forming transaction, developers can specify a custom reliability factor $\hat{K}^{\mathrm{f}}$. This factor will be compared with the ratio of the number of blocks passed on the associated S_k to the default reliable number of confirmations K_{f}^k of this system.

The ability to access this information is important for tracking the status of value carrying units in the Spectrum's global state. The aspects of the implementation of our ledger is described further in the text.

5.6 Clock Synchronization

As we pointed out in 4.1 the protocol requires a common notion of time among all participants. To avoid relying on centralized time oracles which would undermine network security we adopt decentralized logical time synchronization technique based on synchronization beacons [36].

All committees V_n^k members participate in the synchronization process, and it is based on the following logical blocks:

Synchronization slots. Once a consensus participant's local time reaches synchronization slot $sl_{n \cdot R}, n \in \mathbb{N}$, his clocks are adjusted before moving to the next slot (i.e. next epoch).

Synchronization beacons. In addition to other messages, all members of the consensus group generate so-called synchronization beacons. For every local slot $sl_{i,j}^{\mathrm{loc}} \in [n \cdot R + 1, \ldots, n \cdot R + R/6], n \in \mathbb{N}$ every PK_i evaluates $\mathcal{F}_{\mathrm{VRF}}$ functionality with input $x_j^{\mathsf{sync},n} = (\eta_n || sl_{i,j}^{\mathrm{loc}})$ to get a proof $\pi_{i,j}^{\mathrm{sl}}$ and checks if he has the right to release a beacon by comparing the pseudo-random value $y_{i,j}^{\mathsf{sync}} = \mathcal{H}(r_j^{\mathrm{sl}} || \mathsf{SYNC})$

with the corresponding threshold $T^{\text{sync}}_{i,n} = 2^{l_{\text{VRF}}} \cdot \phi(\alpha_i^{n-2})$. If $y^{\mathsf{sync}}_{i,j} < T^{\text{sync}}_{i,n}$ then the participant broadcasts a beacon $b^{\text{sync}}_{i,j} = (v_i^{\text{vrf}}, sl^{\text{loc}}_{i,j}, \pi^{\text{sl}}_{i,j})$.

Arrival times bookkeeping. Every consensus participant PK_i maintains an array $\mathbf{b}_i^{\text{set}}$ of received beacons with beacon's arrival local time $sl^{\text{rec}}_{i,j} : (sl^{\text{loc}}_{i',j}, \mathsf{flag}) \in \mathbb{N} \times (\mathsf{final}, \mathsf{temp})$. Assume a beacon $b^{\text{sync}}_{i',j'}$ emitted by $PK_{i'}$ is fetched by a party PK_i for the first time:

- If PK_i has not yet passed synchronization slot $sl^{\text{loc}}_{i,n\cdot R}$ and the received beacon belongs logically to this party's next epoch, then decision is marked as temporary and PK_i stores a record $sl^{\text{rec}}_{i,j} : (sl^{\text{loc}}_{i',j'}, \mathsf{temp})$. Value $sl^{\text{rec}}_{i,j}$ will be adjusted once this party adjusts its local time-stamp for the next epoch.
- If PK_i has already passed synchronization slot $sl^{\text{loc}}_{i,n\cdot R}$ but not yet passed slot $sl^{\text{loc}}_{i,(n+1)\cdot R}$, then the received time is defined as the current local slot number and is considered final, i.e. $sl^{\text{rec}}_{i,j} : (sl^{\text{loc}}_{i',j'}, \mathsf{final})$.

If a party has already received a beacon for the same slot j' and creator $PK_{i'}$, it will set the arrival time equal to the first one received among those.

The synchronization interval. For a local clock adjustment, which is triggered by a synchronization slot only beacons with recorded arrival time in the interval $[(n-1) \cdot R + 1, \ldots, (n-1) \cdot R + R/6]$ are used.

Computing the adjustment evidence. The adjustment is computed based on the received beacons set $\mathbf{b}_i^{\text{set}}$. Beacon $b^{\text{rec}}_{i,j}$ is only considered valid for adjusting procedure triggered by a synchronization slot if:

1. Recorded time $sl^{\text{loc}}_{i',j'} \in b^{\text{sync}}_{i',j'}$ is final and belongs to the synchronization interval $[(n-1) \cdot R + 1, \ldots, (n-1) \cdot R + R/6]$.
2. Beacon is included into the block whose creation slot belongs to the interval $[(n-1) \cdot R + 1, (n-1) \cdot R + 2 \cdot R/3]$
3. Beacon's proof $\pi^{\text{sl},n}_{i,j}$ is valid.

Adjusting the local clock. Every party PK_i computes $\mathsf{shift}_{i,n}$ to adjust its clock in the synchronization slot $sl_{n\cdot R}$. Value of the shift is calculated as $\hat{\mathsf{shift}}_{i,n} = \mathsf{median}\{sl^{\text{loc}}_{i',j'} - sl^{\text{rec}}_{i,j} \mid \forall b^{\text{sync},n}_{i',j'} \in \mathbf{b}_i^{\text{set}}\}$. Using the calculated $\mathsf{shift}_{i,n}$ value all $\mathbf{b}_i^{\text{set}}$ records with temp status is updated as $sl^{\text{rec},n}_{i,j} + \hat{\mathsf{shift}}_{i,n}$ and the adjustment procedure is performed again to receive a final $\mathsf{shift}_{i,n}$ value.

As a result of the above synchronization procedure, at the beginning of the next round the party PK_i will report a local time equal to $n \cdot R + \mathsf{shift}_{i,n} + 1$. If $\mathsf{shift}_{i,n} > 0$, the party proceeds by emulating its actions for shift rounds. If $\mathsf{shift}_{i,n} < 0$, the party remains a silent observer until its local time has advanced to slot $n \cdot R + 1$ and resumes normally at that round.

5.7 Decentralized On-Chain Asset Management

In order to lift on-chain assets to cross-chain level, Spectrum has to take control over them. Thus, all assets that Spectrum operates on are stored in on-chain *vaults* which are ruled by the consensus.

Each vault corresponding to the connected system S_k stores an epoch number n, an aggregated public key αPK_n^k of the current validator set V_n^k and is guarded with a smart-contract capable of performing an aggregated signature verification $verify : (\sigma_n^k, \alpha PK_n^k, m_n^k) \to 0|1$.

5.7.1 Rotating Authorized Committees in Vaults

As explained before, committees in Spectrum are constantly rotated. Thus, vaults have to be updated accordingly. The transition is performed with the help of "retiring" committee, that must call *changeEpoch* : $(\sigma_n^k, \alpha PK_{n+1}^k)$ on the vault contract, where αPK_{n+1}^k is the aggregated public key of the next commitee.

5.8 Ledger

Spectrum's global state includes a pool of value carrying units called *cells*. A *Cell* encodes monetary value (e.g., fungible or non-fungible tokens) travelling inside the system and across its boards.

TxId =	H(Tx)
CellId =	H(TxId × I)

Each cell has a unique identifier derived from ID of the transaction that produced the cell and its index in the transaction outputs. The identifier remains stable even when cell is modified as we explain below.

Value =	`u64`
ChainId =	`u64`
Version =	`u64`
ProgressPoint =	ChainId × `u64`
ActiveCell =	CellId × Address × Value × Version
BridgeInputs =	[`u64`]
Destination =	ChainId × BridgeInputs
TermCell =	CellId × Value × Destination
Cell =	ActiveCell ⊎ TermCell

We distinguish two essential types of cells depending on the state of the value they encode.

5.8.1 Active cells

Active Cell is a value travelling between owners inside the system. An Active Cell can be modified while preserving its original stable identifier. With each mutation version of the cell is incremented which is initialized with `0` when the cell is created. This opens the door for smooth management of shared cells (e.g., stablecoin bank or liquidity pool).

5.8.2 Authenticators, Addresses and Ownership

Authenticator =	ProveDlog ⊎ Script
Address =	H(Authenticator)

Each active cell has an exclusive owner identified by an address. Address is derived from an authenticator by applying collision resistant hash function to it. To prove ownership of a cell a party must supply an authenticator whose hash matches the owning address. An authenticator can either be a public key or a script. Once authenticated an owner can freely move value locked within the cell by either mutation or elimination of it.

5.8.3 Terminal cells

Terminal cells encode value to be exported into an external system. In contrast to active cells, terminal cells are immutable and value from them cannot be moved within the system anymore.

5.8.4 Transactions and Effects

$$\boxed{\begin{aligned}\text{Imported} &= \text{ActiveCell}\\ \text{Exported} &= \text{CellId}\\ \text{Revoked} &= \text{CellId}\\ \text{Progressed} &= \text{ProgressPoint}\\ \text{Eff} &= \text{Imported} \uplus \text{Exported} \uplus \text{Revoked} \uplus \text{Progressed}\end{aligned}}$$

Global pool of cells is modified by atomic state modifiers called *Effects* and *Transactions*.

Effects are state transitions imported from external systems exclusively by local committees. Below we list possible effects:

1. Import of value. A deposit into one of Spectrum's on-chain vaults which results in creation of a new cell.
2. Export of value. An outbound transaction that transfers value from Spectrum's on-chain vault to user address on particular blockchain.
3. Revocation of previously imported value due to roll-back on the source chain.
4. Signalisation that external system reached particular progress point.

$$\boxed{\begin{aligned}\text{CellRef} &= \text{CellId} \times \text{Version}\\ \text{Inputs} &= \text{CellRef} \times [\text{CellId} \uplus \text{CellRef}]\\ \text{RefInputs} &= [\text{Cell}]\\ \text{EvaluatedOutputs} &= [\text{Cell}]\\ \text{Tx} &= \text{Inputs} \times \text{RefInputs} \times \text{EvaluatedOutputs}\end{aligned}}$$

In contrast to effects, transactions are state transitions triggered by Spectrum users. A transaction accepts cells that it wants to mutate or eliminate as inputs and outputs new cells or upgraded versions of mutated cells. Therefore, scope of transaction is restricted to its inputs and outputs.

Transactions: Referencing inputs. Transaction can reference cells to use as inputs either by cell ref (fully qualified reference) or only by stable identifier. In the latter case, a concrete version of the cell with the given stable identifier will be resolved in the runtime of the transaction. Importantly, each transaction must have at least one fully qualified input, this guarantees that each transaction is unique.

Transactions: Programmability. Some outputs may be computed in the runtime of a transaction as a result of script(s) execution. It is also possible to include pre-evaluated outputs into transaction in order to save on on-chain computations. This design allows dApp developers to choose the amount of on-chain computations of their apps.

5.8.5 Dealing with finality of imported value

Because Spectrum is a cross-chain system, monetary value there is usually imported from an external system (e.g. Cardano or Ergo). Since most of the cryptocurrencies don't provide instant finality of transactions, on-chain transaction that once imported value into Spectrum's on-chain vault may be rolled-back. There are two ways of preventing "dangling" value inside Spectrum. On the one end of spectrum is a conservative approach: wait for settlement on the source chain (e.g. 120 blocks in Ergo) before import to be 100% sure the transaction will not be rolled back. On the other end is a reactive approach: import value immediately and revert locally transactions that depend on that piece of value in the case of rollback. Conservative approach offers simplicity and is cheaper to execute, while reactive one allows to work with imported value inside spectrum with minimal delays.

Observation: Probability of a rollback at a certain height decreases exponentially with square root scaling in the exponent as chain extends [40].

Based on this observation we choose a hybrid approach. Value is imported with a small delay D^c which is configured for each chain and is sufficient to keep probability of rollback low. If rollback happens after the import all transactions directly or transitively depending on the dangling value are reverted.

As long as outbound transactions can not be reverted it is of paramount importance to wait for complete settlement of the imported value before allowing to export it. Each cell is associated with a set of dependencies called *anchors* represented as unique identifier of a chain and a height which the chain is required to reach in order for the anchor to be deemed as *non-anchored.* Active anchors leak from cells in inputs into created cells in outputs. It is impossible for a terminal cell to be exported until all anchors it depends on are reached.

5.9 The Full Protocol

Let's summarize all of the above and describe the full flow of the Spectrum protocol. Protocol is running by a set of manually selected opening consensus groups $\{V_1^k\}_{k=1}^K$ for K connected distributed systems $\{S_k\}_{k=1}^K$. Each group consists of at least M_k stakeholders interacting with each other and with the ideal functionalities $\mathcal{F}_{\text{Init}}$, $\mathcal{F}_{\text{VRF}}$, $\mathcal{H}$, $\mathcal{F}_{\text{LB}}$, $\mathcal{F}_{\text{AggSig}}$, $\mathcal{F}_{\text{KES}}$, $\mathcal{G}_{\text{ImpLClock}}$ and $\mathcal{G}_{\text{Ledger}}$ over a sequence of $L = E \cdot R$ slots $S = \{sl_1, \dots, sl_L\}$ consisting of E epochs with R slots each.

Functionality $\mathcal{F}_{\text{Init}}$ [35] formalizes the procedure of genesis block creation and distribution. Functionality $\mathcal{F}_{\text{AggSig}}$ implements the presented in 5.4 aggregated signature scheme logic. Functionality $\mathcal{G}_{\text{ImpLClock}}$ [36] implements the local clock setting and adjusting logic. Functionality $\mathcal{G}_{\text{Ledger}}$ implements the logic of interaction with the ledger. Also, each protocol participant maintains at least one functionality unit $\mathcal{F}^k_{\text{ConnSys}}$ that allows him to interact with the connected S_k.

Protocol configuration is represented by publicly known set of constants: $R, l_{\text{VRF}}, K_{\text{f}}, K_{\text{g}}, \mathbf{S}_{\text{id}} = \{S_k^{\text{id}}\}_{k=1}^K, \mathbf{f}_{\text{lead}} = \{f_k^{\text{lead}}\}_{k=1}^K, \mathbf{f}_{\text{cons}} = \{f_k^{\text{cons}}\}_{k=1}^K$

5.9.1 Bootstrapping

The system is bootstrapped in a trusted way. All M_k members of $\{V_1^k\}_{k=1}^K$ committees perform the following procedure:

1. On-chain vaults are initialized with an aggregated public key aPK_1^k of the initial committee.

2. All committee V_1^k members i.e. $\forall PK_i \in V_1^k$ must generate the tuple of verification keys $v_i^{\text{ver}} = (v_i^{\text{vrf}}, v_i^{\text{kes}}, \mathbf{S}_{\text{id},i})$, using the ideal functionalities $\mathcal{F}_{\text{VRF}}$ and $\mathcal{F}_{\text{KES}}$. The tuple also includes a set of ids of the connected distributed systems $\mathbf{S}_{\text{id},i} \subset \mathbf{S}_{\text{id}}$, the functionalities $\{\mathcal{F}^k_{\text{ConnSys}}\}_{k=1}^{K'}, K' \leq K$ to interact with which the participant PK_i is equipped with. Verification tuple is committed on-chain in the $\mathsf{VerificationRegTx}(v_i^{\text{ver}})$.

3. Full set of the verification keys tuples $V_{\text{ver}} = \{(v_i^{\text{vrf}}, v_i^{\text{kes}}, \mathbf{S}_{\text{id},i})\}_{i=1}^M$ with the initial stakes $S = \{s_i\}_{i=1}^{M_k}$ must be stored in the genesis block B_0 and acknowledged by all members of the initial consensus group (meaning members of all $\{V_1^k\}_{k=1}^K$ committees).

4. Functionality $\mathcal{F}_{\text{LB}}$, parameterized with the confirmed V_{ver} is evaluated independently by every participant to sample an initial random seed value $\eta \leftarrow \{0,1\}^l_{\text{VRF}}$.

5. Finally, all approved stakeholders should agree on the genesis block $B_0 = (V_{\text{ver}}, S, \eta)$.

5.9.2 Chain Extension

Once the system is bootstrapped, the Spectrum protocol operates in a normal flow. Committee $\{V_1^k\}_{k=1}^K$ members adds notarized reports of events observed on external connected systems $\{S_k\}_{k=1}^K$ into the local ledgers $\{L^{\text{loc},k}\}_{k=1}^K$. Blocks with all protocol updates are stored in the common for all participants super ledger L^+.

1. Before the epoch $e_n > 2$ begins each protocol participant PK_i must update his state variables:
   - Receive new epoch seed η_n from the $\mathcal{F}_{\text{LB}}$.
   - Set the leader lottery thresholds for each k-th committee he is involved in $\mathbf{T}^{\text{lead}} = \{T_{i,n}^{\text{lead},k} = \phi_{f_k^{\text{lead}}}(\alpha_{i,\text{k}}^{n-2}\}_{k=1}^{K'}, K' \leq K$, where $\alpha_{i,\text{k}}^{n-2}$ is a participant's relative stake relative to V_n^k members according to the state of the blockchain at the end of the epoch e_{n-2}.
   - Set the synchronization lottery threshold $T_{i,n}^{\text{sync}} = 2^{l_{\text{VRF}}} \cdot \phi(\alpha_i^{n-2})$, where α_i^{n-2} is a participant's relative stake relative to all K committees members according to the state of the blockchain at the end of the epoch e_{n-2}.
2. In the epochs first (synchronization) slot each PK_i adjusts his local clocks by $\mathsf{shift}_{i,n}$ value calculated according to previously collected synchronization beacons set $\mathbf{b}^{\text{set}}$.
3. During the epoch all online V_n^k member collects existing chains from L^+ and verifying that for every chain, every block, produced up to K_{f} blocks before contains correct data about the corresponding slot sl' leader PK'. Each validator must verify that PK' is indeed the winner of the leader lottery for slot sl' as well a valid member of the legitimate committee $V_{n'}^k$. All forks must be resolved by the densest chain and largest stake rules in the corresponding priority.
4. During the epoch, for every slot $sl_j \in [R \cdot n, R \cdot (n+1)]$ every committee V_n^k member PK_i separately evaluates $\mathcal{F}_{VRF}$ with an input $x_{i,j}^{\text{lead}} = \eta_n || sl_j$ to receive a slot proof $\pi_{i,j}^{\text{sl}}$ and an associated random value r_j^{sl}.

   Then PK_i calculates $y_{i,j}^{\text{lead}} = \mathcal{H}(r_j^{\text{sl}} || \mathsf{LEAD} || S_k^{\text{id}})$ and compares it with the associated threshold $T_{i,n}^{\text{lead},k}$. If $y_{i,j}^{\text{lead}} < T_{i,n}^{\text{lead},k}$ then the participant is the slot sl_j leader.

   Leader is allowed to:

   - Initiate the notarization round in his local committee V_n^k to add new notarized report into $L^{\text{loc},k}$.
   - Propose a new block to be added to the L^+.

   In addition, during the first $R/6$ slots of the epoch all PK_i checks his right to release a synchronization beacon comparing the pseudo-random value $y_{i,j}^{\mathsf{sync},n} = \mathcal{H}(r_j^{\text{sl}} || \mathsf{SYNC})$ with a corresponding threshold $T_{i,n}^{\text{sync}}$. If successful then the participant broadcasts a beacon message $b_{i,j}^{\text{sync}} = (v_i^{\text{vrf}}, sl_{i,j}^{\text{loc}}, \pi_{i,j}^{\text{sl}})$.

5. All committee V_n^k members observe events in their systems S_k and in the $L+$ mempool. If PK_i is a slot sl_j leader, then he is able to propose a report b_j of events observed in S_k, which should be notarized by other members of the V_n^k using the aggregated signature functionality $\mathcal{F}_{\mathrm{AggSig}}$ and then added to the local ledger $L^{\mathrm{loc},k}$.

6. Notarized report b_j^* can first be formed by any member of the V_n^k. The report must be immediately sent to the leader who initiated its notarization and to the members of other committees. After the leader receives enough reports he forms a block B' consisting of all external collected reports and reports from the local $L^{\mathrm{loc},k}$ that have not yet been added to L^+. He must include into the block the proof of his leadership $\pi_{i,j}^{\mathrm{sl}}$, sign the block with $\mathcal{F}_{\mathrm{KES}}$ and broadcast it to his peers from all committees with the correct signature σ_{KES} included.

7. After the finality K_{f} blocks are passed since B' settlement in the L^+, all members of all committees that participated in the formation of the block B' can claim their rewards.

5.9.3 Epoch Transition

1. **Consensus Group Lottery**.

   – At the beginning of the epoch $e_{n-1} > 2$ each verified PK_i willing to participate in the consensus group lottery for the V_n^k commit his willing in the message $\mathsf{VerificationUpdTx}(v_i^{\mathrm{vrf}}, \mathbf{S}_i^{\mathrm{set}})$ if he is already verified, or generate verification keys tuple and broadcasts $\mathsf{VerificationRegTx}(v_i^{\mathrm{ver}})$.

   – At end of the epoch e_{n-1} each verified and willing to participate in the consensus lottery PK_i calculates new consensus lottery thresholds for all committees he wants to be selected in $\{T_{i,n}^{\mathrm{cons},k} = \phi_{f_k^{\mathrm{cons}}}(\alpha_{i,\mathrm{ver}}^{n-2} \cdot A_{\mathrm{m}}^{i,n}\}_{k=1}^{K'}, K' \leq K$, where $\alpha_{i,\mathrm{ver}}^{n-2}$ is a participant's relative stake relative to all verified participants equipped with $\mathcal{F}_{\mathrm{ConnSys}}^k$ according to the state of the blockchain at the end of the epoch e_{n-2} and $A_{\mathrm{m}}^{i,n}$ is an activity multiplier.

   – When every PK_i evaluates $\mathcal{F}_{\mathrm{VRF}}$ with input $x_{i,n}^{\mathsf{cons}} = \eta_n || e_n$ to receive an epoch proof $\pi_{i,n}^{\mathrm{e}}$.

   Then for each $S_k^{\mathrm{id}} \in \mathbf{S}_i^{\mathrm{set}}$ calculates the associated random number $y_{i,n}^{\mathsf{cons},k}$ from the proof $\pi_{i,n}^{\mathrm{e}}$, i.e. $y_{i,j}^{\mathsf{cons},k} = \mathcal{H}(r_n^{\mathrm{e}} || \mathsf{CONS} || S_k^{\mathrm{id}})$. If $y_{i,j}^{\mathsf{cons},k} < T_{i,n}^{\mathrm{cons},k}$ then PK_i is a member of V_n^k committee.

   In order to approve the results of the lottery, the participant broadcasts a message with evidence $\mathsf{ConsLotteryResTx}(e_n, v_i^{\mathrm{vrf}}, S_k^{\mathrm{id}}, \pi_{i,n}^{\mathrm{e}})$.

2. **Committee key aggregation**. Once the new committee is selected, nodes in the V_n^k aggregate their individual public keys PK_i into a joint one aPK_n^k, which is needed to sign the batch applying transactions with the external events: inbound value transfers, outbound value transfers, boxes eliminations.

3. **Committee transition**. Nodes in the V_{n-1}^k publish cross-chain message $m_n^k : (aPK_n^k, \sigma_{n-1}^k)$, where σ_{n-1}^k is an aggregated signature such that $verify : (\sigma_{n-1}^k, aPK_{n-1}^k, m_n^k) = 1$. Finally, vaults are updated such that $vault^k\{(e_{n-1}, aPK_{n-1}^k)\} := (e_n, aPK_n^k)$.

5.9.4 Registration

Any Spectrum stakeholder can register to become a committee member of his local system S_k. To get a chance to be included in the set of validators V_n^k of the epoch e_n participant PK_i should register in the lottery during the epoch e_{n-3} by publishing his verification tuple $(v_i^{\text{vrf}}, v_i^{\text{kes}}, \mathbf{S}_i^{\text{set}})$ into the L^+. Once K_{f} blocks are added on top of this publication the participant is considered as verified. Before verification, PK_i must synchronizes with the network by restoring the current chain C from the genesis block B_0 received from the functionality $\mathcal{F}_{\text{Init}}$. He also must adjust his local clock based on the synchronization beacons of the current global epoch using the functionality $\mathcal{G}_{\text{ImpLClock}}$. When all synchronization processes are completed, PK_i is considered a valid participant of the Spectrum protocol.

In the manner described, the Spectrum protocol reaches consensus and implements the cross-chain interoperability. Our solution is fairly decentralized, fast and scalable, and thus can be used in a large number of applications and scenarios.

6 Applications

6.1 Decentralized Cross-Chain Oracle

The system is capable of providing a notarized set of events observed in supported external system be it a blockchain or general data source(s). Cross-Chain Oracle is simple yet opens the door for interoperability for all dApps on Layer1.

6.2 Custodial Asset Management

In custodial mode of operation the system is capable of managing user assets which are stored on corresponding blockchains in vaults which were defined previously.

Natively Cross-Chain Applications Decentralized custodial management in conjunction with a computational layer can be highly beneficial for expanding the capabilities of the system. This moves us beyond simple bridges to what we call Natively Cross-Chain Applications (NCCAs).

NCCAs are applications that are deployed in cross-chain network and are capable of interacting with other blockchains without the need of external oracles or bridges. Compared to single-chain dApps, NCCAs unlocks an additional functionality by taking advantage of multiple chains simultaneously. They make it possible to aggregate fragmented liquidity on different chains into one chain or a coordinated pool of assets and improve the user experience by enabling the localization and customization of parameters and feature sets of the same application on different chains. These unique advantages make them the future of web3 dApps.

References

[1] Dr Miraz and David Donald. *Atomic Cross-chain Swaps: Development, Trajectory and Potential of Non-monetary Digital Token Swap Facilities.* Jan. 2019. DOI: 10.33166/AETiC.2019.01.005. URL: https://arxiv.org/pdf/1902.04471.pdf.

[2] Stefan Schulte et al. *Towards Blockchain Interoperability.* 2019. URL: https://www.semanticscholar.org/paper/Towards-Blockchain-Interoperability-Schulte-Sigwart/6d777ba49d5e8f4d1a22b8ee287282fdceefeb8d.

[3] Soohyeong Kim, Yongseok Kwon, and Sunghyun Cho. *A Survey of Scalability Solutions on Blockchain.* Oct. 2018. DOI: 10.1109/ICTC.2018.8539529. URL: https://ieeexplore.ieee.org/document/8539529.

[4] Vitalik Buterin. *Chain Interoperability.* 2016. URL: https://allquantor.at/blockchainbib/pdf/vitalik2016chain.pdf.

[5] Rafael Belchior et al. *A Survey on Blockchain Interoperability: Past, Present, and Future Trends.* 2021. arXiv: 2005.14282 [cs.DC]. URL: https://arxiv.org/abs/2005.14282.

[6] Gang Wang. *SoK: Exploring Blockchains Interoperability.* Cryptology ePrint Archive, Paper 2021/537. https://eprint.iacr.org/2021/537. 2021. URL: https://eprint.iacr.org/2021/537.

[7] Randhir Kumar and Rakesh Tripathi. *Content-Based Transaction Access From Distributed Ledger of Blockchain Using Average Hash Technique.* Jan. 2021. DOI: 10.4018/978-1-7998-3295-9.ch003. URL: https://www.researchgate.net/publication/348120393_Content-Based_Transaction_Access_From_Distributed_Ledger_of_Blockchain_Using_Average_Hash_Technique.

[8] Babu Pillai, Kamanashis Biswas, and Vallipuram Muthukkumarasamy. *Blockchain Interoperable Digital Objects.* June 2019. DOI: 10.1007/978-3-030-23404-1_6. URL: https://www.researchgate.net/publication/333860173_Blockchain_Interoperable_Digital_Objects.

[9] Damiano Di Francesco Maesa and Paolo Mori. *Blockchain 3.0 applications survey.* 2020. DOI: https://doi.org/10.1016/j.jpdc.2019.12.019. URL: https://www.sciencedirect.com/science/article/pii/S0743731519308664.

[10] D. Balazs. *Herdius whitepaper.* 2017. URL: https://herdius.com/whitepaper/HerdiusTechnicalPaper.pdf.

[11] Eder John Scheid et al. *Bifröst: a Modular Blockchain Interoperability API.* Oct. 2019. DOI: 10.1109/LCN44214.2019.8990860. URL: https://www.researchgate.net/publication/339269755_Bifrost_a_Modular_Blockchain_Interoperability_API.

[12] S. Thomas and E. Schwartz. *A protocol for interledger payments.* 2015. URL: https://interledger.org/interledger.pdf.

[13] Reza M. Parizi et al. *Integrating Privacy Enhancing Techniques into Blockchains Using Sidechains.* 2019. DOI: 10.1109/CCECE.2019.8861821. URL: https://arxiv.org/pdf/1906.04953.pdf.

[14] Amritraj Singh et al. *Sidechain technologies in blockchain networks: An examination and state-of-the-art review.* 2020. DOI: https://doi.org/10.1016/j.jnca.2019.102471. URL: https://www.sciencedirect.com/science/article/pii/S1084804519303315.

[15] *Intro to loom network — loom sdk.* 2019. URL: https://loomx.io/developers/en/intro-to-loom.html.

[16] Jonas David Nick. *Liquid: A Bitcoin Sidechain*. 2020. URL: https://blockstream.com/assets/downloads/pdf/liquid-whitepaper.pdf.

[17] *Poa-network-whitepaper*. 2018. URL: https://github.com/poanetwork/wiki/wiki/POA-Network-Whitepaper.

[18] J. Chow. *Btc relay*. 2016. URL: http://btcrelay.org/.

[19] N. Rush L. Luu and N. Lin. *Peacerelay: Connecting the many, ethereum blockchains*. 2019. URL: https://github.com/KyberNetwork/peace-relay.

[20] *Hyperledger cactus whitepaper*. 2020. URL: https://github.com/hyperledger/cactus.

[21] Philipp Frauenthaler et al. *Testimonium: A Cost-Efficient Blockchain Relay*. Feb. 2020. URL: https://arxiv.org/abs/2002.12837.

[22] Iddo Bentov et al. *Tesseract: Real-Time Cryptocurrency Exchange using Trusted Hardware*. Cryptology ePrint Archive, Paper 2017/1153. https://eprint.iacr.org/2017/1153. 2017. URL: https://eprint.iacr.org/2017/1153.

[23] Jeff Burdges et al. *Overview of Polkadot and its Design Considerations*. Cryptology ePrint Archive, Paper 2020/641. https://eprint.iacr.org/2020/641. 2020. URL: https://eprint.iacr.org/2020/641.

[24] J. Kwon and E. Buchman. *Cosmos whitepaper*. 2019. URL: https://v1.cosmos.network/resources/whitepaper.

[25] *Wanchain: Building super financial markets for the new digital economy*. 2017. URL: https://wanchain.org/files/Wanchain-Whitepaper-EN-version.pdf.

[26] *Ark ecosystem whitepaper*. 2019. URL: https://ark.io/Whitepaper.pdf.

[27] *Quant overledger whitepaper*. 2018. URL: https://uploads-ssl.webflow.com/6006946fee85fda61f666256/60211c93f1cc59419c779c42_Quant_Overledger_Whitepaper_Sep_2019.pdf.

[28] Zhuotao Liu et al. *HyperService: Interoperability and Programmability Across Heterogeneous Blockchains*. London, United Kingdom, 2019. DOI: 10.1145/3319535.3355503. URL: https://arxiv.org/abs/1908.09343.

[29] Gang Wang et al. *SMChain: A Scalable Blockchain Protocol for Secure Metering Systems in Distributed Industrial Plants*. Cryptology ePrint Archive, Paper 2019/1401. https://eprint.iacr.org/2019/1401. 2019. DOI: 10.1145/3302505.3310086. URL: https://eprint.iacr.org/2019/1401.

[30] Eder J. Scheid et al. *PleBeuS: a Policy-based Blockchain Selection Framework*. 2020. DOI: 10.1109/NOMS47738.2020.9110386. URL: https://ieeexplore.ieee.org/document/9110386.

[31] Enrique Fynn, Alysson Bessani, and Fernando Pedone. *Smart Contracts on the Move*. June 2020. DOI: 10.1109/DSN48063.2020.00040. URL: https://arxiv.org/abs/2004.05933.

[32] *Interledger protocol v4*. 2020. URL: https://interledger.org/rfcs/0027-interledger-protocol-4/.

[33] Aleksei Pupyshev et al. *Gravity: a blockchain-agnostic cross-chain communication and data oracles protocol*. July 2020. URL: https://arxiv.org/abs/2007.00966.

[34] Aleksei Pupyshev et al. *SuSy: a blockchain-agnostic cross-chain asset transfer gateway protocol based on Gravity*. Aug. 2020. URL: https://arxiv.org/abs/2008.13515.

[35] Christian Badertscher et al. *Ouroboros Genesis: Composable Proof-of-Stake Blockchains with Dynamic Availability*. Oct. 2018. DOI: 10.1145/3243734.3243848. URL: https://eprint.iacr.org/2018/378.pdf.

[36] Christian Badertscher et al. *Ouroboros Chronos: Permissionless Clock Synchronization via Proof-of-Stake*. Cryptology ePrint Archive, Paper 2019/838. https://eprint.iacr.org/2019/838. 2019. URL: https://eprint.iacr.org/2019/838.

[37] Alexei Zamyatin et al. *SoK: Communication Across Distributed Ledgers*. Cryptology ePrint Archive, Paper 2019/1128. https://eprint.iacr.org/2019/1128. 2019. URL: https://eprint.iacr.org/2019/1128.

[38] Aggelos Kiayias et al. *Ouroboros: A Provably Secure Proof-of-Stake Blockchain Protocol*. Cryptology ePrint Archive, Paper 2016/889. https://eprint.iacr.org/2016/889. 2016. URL: https://eprint.iacr.org/2016/889.

[39] Miguel Castro. *Practical Byzantine Fault Tolerance*. Apr. 2001. URL: https://pmg.csail.mit.edu/papers/osdi99.pdf.

[40] Bernardo David et al. *Ouroboros Praos: An adaptively-secure, semi-synchronous proof-of-stake protocol*. Cryptology ePrint Archive, Paper 2017/573. https://eprint.iacr.org/2017/573. 2017. URL: https://eprint.iacr.org/2017/573.

[41] Satoshi Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. May 2009. URL: http://www.bitcoin.org/bitcoin.pdf.

[42] Eleftherios Kokoris-Kogias et al. *Enhancing Bitcoin Security and Performance with Strong Consistency via Collective Signing*. 2016. arXiv: 1602.06997 [cs.CR]. URL: https://arxiv.org/abs/1602.06997.

[43] Sunny King and Scott Nadal. *PPCoin: Peer-to-Peer Crypto-Currency with Proof-of-Stake*. 2012. URL: https://api.semanticscholar.org/CorpusID:42319203.

[44] Yossi Gilad et al. *Algorand: Scaling Byzantine Agreements for Cryptocurrencies*. Cryptology ePrint Archive, Paper 2017/454. https://eprint.iacr.org/2017/454. 2017. URL: https://eprint.iacr.org/2017/454.

[45] Silvio Micali, Salil Vadhan, and Michael Rabin. *Verifiable Random Functions*. USA, 1999. URL: https://ieeexplore.ieee.org/document/814584.

[46] Moni Naor and Asaf Ziv. *Primary-Secondary-Resolver Membership Proof Systems*. Cryptology ePrint Archive, Paper 2014/905. https://eprint.iacr.org/2014/905. 2014. URL: https://eprint.iacr.org/2014/905.

[47] Christian Badertscher et al. *On UC-Secure Range Extension and Batch Verification for ECVRF*. Cryptology ePrint Archive, Paper 2022/1045. https://eprint.iacr.org/2022/1045. 2022. URL: https://eprint.iacr.org/2022/1045.

[48] Claus Schnorr. *Efficient signature generation by smart cards*. Jan. 1991. DOI: 10.1007/BF00196725. URL: https://link.springer.com/article/10.1007/BF00196725.

[49] Ewa Syta et al. *Keeping Authorities "Honest or Bust" with Decentralized Witness Cosigning*. May 2016. DOI: 10.1109/SP.2016.38. URL: https://arxiv.org/abs/1503.08768.

[50] K. Itakura. *A public-key cryptosystem suitable for digital multisignatures*. 1983. URL: https://api.semanticscholar.org/CorpusID:60170133.

[51] Olivier Bégassat et al. *Handel: Practical Multi-Signature Aggregation for Large Byzantine Committees*. 2019. arXiv: 1906.05132 [cs.DC]. URL: https://arxiv.org/abs/1906.05132.

[52] Tal Malkin, Daniele Micciancio, and Sara Miner. *Composition and Efficiency Tradeoffs for Forward-Secure Digital Signatures*. Cryptology ePrint Archive, Paper 2001/034. https://eprint.iacr.org/2001/034. 2001. URL: https://eprint.iacr.org/2001/034.

[53] Tal Malkin, Daniele Micciancio, and Sara Miner. *Efficient generic forward-secure signatures with an unbounded number of time periods*. Amsterdam, The Netherlands: IACR, Apr. 2002. URL: https://cseweb.ucsd.edu/~daniele/papers/MMM.html.

Appendices

A A Complete Description of the Spectrum Protocol

The purpose of this section is to formally specify the code of the Spectrum protocol that each participant executes. Each party P is assigned a session ID, sid. Party is connected to all global setups and functionalities with which it shares the same session ID.

A.1 The Main Protocol

Spectrum protocol uses a number of functionalities, namely, $\mathcal{G}_{\text{Ledger}}, \mathcal{G}_{\text{ImpLClock}}$ and $\mathcal{F}^{\Delta}_{\text{N-MC}}$ which are described in detail in [36].

All protocol participants use the imperfect local clocks functionality $\mathcal{G}_{\text{ImpLClock}}$ to proceed at approximately the same speed with the upper bound Δ^{clock} on the drift between any two honest parties. We also assume a diffusion network in which all messages sent by honest parties are guaranteed to be fetched by other protocol participants after a specific delay Δ^{net}. Additionally, the network guarantees that once a message has been fetched by an honest party, this message is fetched by any other honest party within a delay of at most Δ^{net}. We will use a broadcasting network for message diffusion described by the functionality $\mathcal{F}^{\Delta}_{\text{N-MC}}$.

The main protocol is as follows:

Protocol $\mathsf{Spectrum}(P, \text{sid}; \mathcal{G}_{\text{Ledger}}, \mathcal{G}_{\text{ImpLClock}}, \mathcal{F}^{\Delta}_{\text{N-MC}})$

```
Global variables:
  – Read-only: $R, l_{\text{VRF}}, K_{\text{f}}, K_{\text{g}}, \mathbf{S}^{\text{set}} = \{S^{\text{id}}_k\}^K_{k=1}, \mathbf{f}_{\text{lead}} = \{f^{\text{lead}}_k\}^K_{k=1}, \mathbf{f}_{\text{cons}} = \{f^{\text{cons}}_k\}^K_{k=1}$.
    // Hereinafter, the indices and values of epochs and slots are
    // interchangeable, i.e. e_n = n, s_j = j. Also, the id of the
    // k-th connected system S_k usualy is simply denoted by k.
  – Read-write: $v^{\text{vrf}}_P, v^{\text{kes}}_P, sl_j, e_n, \mathbf{S}^{\text{set}}_P, \mathbf{T}^{\text{cons}}, \mathbf{T}^{\text{lead}}, \mathbf{T}^{\text{sync}}$,
    $\mathsf{localTime}, \mathsf{lastTick}, \mathsf{EpochUpdate}(.), \mathsf{state}_j, \mathsf{buffer}, \mathsf{syncBuffer}, \mathsf{isSync}$,
    $\mathsf{fetchCompleted}, \mathsf{futureChains}$.
    // Bold font upper denotes the sets corresponding to the set
    // of external systems (with id-s in the S^set) to which the
    // participant P is connected.
Interacting with the main Ledger: Upon receiving a ledger-specific input $I$ verify
first that all resources are available. If not all resources are available, then ignore the
input, otherwise execute one of the following steps depending on the input $I$:
  – If $I = (\textsc{submit}, \text{sid}, \mathsf{tx})$ :
    set $\mathsf{buffer} \leftarrow \mathsf{buffer}||\mathsf{tx}$;
    send $(\textsc{multicast}, \text{sid}, \mathsf{tx})$ to $\mathcal{F}^{\Delta}_{\text{N-MC}}$.
  – If $I = (\textsc{maintain-ledger}, P, \text{sid})$ :
    invoke the protocol $\mathsf{LedgerMaintenance}(P, R, \text{sid}, \mathcal{C}_{\text{loc}})$;
    if LedgerMaintenance halts then halt the Spectrum protocol execution and ignore
    all future inputs.
  – If $I = (\textsc{read}, \text{sid})$ :
    return actual local chain state;
```

– **If** $I = (\textsf{EXPORT-TIME}, \mathrm{sid})$:

if $\mathsf{isSync} = \mathsf{false}$ then return false to party P;

Otherwise call $\mathsf{UpdateTime}(P, R)$ and do:

1. Set the highest epoch value $e_n \leftarrow \mathsf{EpochUpdate}(.)$.
2. Return $(\mathsf{localTime}, e_n)$ to the caller.

Handling calls to the shared setup:

– **If** $I = (\textsf{CLOCK-GET}, \mathrm{sid}_C)$: forward it to $\mathcal{G}_{\mathrm{ImpLClock}}$ and return its response.

– **If** $I = (\textsf{CLOCK-UPDATE}, \mathrm{sid}_C)$: record that a clock-update was received in the current round. If the party is registered to all its setups, then do nothing further. Otherwise, do the following operations before concluding this round:

1. If this instance is currently time-aware but otherwise stalled or offline, then set $\mathsf{localTime} \leftarrow \mathsf{UpdateTime}(P, R)$ and update the KES signing key using $\mathcal{F}_{\mathrm{KES}}$. If the party has passed a synchronization slot, then set $\mathsf{isSync} \leftarrow \mathsf{false}$.
2. If this instance is only stalled but $\mathsf{isSync} = \mathsf{true}$, then additionally fetch actual chains, extract all new synchronization beacons from the fetched chains, record their arrival times and set $\mathsf{fetchCompleted} \leftarrow \mathsf{true}$. Any unfinished interruptible execution of this round is marked as completed.
3. Forward $(\textsf{CLOCK-UPDATE}, \mathrm{sid}_C)$ to $\mathcal{G}_{\mathrm{ImpLClock}}$ to finally conclude the round.

– **If** $I = (\textsf{EVAL}, x)$: forward x to the $\mathcal{H}$ and output the response.

A.2 Fetching information, stake distribution and time update

Time Update.

Protocol $\mathsf{UpdateTime}(P, R)$

```
   //NB: Only executed if time-aware.
1: Send $(\textsf{CLOCK-GET}, \mathrm{sid}_C)$ to $\mathcal{G}_{\mathrm{ImpLClock}}$ and parse tick from the response.
2: if $\mathsf{lastTick} \neq \mathsf{tick}$ then
3:     Set $\mathsf{lastTick} \leftarrow \mathsf{tick}$.
4:     Set $\mathsf{localTime} \leftarrow \mathsf{localTime} + 1$.
5:     Set $\mathsf{fetchCompleted} \leftarrow \mathsf{false}$.
6: end if
7: Set $e \leftarrow \lceil \mathsf{localTime} / R \rceil$.
8: Set $sl \leftarrow \mathsf{localTime}$.
```

Synchronization Procedure. The synchronization procedure runs on epoch boundary to synchronize time between all committee members.

Protocol
$\mathsf{SyncProc}(P, \mathrm{sid}, R, K_\mathrm{f}, K_\mathrm{g}, \mathbf{S}_\mathrm{id} = \{S_k^{\mathrm{id}}\}_{k=1}^K, \mathbf{f}_{\mathrm{lead}} = \{f_k^{\mathrm{lead}}\}_{k=1}^K, \mathbf{f}_{\mathrm{cons}} = \{f_k^{\mathrm{cons}}\}_{k=1}^K)$

```
1: Set $i \leftarrow \lceil \mathsf{localTime} / R \rceil$.
2: if (not $\mathsf{EpochUpdate}(i) = \mathsf{Done}$) then
3:     $\mathsf{EpochUpdate}(i) \leftarrow \mathsf{Done}$.
4:     Parse $\mathbf{b}^{\mathrm{set}\prime}_i \leftarrow \mathcal{C}_{\mathrm{loc}}[(i-1) \cdot R + 2/3 \cdot R]$.
5:     Let $j \leftarrow i - 1$.
6:     $\mathbf{b}^{\mathrm{set}}_i \leftarrow \{b^{\mathrm{sync}} | b^{\mathrm{sync}} \in \mathbf{b}^{\mathrm{set}\prime}_i \wedge b^{\mathrm{sync}}.\mathsf{get}(sl) \in [R \cdot j, R \cdot j + 1/6 \cdot R]\}$.
7:     for all $b^{\mathrm{sync}} \in \mathbf{b}^{\mathrm{set}}_i$ do
8:         if $b^{\mathrm{sync}} \in \mathsf{syncBuffer}$ then
9:             Parse $b^{\mathrm{sync}}$ as $(v^{\mathrm{vrf}}, sl, \pi^{\mathrm{sl}}, sl^{\mathrm{rec}})$.
```

```
10:             Set $\text{diff}_b = sl - sl^{\text{rec}}$.
11:         else
12:             $\mathbf{b}_i^{\text{set}} \leftarrow \mathbf{b}_i^{\text{set}} \,/\, \{b^{\text{sync}}\}$.
13:         end if
14:     end for
15:     Set $\mathsf{shift}_i \leftarrow \mathsf{med}\{\text{diff}_b | b^{\text{sync}} \in \mathbf{b}_i^{\text{set}}\})$.
16:     for all $b^{\text{sync}} | b^{\text{sync}} \in \mathbf{b}_i^{\text{set}}\} \wedge b^{\text{sync}}.\mathsf{get}(sl^{\text{rec}}) = (sl', \mathsf{temp})$ do
17:         Set $sl^{\text{rec}} \leftarrow (sl' + \mathsf{shift}_i, \mathsf{final})$.
18:     end for
19:     if $\mathsf{shift}_i > 0$ then // Move fast forward.
20:         Set $\mathsf{newTime} \leftarrow \mathsf{localTime} + \mathsf{shift}_i$.
21:         Set $M_{\text{chains}} \leftarrow M_{\text{sync}} \leftarrow \emptyset$.
22:         while $\mathsf{localTime} < \mathsf{newTime}$ do
23:             $\mathsf{localTime} \leftarrow \mathsf{localTime} + 1$.
24:             Let $\mathbf{N}_0$ be the subsequence of $\mathsf{futureChains}$ | $\forall B \in \mathcal{C}$ : $B.\mathsf{get}(sl) \leq$
    $\mathsf{localTime}$.
25:             for $\mathcal{C} \in \mathbf{N}_0$ do
26:                 Remove $\mathcal{C}$ from $\mathsf{futureChains}$.
27:             end for
28:             Set $\mathcal{C}_{\text{loc}} \leftarrow \mathsf{SelectChain}(P, \text{sid}, \mathcal{C}_{\text{loc}}, R, K_{\text{f}}, K_{\text{g}}, \mathcal{N}_0, \mathbf{S}_{\text{id}}, \mathbf{f}_{\text{lead}})$.
29:             Call $\mathsf{UpdateStakeDistribution}(P, R, K_{\text{f}}, C_{\text{loc}}, \mathbf{S}_{\text{id}}, \mathbf{f}_{\text{lead}}, \mathbf{f}_{\text{cons}})$.
30:             Call $\mathsf{LedgerMaintenance}(P, \text{sid}, \mathcal{C}_{\text{loc}}, R, K_{\text{f}}, K_{\text{g}}, \mathbf{S}_{\text{id}}, \mathbf{f}_{\text{lead}}, \mathbf{f}_{\text{cons}})$
              but instead of broadcasting new chains and beacons, add them to
              the local sets $M_{\text{chains}}$ and $M_{\text{sync}}$ respectively.
31:         end while
32:         Broadcast $M_{\text{chains}}$ and $M_{\text{sync}}$.
33:     else if $\mathsf{shift}_i < 0$ then // Need to wait.
34:         Set $t_{\text{work}} \leftarrow \mathsf{localTime}$.
35:         Set $\mathsf{localTime} \leftarrow \mathsf{localTime} + \mathsf{shift}_i$.
36:     end if
37: end if
```

Updating stake distribution. The stake distributions for epochs are defined in the local chain (and all associated state-variables), and are computed as follows:

Protocol $\mathsf{UpdateStakeDistribution}(P, R, K_{\text{f}}, C_{\text{loc}}, \mathbf{S}_{\text{id}} = \{S_k^{\text{id}}\}_{k=1}^K, \mathbf{f}_{\text{lead}} = \{f_k^{\text{lead}}\}_{k=1}^K, \mathbf{f}_{\text{cons}} = \{f_k^{\text{cons}}\}_{k=1}^K)$

```
 1: Set $e_n \leftarrow \lceil sl \,/\, R \rceil$.
    // Main ledger state_m is calculated according to
    // the last block produced up to m-th slot.
 2: Parse $\mathsf{state}_{(n-2)\cdot R} \leftarrow C_{\text{loc}}$.
 3: Parse $\mathsf{state}_{(n-4)\cdot R} \leftarrow C_{\text{loc}}$.
    // Set epoch randomness:
 4: Set $\eta_{n-2} \leftarrow \mathcal{F}_{\text{LB}}(e_{n-2}, C_{\text{loc}})$. // For the consensus lottery.
 5: Set $\eta_n \leftarrow \mathcal{F}_{\text{LB}}(e_n, C_{\text{loc}})$. // For the leader and sync lotteries.
    // Update stakeholders distribution for the consensus group lottery:
 6: for $\forall S_k^{\text{id}} \in \mathbf{S}_{\text{id}}$ do
 7:     Parse verified and equipped with $\mathcal{F}_{\text{ConnSys}}^k$ functionality stakeholders distribution
    $S_k^{\text{ver},n-4}$ from $\mathsf{state}_{(n-4)\cdot R}$ .
 8: end for
    // Update stakeholders distributions for the leader lottery:
 9: for $\forall S_k^{\text{id}} \in \mathbf{S}_{\text{id}}$ do
```

```
10:     Parse $k$-th committee stakeholders distribution $S_k^{\text{cons},n-2}$
            from the $\mathsf{state}_{(n-2)\cdot R}$.
11: end for
    // Update stakeholders distribution for the synchronization lottery:
12: Parse all committees stakeholders distribution $S^{\text{cons},n-2}$ from the $\mathsf{state}_{(n-2)\cdot R}$.
    // Here and below S_k^{\text{id}} is denoted simply by index k.
    // Set lotteries thresholds:
13: for $\forall S_k^{\text{id}} \in \mathbf{S}_{\text{id}}$ do
14:     Calculate relative stake $\alpha_{P,\text{ver}}^{n-3}$ using $S_k^{\text{ver},n-3}$.
15:     Calculate relative stake $\alpha_{P,\text{k}}^{n-2}$ using $S_k^{\text{cons},n-2}$.
16:     Set consensus group lottery threshold for $k$-th committee as $T_{P,n-2}^{\text{cons},k} = \phi_{f_k^{\text{cons}}}(\alpha_{P,\text{ver}}^{n-3})$.
17:     Set leader lottery threshold as $T_{P,n}^{\text{lead},k} = \phi_{f_k^{\text{lead}}}(\alpha_{P,\text{k}}^{n-2})$.
18: end for
19: Calculate relative stake $\alpha_P^{n-2}$ using $S^{\text{cons},n-2}$.
20: Set synchronization lottery threshold as $T_{P',n}^{\text{sync}} = 2^{l_{\text{VRF}}} \cdot \alpha_P^{n-2}$.
21: return $(e_n, \mathsf{state}_{(n-3)\cdot R}, V_{\text{ver}}, \eta_n, \eta_{n-2}, \{T_{P,n-2}^{\text{cons},k}\}_{k=1}^K, \{T_{P,n}^{\text{lead},k}\}_{k=1}^K, T_{P,n}^{\text{sync}})$
```

Processing beacons. The following procedure records and processes beacons, their arrival times, and filters out invalid beacons:

Protocol $\mathsf{ProcessBeacons}(P, \text{sid}, R, l_{\text{VRF}}, K_{\text{f}}, \mathcal{C}_{\text{loc}}, \mathbf{b}^{\text{set}} = \{b_n^{\text{sync}}\}_{n=1}^N)$

```
 1: for all $b_n^{\text{sync}} \in \mathbf{b}^{\text{set}} | b_n^{\text{sync}}.\mathsf{get}(sl_n^{\text{rec}}) = \perp$ do
 2:     $\mathsf{syncBuffer} \leftarrow \mathsf{syncBuffer} \cup \{b_n^{\text{sync}}\}$.
 3:     Set $e \leftarrow \lfloor b_n^{\text{sync}}.\mathsf{get}(sl) \ / \ R \rfloor$.
 4:     if $\mathsf{isSync} \wedge (\mathsf{epochUpdate(e - 1)} = \mathsf{Done})$ then
 5:         Set $sl_n^{\text{rec}} \leftarrow (\mathsf{localTime},\ \mathsf{final})$.
 6:     else
         Set $sl_n^{\text{rec}} \leftarrow (\mathsf{localTime},\ \mathsf{temp})$.
 7:     end if
 8: end for
    // Buffer cleaning. Keep one representative arrival time.
 9: if $\mathsf{isSync}$ then
10:     Set $\mathsf{syncBuffer}_{\text{valid}} \leftarrow \{b^{\text{sync}\prime} \in \mathsf{syncBuffer} | \mathsf{ValidBeacon}(P, \text{sid}, R, l_{\text{VRF}},$
            $K_{\text{f}}, b^{\text{sync}\prime}, \mathcal{C}_{\text{loc}}) = \mathsf{true}\}$.
11:     for all $b^{\text{sync}} \in \mathsf{syncBuffer}_{\text{valid}}$ do
12:         Parse $b^{\text{sync}}$ as $(v^{\text{vrf}}, sl, \pi^{\text{sl}})$.
13:         Set $Q \leftarrow \{b^{\text{sync}\prime} \in \mathsf{syncBuffer}_{\text{valid}} | v^{\text{vrf}} = v^{\text{vrf}\prime} \wedge sl = sl')\}$.
14:         Set $b^{\text{sync, min}} \leftarrow \min_{sl}(Q)$.
15:         Remove from the $\mathsf{syncBuffer}$ all beacons except the $b^{\text{sync, min}}$.
16:     end for
17: end if
```

A.3 Validity Checks

Block validation. The core procedure to validate an incoming blocks. Block validation implies a procedure for preparing the necessary constants to check the validity of the party P' to issue the block. The preparation algorithm is described below:

Protocol $\mathsf{PrepareForBlockValidation}(P, \text{sid}, sl, R, l_{\text{VRF}}, K_{\text{f}}, C_{\text{loc}}, v_{P'}^{\text{vrf}}, \mathbf{S}_{\text{id}} = \{S_k^{\text{id}}\}_{k=1}^K, \mathbf{f}_{\text{lead}} = \{f_k^{\text{lead}}\}_{k=1}^K, \mathbf{f}_{\text{cons}} = \{f_k^{\text{cons}}\}_{k=1}^K)$

```
    // Parse and calculate all necessary values for block validation.
 1: Set $e_n \leftarrow \lceil sl \ / \ R \rceil$.  // Main ledger state_m is calculated according to
    // the last block produced up to m-th slot.
 2: Parse $\mathsf{state}_{(n-2)\cdot R} \leftarrow C_{\mathrm{loc}}$.
 3: Parse $\mathsf{state}_{(n-4)\cdot R} \leftarrow C_{\mathrm{loc}}$.
    // Set epoch randomness:
 4: Set $\eta_{n-2} \leftarrow \mathcal{F}_{\mathrm{LB}}(e_{n-2}, C_{\mathrm{loc}})$. // For the consensus lottery.
 5: Set $\eta_n \leftarrow \mathcal{F}_{\mathrm{LB}}(e_n, C_{\mathrm{loc}})$. // For the leader and sync lotteries.
    // Stakeholders distribution used for the consensus group lottery:
 6: Parse verified and equipped with $\mathcal{F}^k_{\mathrm{ConnSys}}$ functionality stakeholders distribution
    $S_k^{\mathrm{ver},n-4}$ from $\mathsf{state}_{(n-4)\cdot R}$.
    // Stakeholders distribution used for the leader lottery:
 7: Parse $k$-th committee stakeholders distribution $S_k^{\mathrm{cons},n-2}$ from the $\mathsf{state}_{(n-2)\cdot R}$.
    // Stakeholders distribution used for the synchronization lottery:
 8: Parse all committees stakeholders distribution $S^{\mathrm{cons},n-2}$ from the $\mathsf{state}_{(n-2)\cdot R}$.
    // Here and below S_k^{\text{id}} is denoted simply by index k.
    // Set lotteries thresholds:
 9: Calculate participant's relative stake $\alpha^{n-3}_{P',\mathrm{ver}}$ using $S_k^{\mathrm{ver},n-3}$.
10: Calculate participant's relative stake $\alpha^{n-2}_{P',\mathrm{k}}$ using $S_k^{\mathrm{cons},n-2}$.
11: Calculate participant's relative stake $\alpha^{n-2}_{P'}$ using $S^{\mathrm{cons},n-2}$.
12: Set participant's consensus group lottery threshold for $k$-th committee as $T^{\mathrm{cons},k}_{P',n-2} =$
    $\phi_{f_k^{\mathrm{cons}}}(\alpha^{n-3}_{P',\mathrm{ver}})$.
13: Set participant's leader lottery threshold as $T^{\mathrm{lead},k}_{P',n} = \phi_{f_k^{\mathrm{lead}}}(\alpha^{n-2}_{P',\mathrm{k}})$.
14: Set participant's synchronization lottery threshold as $T^{\mathrm{sync}}_{P',n} = 2^{l_{\mathrm{VRF}}} \cdot \alpha^{n-2}_{P'}$.
15: return $(e_n, \mathsf{state}_{(n-3)\cdot R}, V_{\mathrm{ver}}, \eta_n, \eta_{n-2}, T^{\mathrm{cons},k}_{P',n-2}, T^{\mathrm{lead},k}_{P',n}, T^{\mathrm{sync}}_{P',n})$
```

Main Block Validation Protocol is as follows:

Protocol $\mathsf{IsValidBlock}(P, \mathrm{sid}, R, l_{\mathrm{VRF}}, K_{\mathrm{f}}, \mathcal{C}_{\mathrm{loc}}, B, \mathbf{S}_{\mathrm{id}} = \{S_k^{\mathrm{id}}\}_{k=1}^K, \mathbf{f}_{\mathrm{lead}} = \{f_k^{\mathrm{lead}}\}_{k=1}^K, \mathbf{f}_{\mathrm{cons}} = \{f_k^{\mathrm{cons}}\}_{k=1}^K)$

```
    // All indexes except the epoch index are omitted.
 1: Parse $B$ as $(h, sl, v^{\mathrm{vrf}}, S_k^{\mathrm{id}}, \pi^{\mathrm{sl}}, \sigma_{\mathrm{KES}})$.
    // Value h above is a block body hash.
 2: Parse $\mathsf{state} \leftarrow C_{\mathrm{loc}}$.
    // Prepare constants:
 3: Select $(f_k^{\mathrm{lead}}, f_k^{\mathrm{cons}})$ related to the given $S_k^{\mathrm{id}}$.
 4: Set $\mathsf{preparation\_out} \leftarrow \mathsf{PrepareForBlockValidation}(P, \mathrm{sid}, sl, R, l_{\mathrm{VRF}}, K_{\mathrm{f}}, C_{\mathrm{loc}},$
    $v^{\mathrm{vrf}}, S_k^{\mathrm{id}}, f_k^{\mathrm{lead}}, f_k^{\mathrm{cons}})$.
 5: Set $(e_n, \mathsf{state}_{(n-4)\cdot R}, V_{\mathrm{ver}}, \eta_n, \eta_{n-2}, T^{\mathrm{cons}}, T^{\mathrm{lead}}, T^{\mathrm{sync}}) = \mathsf{preparation\_out}$.
    // Check consensus membership:
 6: Parse $\mathbf{S}_{\mathrm{id},P'}$ related to the given $v^{\mathrm{vrf}}$ from $V_{\mathrm{ver}}$.
    // P' above is the same authority as v^vrf. We used this notation
    // to separete leader's S_{id} set from the global one.
 7: Set $\mathsf{valid\_committee} \leftarrow (S_k^{\mathrm{id}} \in \mathbf{S}_{\mathrm{id},P'}) \wedge (\mathbf{S}_{\mathrm{id},P'} \subset \mathbf{S}_{\mathrm{id}})$.
 8: Parse $\pi^{\mathrm{e}}$ related to $v^{\mathrm{vrf}}$ from the $\mathsf{state}$.
 9: Set $\mathsf{valid\_epoch\_proof} \leftarrow \mathcal{F}_{\mathrm{VRF}}.\mathsf{verify}(v^{\mathrm{vrf}}, \mathcal{H}(\eta_{n-2}||e_n), \pi^{\mathrm{e}})$.
10: Extract the random value $r^{\mathrm{e}} \leftarrow \pi^{\mathrm{e}}$.
11: Set $y^{\mathrm{cons}} \leftarrow \mathcal{H}(r^{\mathrm{e}}||\mathsf{CONS}||S_k^{\mathrm{id}})$.
12: Set $\mathsf{valid\_member} \leftarrow \mathsf{valid\_committee} \wedge \mathsf{valid\_epoch\_proof} \wedge$
        $(\mathrm{v}^{\mathrm{vrf}} \in V_{\mathrm{ver}}) \wedge (y^{\mathrm{cons}} < T^{\mathrm{cons}})$.
    // Check the leadership:
```

```
13: Set valid_slot_proof $\leftarrow \mathcal{F}_{\mathrm{VRF}}$.verify( $v^{\mathrm{vrf}}, \mathcal{H}(\eta_n||sl), \pi^{\mathrm{sl}}$).
14: Extract the random value $r^{\mathrm{sl}} \leftarrow \pi^{\mathrm{sl}}$.
15: Set $y^{\mathrm{lead}} \leftarrow \mathcal{H}(r^{\mathrm{sl}}||\mathsf{LEAD}||S_k^{\mathrm{id}})$.
16: Set valid_leader $\leftarrow (y^{\mathrm{lead}} < T^{\mathrm{lead}}) \wedge$ valid_slot_proof.
    // Check KES signature:
17: Parse $v^{\mathrm{kes}}$ from $V_{\mathrm{ver}}$.
18: Set state_hash $\leftarrow \mathcal{H}$(state).
19: Set $\pi_h^{\mathrm{sl}} \leftarrow \mathcal{H}(\pi^{\mathrm{sl}})$.
20: Set valid_signature $\leftarrow \mathcal{F}_{\mathrm{KES}}$.verify( $\mathcal{H}(h||\mathsf{state\_hash}||sl||\pi_h^{\mathrm{sl}}), \sigma_{\mathrm{KES}}, v^{\mathrm{kes}}$).
    // Check synchronization beacons:
21: Parse $\mathbf{b}^{\mathrm{set}}$ from state.
22: if $\exists b^{\mathrm{sync}} \in \mathbf{b}^{\mathrm{set}} : sl > (e_n - 1) \cdot R + 2 \cdot R/3$ then
23:     Set valid_sync $\leftarrow$ false.
24: else if $\exists b^{\mathrm{sync}} \in B : (b^{\mathrm{sync}}.\mathsf{get}(sl) > sl) \vee (b^{\mathrm{sync}}.\mathsf{get}(sl) \notin [(e_n - 1) \cdot R + 1, e_n \cdot R])$
    then
25:     Set valid_sync $\leftarrow$ false.
26: end if
27: for each $b^{\mathrm{sync}} \in B$ do
28:     Parse $b^{\mathrm{sync}}$ as $(v^{\mathrm{vrf}'}, sl', \pi^{\mathrm{sl}'})$.
29:     if $\mathcal{C}_{\mathrm{loc}}$ contains more than one beacon with $(v^{\mathrm{vrf}'}, sl', .)$ then
30:         Set valid_sync $\leftarrow$ false.
31:     end if
32:     Set valid_slot_proof $\leftarrow \mathcal{F}_{\mathrm{VRF}}$.verify( $v^{\mathrm{vrf}}, \mathcal{H}(\eta_n||sl'), \pi^{\mathrm{sl}'}$).
33:     Extract the random value $r^{\mathrm{sl}'} \leftarrow \pi^{\mathrm{sl}'}$.
34:     Set $y^{\mathrm{sync}} \leftarrow \mathcal{H}(r^{\mathrm{sl}'}||\mathsf{SYNC})$.
35:     Set valid_sync $\leftarrow$ valid_slot_proof $\wedge (y^{\mathrm{sync}} < T^{\mathrm{sync}})$.
36: end for
37: if (valid_parent $\vee$ valid_member $\vee$ valid_proof $\vee$ valid_leader $\vee$ valid_signature $\vee$
    valid_sync) then
38:     return false
39: end if
```

Chain validation. The core procedure to distinguish valid chains from the invalid is as follows:

Protocol IsValidChain($P, \mathrm{sid}, \mathcal{C}, R, l_{\mathrm{VRF}}, K_{\mathrm{f}}, \mathbf{S}_{\mathrm{id}} = \{S_k^{\mathrm{id}}\}_{k=1}^K, \mathbf{f}_{\mathrm{lead}} = \{f_k^{\mathrm{lead}}\}_{k=1}^K, \mathbf{f}_{\mathrm{cons}} = \{f_k^{\mathrm{cons}}\}_{k=1}^K$)

```
 1: if $\exists B \in \mathcal{C} : B.\mathsf{get}(sl) >$ localTime then
 2:     return false
 3: end if
 4: for each $e_j \in \mathcal{C}$ do // meaning for all unique e_j values for which
    // there are blocks in the C.
 5:     for each block $B \in \mathcal{C} \mid B.\mathsf{get}(sl) \in e_j$ do
            // Check parent:
 6:         Set valid_parent $\leftarrow (\mathcal{H}(B^{-1}) = h) \wedge (B^{-1}.\mathsf{get}(sl) < sl)$,
            where $B^{-1}$ is the last block before $B$.
 7:         Set valid_block $\leftarrow$ IsValidBlock($P, \mathrm{sid}, R, l_{\mathrm{VRF}}, K_{\mathrm{f}}, \mathcal{C}_{\mathrm{loc}}, B, \mathbf{S}_{\mathrm{id}}$,
            $\mathbf{f}_{\mathrm{lead}}, \mathbf{f}_{\mathrm{cons}}$).
 8:         if (valid_block $\wedge$ valid_parent) then
 9:             return false
10:         end if
11:     end for
```

```
12: end for
13: return true
```

The synchronisation beacon validity. Beacons validity is related to chain validity as one has to verify validity of the leadership:

Protocol $\mathsf{ValidBeacon}(P, \mathrm{sid}, R, l_{\mathrm{VRF}}, K_{\mathrm{f}}, b^{\mathrm{sync}}, \mathcal{C}_{\mathrm{loc}})$

```
 1: Parse synchronization beacon $b^{\mathrm{sync}}$ as $(v^{\mathrm{vrf}}_{P'}, sl, \pi^{\mathrm{sl}})$.
 2: Set $e_n \leftarrow \lceil sl \,/\, R \rceil$.
 3: if $\nexists B \in \mathcal{C}_{\mathrm{loc}} | B.\mathsf{get}(\mathsf{sl}) \in e_n$ then
 4:     return false
 5: end if
    // Check synchronization lottery results for patry P':
 6: Set $\eta_n \leftarrow \mathcal{F}_{\mathrm{LB}}(e_n, C_{\mathrm{loc}})$.
 7: Parse $\mathsf{state}_{(n-2)\cdot R} \leftarrow C_{\mathrm{loc}}$.
 8: Parse all committees stakeholders distribution $S^{\mathrm{cons},n-2}$ from the $\mathsf{state}_{(n-2)\cdot R}$.
 9: Calculate participant's relative stake $\alpha^{n-2}_{P'}$ using $S^{\mathrm{cons},n-2}$.
10: Set participant's synchronization lottery threshold as $T^{\mathrm{sync}}_{P',n} = 2^{l_{\mathrm{VRF}}} \cdot \alpha^{n-2}_{P'}$.
11: Set $\mathsf{valid\_slot\_proof} \leftarrow \mathcal{F}_{\mathrm{VRF}}.\mathsf{verify}(\ v^{\mathrm{vrf}}_{P'}, \mathcal{H}(\eta_n || sl), \pi^{\mathrm{sl}})$.
12: Extract the random value $r^{\mathrm{sl}} \leftarrow \pi^{\mathrm{sl}}$.
13: Set $y^{\mathrm{sync}} \leftarrow \mathcal{H}(r^{\mathrm{sl}} || \mathsf{SYNC})$.
14: Set $\mathsf{valid\_sync} \leftarrow (y^{\mathrm{sync}} < T^{\mathrm{sync}}) \wedge \mathsf{valid\_slot\_proof}$.
15: return valid_sync
```

A.4 Chain Selection Rules

Chain selection consists of two steps: filtering out valid chains, and second compare them using the Genesis rule [35].

Maximum Valid Rule. The Genesis chain selection rule:

Algorithm $\mathsf{maxValidChain}(\mathcal{C}_{\mathrm{loc}}, \mathbf{N} = \{\mathcal{C}_i\}_{i=1}^{N}, K_{\mathrm{f}}, K_{\mathrm{g}})$

```
    // Set local chain C_loc as initially maximum valid chain:
 1: Set $\mathcal{C}_{\max} \leftarrow \mathcal{C}_{\mathrm{loc}}$.
 2: for each $\mathcal{C}_i \in \mathbf{N}$ do
 3:     if $\mathcal{C}_i$ forks from $\mathcal{C}_{\max}$ at most $K_{\mathrm{f}}$ blocks then
 4:         if $|\mathcal{C}_i| > |\mathcal{C}_{\max}|$ then
 5:             Set $\mathcal{C}_{\max} \leftarrow \mathcal{C}_i$.
 6:         else if $|\mathcal{C}_i| == |\mathcal{C}_{\max}|$ then
 7:             Set $\mathcal{C}_{\max} \leftarrow \mathsf{maxStakeChain}(\mathcal{C}_i, \mathcal{C}_{\max})$.
 8:         end if
 9:     else
10:         Let $j \leftarrow \max\{j' \geq 0 \mid \mathcal{C}_{\max}$ and $\mathcal{C}_i$ have the same block in $sl_{j'}\}$.
11:         if $|\mathcal{C}_i[j : j + K_{\mathrm{g}}]| > |\mathcal{C}_{\max}[j : j + K_{\mathrm{g}}]|$ then
12:             Set $\mathcal{C}_{\max} \leftarrow \mathcal{C}_i$.
13:         else if $|\mathcal{C}_i| = |\mathcal{C}_{\max}|$ then
14:             Set $\mathcal{C}_{\max} \leftarrow \mathsf{maxStakeChain}(\mathcal{C}_i, \mathcal{C}_{\max})$.
15:         end if
16:     end if
17: end for
18: return $\mathcal{C}_{\max}$
```

Maximum Stake Rule. Rule to resolve conflicts that arise after applying the Genesis rule:

```
Algorithm maxStakeChain($\mathcal{C}_i, \mathcal{C}_{i'}$)
1: if $\sum\{B_k.\mathsf{get}(s), \forall B_k \in \mathcal{C}_i\} > \sum\{B_k.\mathsf{get}(s), \forall B_k \in \mathcal{C}_{i'}\}$ then
   // Used above value s is the stake of the leader
   // who produced the block B_k.
2:     Set $\mathcal{C}_{\max} \leftarrow \mathcal{C}_i$.
3: else  // It is assumed that the input chains are the same size.
4:     Set $\mathcal{C}_{\max} \leftarrow \mathcal{C}_{i'}$.
5: end if
6: return $\mathcal{C}_{\max}$
```

Chain Selection Protocol. The main chain selection protocol is as follows:

```
Protocol SelectChain($P, \mathrm{sid}, \mathcal{C}_{\mathrm{loc}}, R, K_{\mathrm{f}}, K_{\mathrm{g}}, \mathbf{N}_0,$
$\mathbf{S}_{\mathrm{id}} = \{S_k^{\mathrm{id}}\}_{k=1}^K, \mathbf{f}_{\mathrm{lead}} = \{f_k^{\mathrm{lead}}\}_{k=1}^K, \mathbf{f}_{\mathrm{cons}} = \{f_k^{\mathrm{cons}}\}_{k=1}^K$)
1: Initialize $\mathbf{N}_{\mathrm{valid}} \leftarrow \emptyset$.
   // Filter all valid chains:
2: for each $\mathcal{C} \in \mathbf{N}_0$ do
3:     Set is_valid_chain ← IsValidChain($P, \mathrm{sid}, \mathcal{C}, R, l_{\mathrm{VRF}}, K_{\mathrm{f}}, \mathbf{S}_{\mathrm{id}},$
           $\mathbf{f}_{\mathrm{lead}}, \mathbf{f}_{\mathrm{cons}}$).
4:     if is_valid_chain = true then
5:         Update $\mathbf{N}_{\mathrm{valid}} \leftarrow \mathbf{N}_{\mathrm{valid}} \cup \mathcal{C}$.
6:     end if
7: end for
   // Set local chain as maximum valid chain:
8: Set $\mathcal{C}_{\mathrm{loc}} \leftarrow$ maxValidChain($\mathcal{C}_{\mathrm{loc}}, \mathbf{N}_0, K_{\mathrm{f}}, K_{\mathrm{g}}$).
```

A.5 Ledger Maintenance

When a protocol is executed, every party P performs different actions depending on its role and the current local time. The main logic with all necessary actions are included into the main LedgerMaintenance procedure. At different points in time, participants perform auxiliary protocols, which we will describe below.

Evaluation Protocol. In normal protocol execution, each participant performs the following procedure:

```
Protocol EvaluationProcedure($P, \mathrm{sid}, R, sl, \mathcal{C}_{\mathrm{loc}}, \mathbf{S}_{\mathrm{id}}$)
   // Synchronization lottery:
1: Set $y^{\mathrm{sync}} \leftarrow \mathcal{H}(r^{\mathrm{sl}} || \mathsf{SYNC})$.
2: Set valid_sync ← $y^{\mathrm{sync}} < T^{\mathrm{sync}}$.
3: if valid_sync then
4:     if $sl \in [R \cdot n, R \cdot n + 1 / 6 \cdot R]$ then
5:         Set $b^{\mathrm{sync}} \leftarrow (v^{\mathrm{vrf}}, sl, \pi^{\mathrm{sl}})$.
6:         Broadcast $b^{\mathrm{sync}}$ to known peers.
7:     end if
8: end if
9: for each $S_k^{\mathrm{id}} \in \mathbf{S}_{\mathrm{id}}$ do
       // All participant's state constants involved are specific
       to the k-th committee.
```

```
10:     Select $T^{\text{lead}}$ related to the $S_k^{\text{id}}$.
        // Leader lottery:
11:     Set $(r^{\text{sl}}, \pi^{\text{sl}}) \leftarrow \mathcal{F}_{\text{VRF}}.\mathsf{eval}(\eta_n || sl)$.
12:     Set $y^{\text{lead}} \leftarrow \mathcal{H}(r^{\text{sl}} || \mathsf{LEAD} || S_k^{\text{id}})$.
13:     Set $\mathsf{valid\_leader} \leftarrow y^{\text{lead}} < T^{\text{lead}}$.
14:     if $\mathsf{valid\_leader}$ then
15:         Set actual $\mathsf{state}$ extracted from the $\mathsf{buffer}$.
16:         Set $h \leftarrow \mathcal{H}(\mathsf{head}(\mathcal{C}_{\text{loc}}))$. // head(C) gets the latest block from C.
17:         if $sl \in [R \cdot n, R \cdot n + 2 \,/\, 3 \cdot R]$ then
            // Set valid synchronization beacon set as:
18:             $\mathbf{b}^{\text{set}} \leftarrow \{b' \in \mathsf{syncBuffer} | \mathsf{validBeacon}(P, \text{sid}, R, l_{\text{VRF}}, K_{\text{f}}, b', \mathcal{C}_{\text{loc}})$
            $= \mathsf{true}\}$.
19:             for each $b$ in $\mathbf{b}^{\text{set}}$ do
20:                 Set $sl^* \leftarrow b.\mathsf{get}(sl))$.
21:                 Set $v^{\text{vrf}*} \leftarrow b.\mathsf{get}(v^{\text{vrf}})$.
22:                 if $(sl^* > sl) \vee (sl^* \geq (n-1) \cdot R) \vee (\exists b' \in \mathcal{C}_{\text{loc}}|$
            $(b'.\mathsf{get}(v^{\text{vrf}})) = v^{\text{vrf}*} \wedge b'.\mathsf{get}(sl) = sl^*))$ then
23:                     Remove $b$ from the $\mathbf{b}^{\text{set}}$.
24:                 end if
25:             end for
26:         end if
27:         Set $\mathsf{state\_hash} \leftarrow \mathcal{H}(\mathsf{state})$.
28:         Set $\pi_h^{\text{sl}} \leftarrow \mathcal{H}(\pi^{\text{sl}})$.
29:         Set $\sigma_{\text{KES}} \leftarrow \mathcal{F}_{\text{KES}}.\mathsf{sign}(\mathcal{H}(h || \mathsf{state\_hash} || sl || \pi_h^{\text{sl}}))$.
30:         Set $B \leftarrow ((h, sl, \mathsf{state\_hash}, \pi_h^{\text{sl}}), sl, S_k^{\text{id}}, v^{\text{vrf}}, \pi^{\text{sl}}, \sigma_{\text{KES}})$.
31:         Update $\mathcal{C}_{\text{loc}} \leftarrow \mathcal{C}_{\text{loc}} || B$ and broadcast it to all known peers.
32:     end if
33: end for
```

Consensus Lottery Protocol. When moving between epochs, a new consensus group must be selected. To do this, each participant performs the following protocol:

Protocol $\mathsf{ConsensusLottery}(P, \text{sid}, e_n, \mathcal{C}_{\text{loc}}, K_{\text{f}}, K_{\text{g}}, \mathbf{S}_{\text{id}} = \{S_k^{\text{id}}\}_{k=1}^{K}, \mathbf{f}_{\text{lead}} = \{f_k^{\text{lead}}\}_{k=1}^{K}, \mathbf{f}_{\text{cons}} = \{f_k^{\text{cons}}\}_{k=1}^{K})$

```
    // The lottery at the e_n > 2 selects committees for e_{n + 3}
 1: Parse $\mathsf{state}_{(n-2)\cdot R} \leftarrow C_{\text{loc}}$.
 2: for each $S_k^{\text{id}} \in \mathbf{S}_{\text{id}}$ do
 3:     Set $(r^{\text{e}}, \pi^{\text{e}}) \leftarrow \mathcal{F}_{\text{VRF}}.\mathsf{eval}(\mathcal{H}(\eta_n || e_n))$.
 4:     Parse $k$-th committee stakeholders distribution $S_k^{n-2}$ from the $\mathsf{state}_{(n-2)\cdot R}$.
 5:     Calculate participant's relative stake $\alpha_{P'}^{n-2}$ using $S_k^{n-2}$.
 6:     Set participant's consensus group lottery threshold for $k$-th committee
          as $T_{P',n-2}^{k} = \phi_{f_k^{\text{cons}}}(\alpha_{P'}^{n-2})$.
 7:     Set $y^{\text{cons}} \leftarrow \mathcal{H}(|| r^{\text{e}} || \mathsf{CONS} || S_k^{\text{id}})$.
 8:     Set $\mathsf{is\_member} \leftarrow y^{\text{cons}} < T_{P',n-2}^{k}$.
 9:     if $\mathsf{is\_member} = \mathsf{true}$ then
10:         Broadcast message with consensus membership proof
              $\mathsf{ConsLotteryResTx}(e_n, v_P^{\text{vrf}}, S_k^{\text{id}}, \pi^{\text{e}})$.
11:     end if
12: end for
    // Commit to participate in the e_{n + 3} consensus lottery:
13: Update the verification information broadcasting the $\mathsf{VerificationUpdTx}(v^{\text{vrf}}, \mathbf{S}_P^{\text{set}})$.
```

Main Ledger Maintenance Protocol.

Protocol $\mathsf{LedgerMaintenance}(P, \mathrm{sid}, \mathcal{C}_{\mathrm{loc}}, R, K_{\mathrm{f}}, K_{\mathrm{g}}, \mathbf{S}_{\mathrm{id}} = \{S_k^{\mathrm{id}}\}_{k=1}^K, \mathbf{f}_{\mathrm{lead}} = \{f_k^{\mathrm{lead}}\}_{k=1}^K, \mathbf{f}_{\mathrm{cons}} = \{f_k^{\mathrm{cons}}\}_{k=1}^K)$

```
    // Normal operation:
 1: Fetch the latest protocol data: $(\{\mathcal{C}_m\}_{m=1}^M, \{\mathsf{tx}_k\}_{i=k}^K)$.
 2: Add $\{\mathcal{C}_m\}_{m=1}^M$ into futureChains.
 3: Add $\{\mathsf{tx}_k\}_{k=1}^K$ into buffer.
 4: Call UpdateTime$(P, R)$.
    // Process arrived synchronisation beacons:
 5: Extract beacons $\mathbf{b}^{\mathrm{set}} \leftarrow \{b_n^{\mathrm{sync}}\}_{n=1}^N$ contained in $\{\mathcal{C}_m\}_{m=1}^M$ and not yet contained in
    syncBuffer.
 6: Call ProcessBeacons$(P, \mathrm{sid}, R, l_{\mathrm{VRF}}, K_{\mathrm{f}}, \mathcal{C}_{\mathrm{loc}}, \mathbf{b}^{\mathrm{set}})$.
    // Filter chains:
 7: Let $\mathbf{N}_0$ be the subsequence of futureChains $\mid \forall B \in \mathcal{C}: B.\mathsf{get}(sl) \leq$ localTime.
 8: for $\mathcal{C} \in \mathbf{N}_0$ do
 9:     Remove $\mathcal{C}$ from futureChains.
10: end for
11: Set $\mathcal{C}_{\mathrm{loc}} \leftarrow$ SelectChain$(P, \mathrm{sid}, \mathcal{C}_{\mathrm{loc}}, R, K_{\mathrm{f}}, K_{\mathrm{g}}, \mathbf{N}_0, \mathbf{S}_{\mathrm{id}}, \mathbf{f}_{\mathrm{lead}}, \mathbf{f}_{\mathrm{cons}})$
    // Perform actions according to the current local
    // stage of the protocol:
12: Set $sl \leftarrow$ localTime.
13: if $sl < sl^{\mathrm{work}}$ then
14:     Call EvaluationProcedure$(P, \mathrm{sid}, R, sl, \mathsf{buffer}, \mathsf{syncBuffer}, \mathcal{C}_{\mathrm{loc}}, \mathbf{S}_{\mathrm{id}})$.
15:     Set $sl^{\mathrm{work}} \leftarrow sl$.
16:     if $sl \bmod R = 0$ then
17:         Call UpdateStakeDistribution$(P, R, K_{\mathrm{f}}, C_{\mathrm{loc}}, \mathbf{S}_{\mathrm{id}}, \mathbf{f}_{\mathrm{lead}}, \mathbf{f}_{\mathrm{cons}})$.
18:         Calculate $e_n$ for the given $sl$.
19:         Parse $\mathsf{state}_{(n-2)\cdot R}$ from $C_{\mathrm{loc}}$.
20:         for every party's $P$ $k$-th connected system do
21:             Call ConsensusLottery$(P, \mathrm{sid}, e_n, \mathcal{C}_{\mathrm{loc}}, K_{\mathrm{f}}, K_{\mathrm{g}}, \mathbf{S}_{\mathrm{id}}, \mathbf{f}_{\mathrm{lead}}, \mathbf{f}_{\mathrm{cons}})$.
22:         end for
23:         Call SyncProc$(P, \mathrm{sid}, R, K_{\mathrm{f}}, K_{\mathrm{g}}, \mathbf{S}_{\mathrm{id}}, \mathbf{f}_{\mathrm{lead}}, \mathbf{f}_{\mathrm{cons}})$.
24:     end if
25: end if
```

B List of Symbols

Functionalities:

- ❑ $\mathcal{H}$ – ideal hash function (random oracle).
- ❑ $\mathcal{F}_{\text{VRF}}$ – verifiable random function.
- ❑ $\mathcal{F}_{\text{KES}}$ – key evolving digital signature scheme.
- ❑ $\mathcal{F}_{\text{LB}}$ – leaky beacon.
- ❑ $\mathcal{F}_{\text{AggSig}}$ – collective signature aggregation functionality.
- ❑ $\mathcal{F}_{\text{Init}}$ – functionality providing the genesis block.
- ❑ $\mathcal{F}^{\Delta}_{\text{N-MC}}$ – functionality providing the genesis block.
- ❑ $\mathcal{F}^{k}_{\text{ConnSys}}$ – functionality to interact with k-th connected distributed system S_k.
- ❑ $\mathcal{G}_{\text{ImpLClock}}$ – imperfect local clock functionality.
- ❑ $\mathcal{G}_{\text{PerfLClock}}$ – perfect local clock functionality.
- ❑ $\mathcal{G}_{\text{Ledger}}$ – the ledger functionality.

Main State Variables of The Spectrum protocol:

- ❑ $sl_j \in \mathbb{N}$ – the smallest discrete time unit used in the protocol.
- ❑ $e_n \in \mathbb{N}$ – the largest discrete time unit used in the protocol.
- ❑ $R \in \mathbb{N}$ – epoch boundaries, each epoch consists of R slots.
- ❑ l_{VRF} – the output length of the VRF in bits.
- ❑ U_{c} – transaction confirmation time in slots.
- ❑ $K_{\text{f}} \in \mathbb{N}$ – number of blocks to achieve finality in the L^{+}.
- ❑ $K_{\text{g}} \in \mathbb{N}$ – number of blocks to consider chain growth (is used in chain selection).
- ❑ $\mathbf{S}^{\text{set}} = \{S^{\text{id}}_k\}_{k=1}^{K}$ – set of connected distributed systems' ids (i.e. K committees of validators for each system).
- ❑ $\mathbf{f}_{\text{lead}} = \{f^{\text{lead}}_k\}_{k=1}^{K}$ – set of target number of leaders per slot in each k-th committee.
- ❑ $\mathbf{f}_{\text{cons}} = \{f^{\text{cons}}_k\}_{k=1}^{K}$ – set of target fraction of each k committee members.

Main Spectrum's Consensus Entities, Actors and Variables:

- ❑ L^{+} – the main Spectrum's super-ledger (stores blocks).
- ❑ $L^{\text{loc},k}$ – ledger of the k-th connected distributed system (stores notarized reports).
- ❑ V^{k}_{n} – validators set (committee) of k-th connected distributed system active in the epoch e_n.

Main State Variables of The Spectrum protocol participant:

- ❑ P – protocol participant (party).
- ❑ PK_{P} – public key of the party P.
- ❑ PK_{P}^{k} – public key of the k-th connected external system of the party P.
- ❑ s_{P} – stake value of the party P.
- ❑ $S^{\mathrm{sync},n}$ – stakeholders distribution of all K committees members for the epoch n (used for the synchronization lottery).
- ❑ $S_{k}^{\mathrm{cons},n}$ – stakeholders distribution of the V_{n+2}^{k} members calculated for the epoch n (used for the leader lottery).
- ❑ $S_{k}^{\mathrm{ver},n}$ – stakeholders distribution of verified and equipped with functionalities $\mathcal{F}_{\mathrm{ConnSys}}^{k}$ participants fot the epoch e_n (used for the consensus lottery to select V_{n+4}^{k}).
- ❑ η_n – random seed of the epoch e_n (epoch randomness).
- ❑ $v_{\mathrm{P}}^{\mathrm{vrf}}$ – VRF public key of the party P.
- ❑ $v_{\mathrm{P}}^{\mathrm{kes}}$ – KES scheme public of the party P.
- ❑ $\pi_{\mathrm{P},j}^{\mathrm{sl}}$ – VRF slot proof produced by the party P for slot sl_j (used in the leader lottery and in the synchronization lottery).
- ❑ $\pi_{\mathrm{P},n}^{\mathrm{e}}$ – VRF epoch proof produced by the party P for epoch e_n (used in the consensus group lottery).
- ❑ $b_{P,j}^{\mathrm{sync}}$ – synchronization beacon produced by party P for slot sl_j.
- ❑ $T_{\mathrm{P},n}^{\mathrm{cons},k}$ – consensus group lottery threshold (for related V_n^k committee) calculated for the party P for epoch e_n.
- ❑ $T_{\mathrm{P},n}^{\mathrm{lead},k}$ – leader lottery threshold (for related V_n^k committee) calculated for the party P for epoch e_n.
- ❑ $T_{\mathrm{P},n}^{\mathrm{sync}}$ – synchronization beacon lottery threshold calculated for the party P for epoch e_n.
- ❑ $\mathbf{S}_{P}^{\mathrm{set}}$ – set of actual connected to participant P distributed systems' ids.
- ❑ $\mathbf{T}^{\mathrm{cons}}$ – set of actual consensus group lottery thresholds for different committees.
- ❑ $\mathbf{T}^{\mathrm{lead}}$ – set of actual leader lottery thresholds for different committees.
- ❑ $\mathbf{T}^{\mathrm{sync}}$ – set of actual synchronization lottery thresholds for different committees.
- ❑ $\mathcal{C}_{\mathrm{loc}}$ – the local chain the party adopts based on which it does evaluation and exports the ledger state.
- ❑ isSync – the party's stores synchronization status.
- ❑ buffer – the buffer of transactions.
- ❑ syncBuffer – the buffer of the synchronization beacons.
- ❑ futureChains – a buffer to store chains that are not yet processed.

- fetchCompleted – a variable to store whether the round messages have been fetched.
- localTime – the party's current local slot.
- lastTick – the last tick received from $\mathcal{G}_{\text{PerfLClock}}$.
- EpochUpdate($\cdot$) – a function table to remember which clock adjustments have been done already.