Filecoin Hierarchical Consensus Specification (aka Project B1)

ConsensusLab
Protocol Labs

July 15, 2022

1 Introduction

Consensus, or establishing total order across transactions, poses a major scalability bottleneck in blockchain networks [**?**]. In short, the main challenge with consensus is that it requires all *nodes* (often called *validators*, or *miners*) to process all transactions. Regardless of the specific consensus protocol implementation used, this makes blockchain performance limited to that of a single miner at best.

Borrowing ideas from traditional distributed databases, one possible approach to overcoming this limitation is to resort to the partitioning, or *sharding*, of state processing and transaction ordering. In a sharded system, the blockchain stack is divided into different groups called *shards*. Each shard is operated by its own set of miners, keeps a subset of the state, and is responsible for processing a part of the transactions sent to the system. The rationale behind sharding is that by dividing the whole blockchain network into different groups, the load of the system is balanced among them, increasing the overall transaction throughput of the system. Instead of every node having to process all transaction sent to the system, each shard processes and handles the state of a subset of transactions and objects.

Existing sharded designs [**?**, **?**, **?**] often follow a similar approach to the one traditionally used in distributed databases, where the system is treated monotonically, and a sharded system acts as a distributed controller which assigns miners to different shards, and attempts to distribute the state evenly across shards to balance the load of the system. Many of these designs use a static hash-based assignment to deterministically select what state needs to be kept and what transactions are to be processed by each shard.

The main challenge with applying traditional sharding to the Byzantine fault-tolerant context of the blockchain lies in the security/performance tradeoff. As miners are assigned to shards, there is a danger of dilution of security compared to the original single-chain (single-shard) solution. For instance, in both Proof-of-Work and Proof-of-Stake blockchains sharding may lead to the ability of the attacker to compromise a single shard with only fraction of the mining power, potentially leading to compromising the system as a whole. Such attacks are often referred to as *1% attacks* [**?**, **?**, **?**]. To circumvent such attacks, sharding systems need to re-assign randomly and periodically miners to shards in an unpredictable way to cope with a semi-dynamic adversary [**?**]. In the following, we refer to this approach as to *traditional sharding*.

We believe that the traditional sharding approach to scaling, that considers the system as a monolith, is not suitable for decentralized blockchains. Instead, in this project we depart from the traditional sharding approach to build *hierarchical consensus*. In *hierarchical consensus* instead of algorithmically assigning node membership and evenly distributing the state, we follow a "sharding-like" approach where users and miners are grouped into *subnets* and where they *can freely choose the subnets they want to belong to*. What is more, users can spawn new child subnets from the one they are operating in according to their needs, and become a miner there even if they are not currently one provided they fulfill all the miner requirements set by the protocol.

We refer to the state of a subnet as the *state tree or chain* holding all the data for the subnet. Each subnet keeps its state in an independent state tree (or chain) and processes transactions that involve objects

that are stored in their state tree. Every subnet can run its own independent consensus algorithm and have its own security requirements and performance guarantees.

All subnets in the system are organized hierarchically where each of them will have one parent subnet and can have any number of child subnets, except root subnets which have no parent and are hence called *root networks*, or *rootnets* (which are the initial anchor of trust of the protocol). As a major difference compared to traditional sharding, subnets in hierarchical consensus are firewalled [**?**] in the sense that a security violation in a given subnet is limited in effect to that particular subnet and its child subnets, with a limited impact to its ancestor subnets. The economic impact of an attack on a subnet's ancestors is at most of the total circulating supply (or locked value) in the subnet. Moreover, ancestor subnets *help secure* their descendant subnets — for instance, checkpointing a Proof-of-Stake subnet into its parent may help alleviate long-range and similar attacks. In addition, rootnets in *hierarchical consensus* are also able to commit in parallel into other blockchains/rootnets with better or complementary security guarantees. For instance, the rootnet in Filecoin hierarchical consensus can leverage the high security of the Bitcoin network by periodically committing a checkpoint of its state (see ConsensusLab project B2, [**?**]).

At a high level, a *hierarchical consensus* allows for incremental, on-demand, scaling and simplifies deployment of new use cases on a blockchain network. Our design is inspired by the Proof-of-Stake sidechain design, to our knowledge first proposed in [**?**]. In our case, hierarchical consensus generalizes the approach of [**?**], minding the specifics of Filecoin, which does not use Proof-of-Stake as a sybil attack protection on the root chain, but rather copes with sybils in a way specifically tailored to data storage (Proof-of-SpaceTime (PoST) and Proofs-of-Replication (PoRep) [**?**, **?**]). Also, unlike [**?**]'s sidechains, hierarchical consensus subnets can run any type of consensus, and are not limited to running PoS-based consensus algorithms.

TODO: In the following, we first give, in Section 2, a high level overview of the system, including its specification. We elaborate on incentivization and tokenomics policies of hierarchical consensus, including the treatment specific to Filecoin storage power tables [**?**] in Section 4. Section **??** discusses implementation details. We give preliminary performance evaluation of hierarchical consensus in Section **??**. Section **??** overviews related work and Section **??** concludes.

2 System Overview

Figure 1 depicts a high-level overview of a *hierarchical consensus* system. The illustration starts with a root network with its own chain keeping the state and processing the transaction of the whole system, like Filecoin. At some point, a subset of users in the root chain chooses to launch a new use case that requires faster validation times (i.e., lower latency) or higher throughput. To accommodate the performance requirements they expect in their use case, they choose to spawn their own subnet, $Subnet_{11}$. This subnet spawns a *new chain* with its own state and a subset of the participants of the root chain. From this point on, $Subnet_{11}$ processes transactions involving the objects in the subnet, and keeps its own state independently from the root chain.

Subnets are able to interact with the state of other subnets and that of the rootnet through *cross-net* messages, and the state consistency between different subnets is achieved by committing checkpoints to ancestor chains (i.e., chains in the upper level of the hierarchy, see Sec. 2.7).

We follow the approach in which miners and full-nodes in a given subnet s has trusted access to state of its direct ancestor subnet. We implement this by having nodes on subnet s itself sync with the chain of its ancestor (i.e., obtain the full state thereof). Converse is not true, miners on parent subnets do not need to sync the chain or be miners on child subnets, see Figure 2 for an illustration.

In order for miners to spawn a new subnet, they need to deposit an initial collateral from one of their accounts in one of the higher chains in the hierarchy from which the subnet wants to be spawned. These deposits are used to insure participants of a child subnet in the case of an attack. In *hierarchical consensus* it may be impossible to enforce an honest majority of mining power in every subnet, which can result in the subnet chain being compromised or attacked. Child chains in *hierarchical consensus* are implemented following a *firewall* security property analogous to the one introduced in [**?**], ensuring that the impact of a child chain being compromised is limited from the perspective of the parent chain in at most the circulating

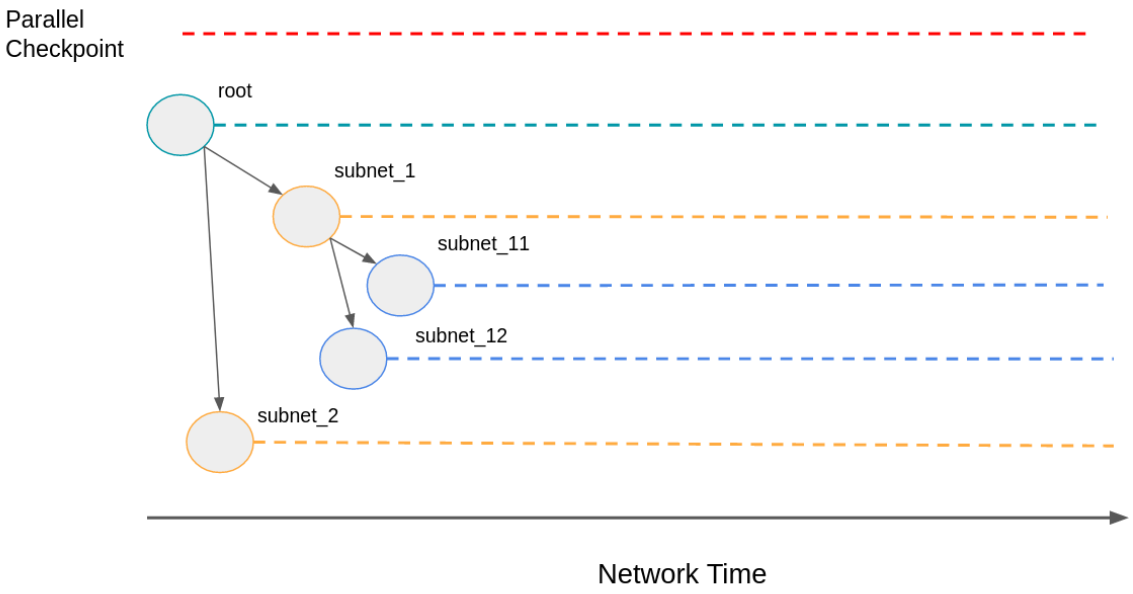

Figure 1: **System Overview.** Subnets are spawned from the root chain building a hierarchy of independent networks.

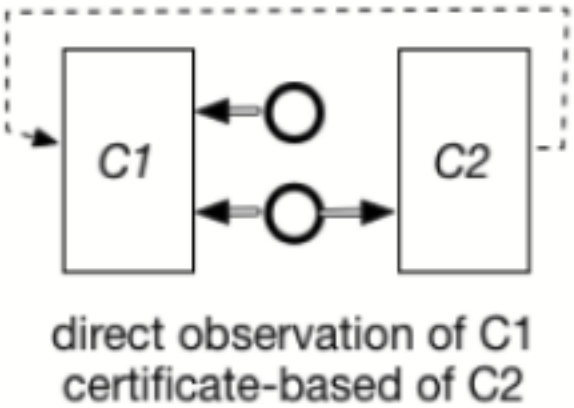

Figure 2: **Cross-net messages propagation approach.** C1 is the parent of C2. C2 is informed of state changes in its parent and cross-net messages directed to her through direct observation, as every child stays in-sync with its parent. Cross-net messages propagation from childs to parent, on the other hand, is certificate-based. Parent learn of new cross-net messages directed to them through the certificates (in our case checkpoints) committed from them.

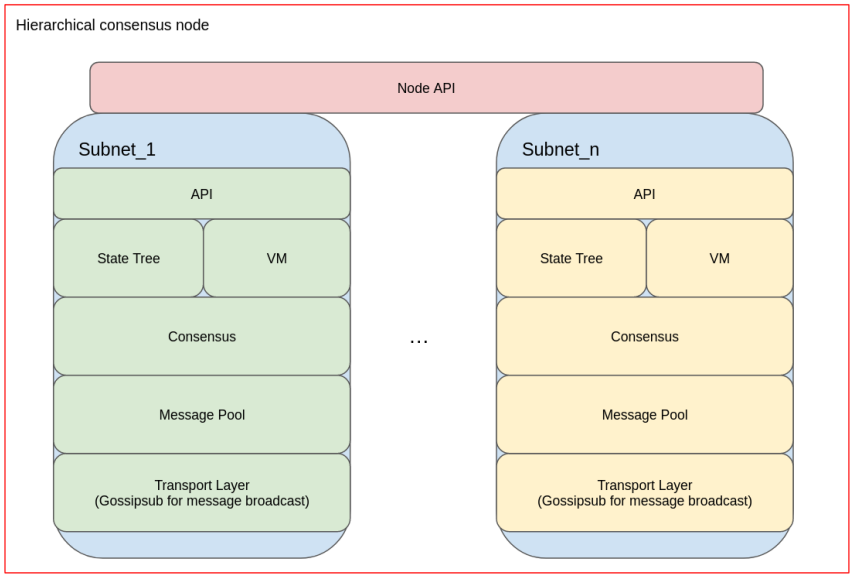

Figure 3: Hierarchical Consensus node and subnet stack

supply in the child chain. The circulating supply is determined by the balance between cross-net transactions entering the subnet and cross-net transactions leaving the subnet. The circulating supply can never be negative, and funding miners can't have access to their deposit in the top chain, so addresses in the child chain are funded either through cross-net transactions from other chains, or mining rewards earned in the child subnet. This mechanism is described in detail in Section 2.1.

In the following subsections we present in detail the operation of *hierarchical consensus*, the lifecycle of a subnet, the different protocols involved, and the incentives system.

2.1 Subnet Actor (SA)

In order to instantiate a new subnet in the system, users need to deploy a new actor in the parent chain implementing the *Subnet Actor (SA) interface*. The SA interface determines the core functions and basic rules required for an actor to implement the logic for a new subnet. This approach gives users total flexibility to configure the consensus, security assumption, checkpointing strategy, policies, etc. of their new subnet so it fulfills all the needs of their use case.

The Subnet Actor is the public contract accessible by users in the system to determine the kind of child subnet being spawned and controlled by the actor. From the moment the SA for a new subnet is spawned in the parent chain, users looking to participate from the subnet can instantiate their new chain and even start mining from it. However, they won't be able to send funds to this new chain, or receive mining rewards in FIL[1] (i.e., the native token of the entire system).

A new subnet instantiates a new independent chain with all its subnet-specific requirements to operate independently. This includes, in particular: a new pubsub topic that peers use as the transport layer to exchange chain-specific messages, a new mempool instance, a new instance of the VM, as well as any other additional module required by the consensus that the subnet is running (like system actors, mining power resources, etc.). Figure 3 depicts how every new subnet creates a new independent instance of each of these modules.

In order for the new subnet to be able to interact with the rest of the subnets of the hierarchy, it needs

[1]Users can still mint their own token for their subnet and use it to reward miners, but this would be equivalent to spawning an independent network

to register and deposit a minimum collateral in the Subnet Coordinator Actor (SCA) of its parent chain. This protocol is described in detail in section 2.3. We use the term *founder* to refer to the initial set of users responsible for defining the subnet specifics and requirements, deploy the Subnet Actor for the subnet, and providing the initial collateral to register the subnet to the

The interface that needs to be implemented by a Subnet Actor is the following:

- *Constructor*: Contains the code triggered when the actor is deployed. It receives as input the configuration for the subnet (implementation of the consensus algorithm to use, signature policy to accept checkpoints as valid, minimum collateral required for participants to be entitled to mine, and whatever other policy the founder of the chain wants to specify in the subnet contract). The constructor is responsible for initializing the actor state (Listing 2.1). The list of arguments of this function may differ according to the specific implementation of the Subnet Actor.

- *Join*: Subnets may implement different policies and requirements to accept new members. For instance, the minimum collateral required in order to mine or even participate in the subnet. A subnet may also require a minimum number of participants to ensure the security of its consensus protocol. Additionally, more complex policies may be enforced by SA, such as the requirement to have a minimum delay with the rest of the validators participating in the subnet. This kind of requirements ensures that new participants joining a subnet (which may be optimized for a specific use case) doesn't harm its performance and optimal operation. Users looking to join a subnet send a message to the join function of the SA which validates all of these requirements for the new member to be accepted.

  When all the spawning requirements determined for the subnet in the actor are met, and the subnet is ready to "go public" and interact with the rest of the hierarchy, the actor needs to send a message to the SCA to register the actor. As described below, in order for the SCA to accept the registration of a new subnet, the register message transaction needs to include an amount of tokens greater than *minCollateral* as a deposit to cover potential attacks over the subnet (and to ensure the firewall requirement for subnets in the hierarchical consensus). Once a SA has been registered in the SCA, it can send and receive messages to other subnets, and start checkpointing to its parent chain.

  Once the chain has been registered, the subnet may still accept new users to join, and it can update its stake in the SCA sending a message to *AddCollateral* including the amount of funds to be added to the subnet's collateral.

- *SubmitCheckpoint*: This function verifies the validity of the committed checkpoints to be propagated to the SCA, and from there to the top of the hierarchy. Every time a new checkpoint is committed by miners of the subnet, this function is triggered and it verifies that the checkpoint follows the right format and that is consistent with previous checkpoints. Additionally, and depending on the implementation of the SA, it may perform additional verification like checking a minimum number of signers, verify threshold signatures according to the number of miners, fraud detection, etc. When these verifications are successful, this function needs to send a message to the *Checkpoint* function of the SCA to propagate the checkpoint to the top of the chain and unlock any pending mining rewards for the chain. As detailed in Section , checkpoints also include information for the commitment of cross-net transactions.

- *Leave*: This function can be used by miners and users looking to leave the subnet and recover their collateral. As with every other function in the SA interface, subnet founders are free to implement their own leaving policies (time required to leave, fee for leaving, collateral locking time, etc.). This function needs to trigger the *ReleaseCollateral* function in the SCA to recover the corresponding part of the collateral in the coordinator. Some additional checks may be required in the actor to ensure that the minimum collateral to keep interacting with the rest of the hierarchy is still kept for the subnet in the SCA.

- *Kill* is used to remove the subnet and the chain from the hierarchy. This function needs to send a message to the *Kill* function in the SCA of the subnet's parent to release the collateral of the subnet

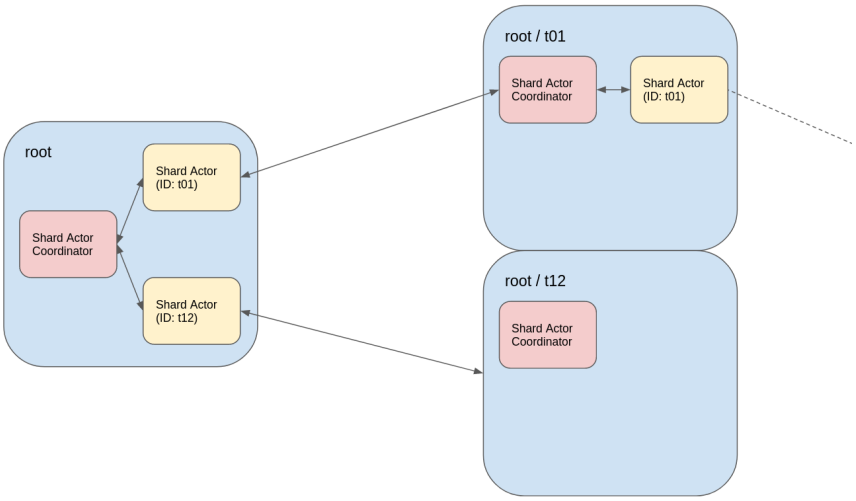

Figure 4: Subnet Actor and Subnet Coordinator Actor architecture

and return the funds to their legitimate owners. Again, the specific policies and verification for a subnet to be killed are configured in the implementation of the SA by the subnet founders. *Kill* can be called by any participant in the subnet, and its up to the specific implementation of the subnet to determine its behavior, and when to trigger the final *Kill* signal to SCA to unregister from the hierarchy.

In the first implementations of the protocol we provide a set of templates for different implementations of SA with specific consensus algorithms and checkpointing policies, but users are free to implement and deploy their own SA implementations with custom policies and custom consensus algorithms. A representation of the actor architecture for hierarchical consensus is depicted in Figure 4.

2.2 Subnet Coordinator Actor (SCA)

The main entity responsible for handling all the lifecycle of child subnets in a specific chain is the *Subnet Coordinator Actor* (SCA). The SCA is a system actor (in Filecoin lingua), i.e., a custom, *built-in smart-contract*, that exposes the interface for subnets to interact with the hierarchical consensus protocol. This smart-contract includes all the available functionalities related to subnets and their management. It also enforces all the security assumptions, fund management, and cryptoeconomics of the hierarchical consensus, as Subnet Actors are user-defined and can't be trusted. The SCA exposes the following functions:

- *Constructor*: It deploys the actor in a new subnet; this is done whenever a new subnet is instantiated in the system. There is only one Subnet Coordinator Actor actor per subnet (i.e., singleton system actor). Every time a new subnet is spawned, a brand new instance of this actor is deployed for the subnet. It initializes the SCA state (Listing 2.
- *Register*: It registers a new subnet in the hierarchy. This function needs to be triggered by the actor that keeps the contract for the newly registered subnet. The amount of tokens included in the transaction calling the *Register* function in the actor is added to the collateral of the subnet, along with a pointer to the Subnet Actor ID. These tokens are staked as a collateral to cover penalties for miner misbehaviour. Initially, there is not a minimum number of miners required to spawn a subnet as long as the minimum collateral is fulfilled. This stake will have a key role for the security model in subnets.
- *CommitChildCheckpoint*: Commits a new checkpoint from a child subnet in the network. This function is also triggered by the SA handling a specific child subnet. The child SA is responsible for making

Listing 1 Subnet Actor State. Data stored in the state of Subnet Actor actor

```
type SubnetState struct {
        // Human-readable name of the subnet.
        Name string
        // ID of the parent subnet
        ParentID hierarchical.SubnetID
        // Type of Consensus algorithm.
        Consensus hierarchical.ConsensusType
        // Minimum collateral required for an address to join the subnet
        // as a miner
        MinMinerCollateral abi.TokenAmount
        // List of miners in the subnet.
        Miners []address.Address
        // Total collateral currently deposited in the
        TotalCollateral abi.TokenAmount
        // BalanceTable with the distribution of collateral by address
        Collateral cid.Cid // HAMT[tokenAmount]address
        // State of the subnet (Active, Inactive, Terminating)
        Status Status
        // Genesis bootstrap for the subnet. This is created
        // when the subnet is generated.
        Genesis     []byte
        // Checkpointing period.
        CheckPeriod abi.ChainEpoch
        // Checkpoints submit to SubnetActor per epoch
        Checkpoints cid.Cid // HAMT[epoch]Checkpoint
        // WindowChecks
        WindowChecks cid.Cid // HAMT[cid]CheckVotes
}
```

the basic signature and policy verification of the checkpoint. It then triggers this function in SCA to propagate the checkpoint to the top of the hierarchy. When receiving a new checkpoint from a child chain, this function performs an additional format and validity check, and it aggregates the content with the checkpoints of its other child chains and periodically propagates it to the top of the hierarchy. The checkpointing period for a subnet can be determined in the moment of spawning it, and is defined as the number of epochs in the subnet between the commitment of two subsequent checkpoints.

- *RawCheckpoint*: It constructs and returns the checkpoint template that needs to be signed and populated by miners and committed to the corresponding SA before the next checkpoint window. This function aggregates all the information and gives a constructed version of the checkpoint to all miners. See 2.5 for more details about the protocol.

- *CrossMessage*: Users in a subnet looking to send a message to an address outside their chain need to send the transaction to this function. The function takes care of including every cross-net message in the next checkpoint for its propagation to its corresponding destination. See section 2.7 for details about this protocol.

- *AddCollateral*: Triggered by a SA to update the collateral of a subnet by the amount included in the message.

- *ReleaseCollateral*: *AddCollateral*'s counterpart, it releases an amount of collateral to a specific address from the subnet's collateral.
- *SubmitFraud*: Submits a fraud proof for a misbehaving miner from a subnet. A valid fraud proof for miner slashes its collateral and distributes the slashed collateral proportionately to all the users impacted by the attack. The fraud protocol is detailed in section **??**.
- *Fund*: Use to send top-down transactions to inject new funds in a specific address of the child subnet. See section 2.7.
- *Release*: Release funds from a child subnet. This is triggered when the corresponding tokens for the address in the child subnet are burned. See section 2.7.
- *Kill*: This function removes a subnet from the parent SCA registry and returns all the collateral and outstanding balances to the corresponding addresses in the current chain. The SA for a child chain needs to trigger this function if all the killing verifications have passed and it wants to effectively remove the subnet from the hierarchical consensus. See section 2.6.
- *ApplyMessage*: This function is called by every peer after a block including cross-net messages is validated by a subnet consesus algorithm. This functions executes all cross-net messages in a block, and triggers the corresponding state changes in the subnet. See section 2.7 for a detailed description of how this function works.
- *Save*: Saves a snapshot of the current state of the subnet in a Filecoin sector, or any other native decentralized storage supported by the implementation. This enables the persistence of subnet state even after they are kill. See section 2.6 for more details on this scheme.

2.3 Spawning and joining a subnet

To spawn a new subnet, peers need to deploy a new SA implementing the core logic for the new subnet. The contract specifies the consensus protocol to be run by the subnet, and the set of policies to be enforced for new members, leaving members, submitted checkpoints, to killing the subnet, etc. For new subnets to be able to interact with the rest of the chains in the hierarchy, receive funds and messages from other subnet, and have access to mining rewards in the native tokens, they need to be registered in the SCA of the parent chain. To be registered in the SCA, the SA needs to send a new message to the *Register* function of the SCA. This transactions needs to include the amount of tokens the subnet wants to add as collateral in the parent chain to secure the child chain. In order to perform this transaction, the SA needs to have enough funds available. This funds are are provided by joining and staking the subnet through the SA according to its policy to fund the register transaction. For a subnet to be registered, at least $minCollateral_{subnet}$ needs to be staked in the SCA.

Subnets can run any consensus algorithm to validate blocks of the chain. Subnets can also determine the consensus proofs they want to include for light clients. Subnets periodically commit a proof of their state in their parent. These proofs are propagated to the top of the hierarchy, making them accessible to any member of the system. This proof should include enough information for any client receiving this information to be able to verify the correctness of the subnet consensus. Subnets are free to choose a proof scheme that suits their consensus best. With this, users are able to determine the level of trust over a subnet according to the security level of the consensus run by the subnet, and the proofs provided to light clients. As described in section 2.5, for the initial implementation this proof consists of information about the latest block in the chain sealed with the signature of a majority of the subnet[2].

As part of the framework, some subnets may allow miners that are not direct participants of child chains to dedicate part of their mining resources from the parent chain to increase the security in a child chains. In

[2]This proof is quite weak, as it requires for light clients to have access to the state of the subnet to verify it, which can be tricky. More complex and secure proofs will be included in next iterations of the protocol

Listing 2 Subnet Coordinator Actor State. Data stored in the state of Subnet Coordinator Actor actor

```
type SCAState struct {
        // ID of the current network
        NetworkName hierarchical.SubnetID
        // Total subnets below this one.
        TotalSubnets uint64
        // Minimum collateral to create a new subnet
        minCollateral abi.TokenAmount
        // List of subnets
        Subnets cid.Cid // HAMT[cid.Cid]Subnet

        // Checkpoint period in number of epochs
        CheckPeriod abi.ChainEpoch
        // Checkpoints committed in SCA
        Checkpoints cid.Cid // HAMT[epoch]Checkpoint

        // CheckMsgMetaRegistry
        // Stores information about the list of messages and child msgMetas being
        // propagated in checkpoints to the top of the hierarchy.
        CheckMsgsMetaRegistry cid.Cid // HAMT[cid]CrossMsgMeta
        Nonce                 uint64  // Latest nonce of cross message sent from subnet.
        BottomUpNonce         uint64  // BottomUpNonce of messages for msgMeta received from checkpoints
        BottomUpMsgsMeta      cid.Cid // AMT[schema.CrossMsgMeta] from child subnets to apply.

        // AppliedNonces
        //
        // Keep track of the next nonce of the message to be applied.
        AppliedBottomUpNonce uint64
        AppliedTopDownNonce  uint64
}
```

papers like [**?**] , this is referred as a *merging consensus*. In the end, the goal is to introduce external members to the validation committee if the child chain to increase its security (and potentially honest majority). Even more, a subnet may choose to implement a random membership assignments where validators in the top chain that volunteer to mine in child chains are assigned randomly to child subnet with the same mining resource as the one they currently run for an additional reward. This is out of the scope in our first implementation of the system, but all the specific policies will be configurable in the SA.

As it will be described in detail in section 2.5, subnets use a checkpointing protocol to anchor their security to that of their parent chain and of all of the chains in the hierarchy in their path to the root chain. These checkpoints also propagate cross-net message information (through cross-net message metadata) to the top of the chain. Child chains belonging to other branches of the hierarchy can pick up this information when the checkpoint reaches a common parent. Section 2.7 presents this protocol in detail.

Every subnet keeps its own state, processes transactions that affect the state they keep, and uses an independent transport layer to interact with other peers belonging (or interested) in the subnet. Subnets are identified with a unique *ID* that is inferred deterministically from the ID of the SA that represents the contract of the new subnet in the parent network, and the ID of the parent from where they are spawned. These IDs are assigned by the SCA when new subnets are registered. When SA sends its *register* message to SCA, SCA checks the source actor of the message, and generates the ID accordingly. This deterministic naming enables the discovery and interaction with subnets from any other point in the *hierarchical consensus*

without the need of a discovery service[3]. This will be key for every cross-net protocol as detailed in section 2.4.

Once a subnet has been spawned, any new miner will be able to join the subnet and start mining in it by ascribing to the joining policy implemented in the SA. Users looking to join subnet need to send a message to the *Join* function of the specific SA of the subnet they want to join.

A common policy that may be worth implementing by child subnets is that where miner's collateral is locked for a certain number of epochs, although this is not required by the core protocol. If a miner tries to leave the subnet before the locking time has elapsed, it will be penalized proportionately to the time left for the locking time to finish. This is used to disincentivize big miners from abusing child chains and performing flash or goldfinger attacks to them **TODO: references?**. The granularity of the locking period for staking can be specified in the checkpointing period of a subnets parent chain. Details of the incentive system are described in section 4.

Every subnet keeps a pubsub topic in the network for subnet-specific communication. Messages to a specific subnet are sent to the subnet's pubsub topic and picked up by peers belonging to the subnet. Optionally, other peers may choose to listen to messages from other subnets, although they won't be entitled to mine in the subnet until they join the subnet as miners. Subnets and their underlying transport layer can be directly discovered using the deterministic naming convention used in the system (see Section 2.4). Thus, to send a message to a subnet with ID_1, a peer just needs to publish the message to the pubsub topic of the subnet which is uniquely identified after the subnet's ID_1. With this, no special discovery protocols are needed for the exchange of messages between subnets, as all available subnets are directly discoverable through their pubsub topic and can be picked up by peers that belong to the subnet.

2.4 Naming

Every subnet is identified with a unique ID. This ID is assigned deterministically by the SCA when a new subnet is registered according its location in the hierarchy. The rootnet in the hierarchical consensus always has the same ID, *root*. From there on, every subnet spawned from the root chain is identified through the ID of their SA. Thus, if a new subnet is being registered from an actor with ID t_{ij}, it is assigned $root/t_{ij}/$ as its ID. Actor IDs are unique through the lifetime of a network. Generating subnet IDs using the SA ID ensures that they are unique throughout the whole history of the system and of any of its underlying chains.

When moving deeper into the consensus hierarchy, SCAs in child chains use the same protocol to assign IDs to the new child chains they spawn. They add to the parent chain a suffix to the path including the actor ID of the corresponding SA. Consequently, a subnet represented by SA t_{mn} spawned in $root/i/$ is identified as $root/i/t_{mn}/$.

This naming convention allows to deterministically discover and interact with any subnet in the system. It also offers an implicit map of the hierarchy. Peers looking to interact with a subnet only need to know their ID, and publish a message to the pubsub topic with the same name. Other peers participating in that subnet will be subscribed to that topic and are able to pick up the message.

Peers can also poll the available child chains of a specific subnet sending a query to its SA. This allows any peer to traverse the full hierarchy and update their view of available subnets.

In the future, *hierarchical consensus* may implement an additional DNS-like actor in the system that allows the discovery of subnets using human-readable names, thus making the transaction between a domain name and the underlying ID of a subnet.

Additionally, user and actor addresses are allowed to be shared between subnets, holding different balances and state in each network. Even more, a pair of keys from a user control the same address in every subnet in the system. To disambiguate addresses between different subnets, we implement a new kind of address that adds subnet-related information to raw addresses. We call this kind of address the *f04* address. *f04* addresses can be easily built from a raw address by prefixing the ID of a subnet. The payload of *f04* addresses have the following form:

$$f04 < subnet_{ID} >::< Address_{raw} >$$

[3]In the future we may include DNS-like name resolution protocols to support human readable names to discover subnets

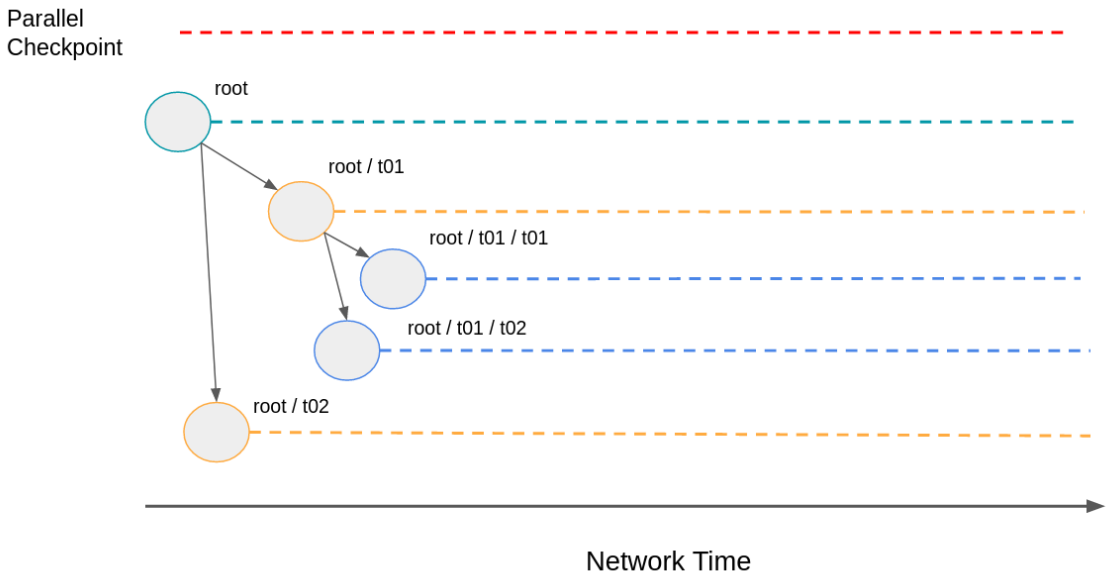

Figure 5: Subnet Naming Convention

For instance:

$$f04(/root/t0100 :: rawAddr)$$

f04 addresses are mainly used in the context of cross-net messages in order to specify the subnet for the source or destination address in a message. Thus, any subnet processing the message in any point of the hierarchy is able to uniquely identify the source and the destination to route the message.

2.5 Checkpoints and mining

Our *Hierarchical consensus* uses a checkpointing protocol to anchor child subnet's security to the one of its parent chains and the ones on the top level of the hierarchy. Each subnet may run its own consensus algorithm to validate transactions, and the period of transaction validation may be different in every subnet. Thus, child chains need to periodically checkpoint their state into the top chain to leverage the parent chain's security. Child subnet chains are allowed to determine their checkpointing periodic. The checkpointing period and the specifics of checkpointing signature and validation is handled by the SA of a child subnet. The only thing enforced by the SCA is that the checkpoint has the right format, that it has been committed by the right SA in the right checkpointing window, and that is consistent with previous checkpoints committed by the subnet. SA s for registered child subnets are the only ones entitled to commit checkpoints in the SCA.

Checkpoints are also used to propagate information from a child chain to the upper levels in the hierarchy. Once these checkpoints reach a common parent, other subnets in the hierarchy with non-overlapping ancestors are able to pick-up the information targeting their subnet inspecting the checkpoints. Section 2.7 details this mechanism.

Checkpoints for a subnet can be verified at any point using the state of the subnet chain. If at any point a user in the system detects a misbehavior from a miner in a subnet, it can generate a *fraud proof* using the state of the chain and the checkpoint in upper chains from the hierarchy to penalize the misbehaving miner. A misbehavior is penalized with a slash of their stake in the chain. If a miner's collateral gets below $minCollateral_{peer}$ after being slashed, it will lose mining rights in the subnet. If the fraud proof includes enough information, the slashed collateral of the miner is proportionately distributed in the parent chain

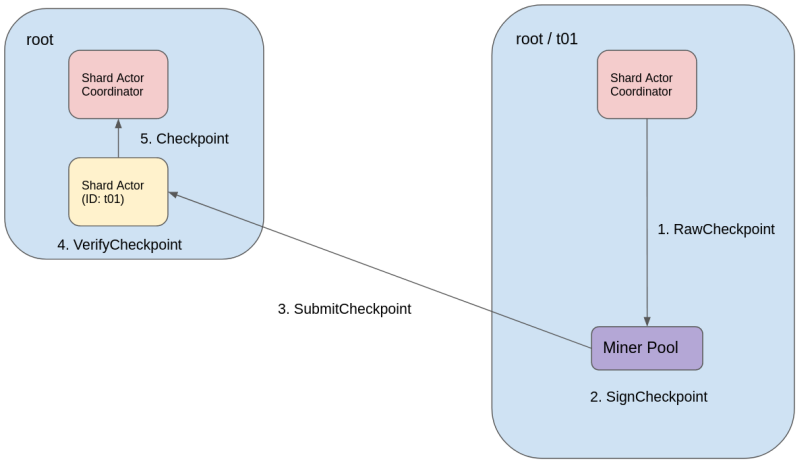

Figure 6: **Checkpointing protocol.** Miners in the subnet get the *RawCheckpoint* template for the next checkpointing window, populate it with the required information and sign it. Miners then submit their signed checkpoint to SA in the parent chain. When the majority of valid signed checkpoints enforced by the SA is fulfilled, it submits the checkpoint to the parent's SCA for its propagation to the rest of the hierarchy.

to the addresses of the parties impacted in the attack. Further details on slashing and fraud proofs can be found in section **??**.

The *hierarchical consensus* does not influence in any way the native token cryptoeconomics of the root chain in the hierarchy. The baseline token issuance and mining rewards of the root chain are conveniently distributed between all the subnets of the hierarchy. Miners in subnets are rewarded with the transaction fees of all the transactions being validated in their subnets. Optionally, founding members of a subnet may choose to create a new token specifically for the subnet and give additional mining rewards using this subnet-specific token, but this is out of scope in the initial stages of the implementation.

2.5.1 Checkpointing protocol

Checkpoints need to be signed by miners of a child chain and committed to the parent chain through their corresponding SA. The specific signature policy is defined in the SA and determines the type and minimum number of signatures required for a checkpoint to be accepted and validated by the SA for its propagation to the top chain. Any signature scheme may be used here: a more naive and static (multi-sig) approach where all miners sign the checkpoint using their private key and commit it to the SA, and the SA checks that the checkpoint is the same for all miners up to a threshold to propagated; or a more advanced protocol where a threshold signature from all miners in the subnet is used to sign the checkpoint and commit it to the subnet. In the reference implementation of the protocol, SA waits for a minimum number of miners in the subnet to send a signed checkpoint before it commits it in SCA (Figure 6.

Miners can access the template of the checkpoint that needs to be signed and populated in the current signing window by calling the *RawCheckpoint* function of the SCA in $/root/t01/t30$. Once signed, checkpoints from $/root/t01/t30$ are committed in the SA $t30$ of the subnet chain $/root/t01$ by sending a message to the *SubmitCheckpoint* function of that actor. After performing the corresponding checks this actor triggers a message function to the *CommitChildCheckpoint* function of the SCA in $/root/t01$ that is responsible for aggregating the checkpoint with $/root/t01/t30$ from the ones from the rest of its other childs, and to generate a new checkpoint and propagate to its parent chain, $/root$. As checkpoints flow up the chain, the SCA of each chain picks up this checkpoints and inspect it to propagate potential state changes (like update of balances in a monetary transaction) triggered by messages included in the cross-net messages field and with its subnet as a destination. See Figure 7 An extended description of this protocol can be found in section 2.7.

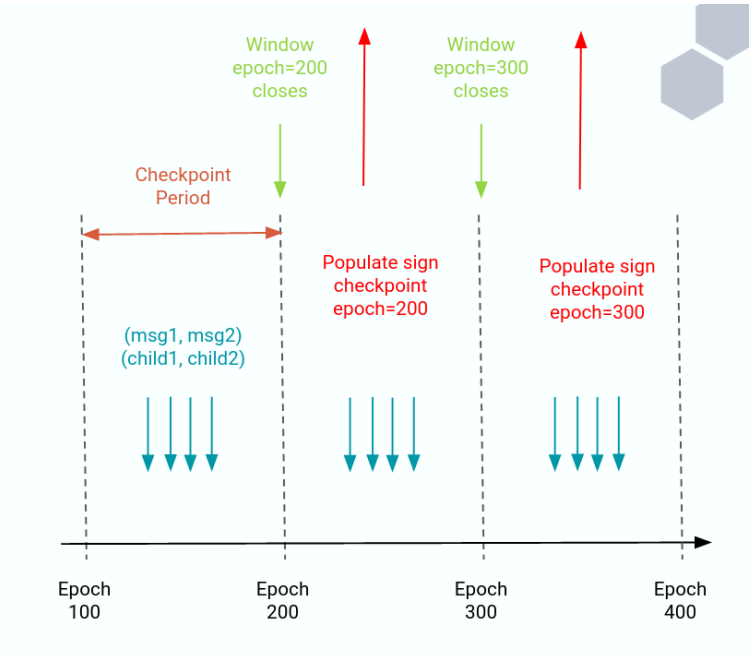

Figure 7: **Checkpoint template population.** The ceckpoint period determined by SA determines the checkpoint window for which new cross-net messages are accepted in the current checkpoint. When the end of the period is reached, new cross-net messages are populated in the next checkpoint, and a signature window is opened for the checkpoint of the previous epoch.

Listing 3 shows the information included in a checkpoint. Checkpoints are always identified though their CID, and include the corresponding signature from miners in the subnet chain (this can be the signature of an individual miner, an array of signatures, or a threshold signature, depending on the SA policy). Inside the checkpoint we can find (Listing 3:

- The source subnet of the checkpoint
- The content identifier CID of the latest block from the subnet chain, *latestBlockCid*, being committed in the checkpoint, and its *height*.
- A pointer to the CID of the previous checkpoint generated by the subnet.
- The meta-tree: a tree of cross message metadata including every cross-net message being propagated upwards by the subnet and its corresponding child subnets. We call this cross message metadata *CrossMsgMeta*. The *CrossMsgMeta* for a cross message includes information about the source subnet of the message, the destination subnet, the cross-net message nonce, and the cid (message digest) of all the messages included. Any subnet looking to know the specific message for that CID of the messages being propagated – which will be the case for the destination subnet of the messages, see Section 2.7– only needs to send a query message for the cross-net message CID to the pubsub topic of the source chain. The list of *CrossMsgMeta* propagated from a child chain to its parent is aggregated as the checkpoint moves up the hierarchy. Thus, every subnet only sees the message aggregation (i.e., the digest of all the subnets childrens CrossMsgMeta list) of its child chains. An illustration of how this works for cross-net message exchanges is depicted in figure 12.
- A tree including the CID of every checkpoint of the child chains. In this case, this data structure is a single tree that includes the subnet ID and the corresponding checkpoint CID for every child chain.

As it was the case for the cross-net message list of trees, this data is updated by every new checkpoint in its way up the hierarchy.

Listing 3 Checkpoint data structure

```
// CrossMsgMeta includes information about the messages being propagated
// from and to a subnet.
//
// MsgsCid is the cid of the list of cids of the mesasges propagated
// for a specific subnet in that checkpoint
type CrossMsgMeta struct {
        From    string // Source of the messages being propagated
        To      string // Determines the destination of the messages included in MsgsCid
        MsgsCid cid.Cid
        Nonce   int    // Nonce of the msgMeta
}

// CheckData is the data included in a Checkpoint.
type CheckData struct {
        Source string
        TipSet []byte
        Epoch  abi.ChainEpoch
        PrevCheckCid cid.Cid
        Childs       []ChildCheck   // List of child checks
        CrossMsgs    []CrossMsgMeta // List with meta of msgs being propagated (a.k.a meta-tree).
}

// Checkpoint data structure
//
// - Data includes all the data for the checkpoint. The Cid of Data
// is what identifies a checkpoint uniquely.
// - Signature adds the signature from a miner. According to the verifier
// used for checkpoint this may be different things.
type Checkpoint struct {
        Data      CheckData
        Signature []byte
}
```

2.5.2 Power handling in child chains

The mining power in Filecoin's Expected Consensus is determined by the amount of committed capacity a miner has in the chain. Miners looking to increase their mining power need to add new sector to the network to increase their capacity. The actor responsible for keeping the power table and tracking the updates of power for each miner is a system actor called the *Power Actor*.

Storage miners in the Filecoin network have to prove that they hold a copy of the data at any given point in time. The proof that a storage miner indeed keeps a copy of the data they have promised to store is achieved through "challenges", that is, by providing answers to specific questions posed by the system. Each miner in the Filecoin network is represented on-chain through a *Storage Miner Actor*. This actor is the one responsible for handling all the logic related to the miner: adding new sectors, committing storage proofs, etc.

Hierarchical consensus has no impact on how data is stored in the Filecoin network, or how sectors and proofs are handled. If the Filecoin network is the root network for a hierarchical consensus, the *Power Actor* always lives in the root chain, and tracks the global power table for the whole hierarchy (recalling from section 2, existing miners in hierarchical consensus will always keep mining in the root chain in order not to dilute its power). Hence, sector bookkeeping and power information stays in the root chain of the hierarchy.

Miners, on the other hand, can move or even be created in any subnet of the hierarchy. Adding sectors and proving storage under this scenarios works like any other cross-net message. Miners interact with their miner actor in their subnet, which triggers the corresponding cross-net message to the power actor in the root chain when needed. This will require some tweaks in the current implementation of the *Storage Miner Actor*, so the current calls to the power actor are sent to the subnet's SCA for them to be relayed to the power actor in the root. Figure 8 depicts the architecture of the system including power and miner-related actors.

This subnet-oriented approach for Filecoin storage introduces a set of new opportunities and use cases: it allows to decouple sector storage and proofs, from sector bookkeeping and power management; it offers a foundation to decoupling retrieval markets and storage markets in the Filecoin network, which are currently very intertwined. By modularizing all storage-related functionalities (sector bookkeeping, miner operation, power actors, retrieval deals, payment channels, etc.) we open the door to the deployment of specialized subnets optimized for each of these tasks. Thus, we could spawn a subnet to handle retrieval deals in a specific location, a specialized subnet handling payment channels, or one for storage-based operations. This approach would prevent from impacting the throughput of the root chain with storage management operations. Each subnet can have its own consensus, distributing the load between of the Filecoin network according to the specific task. This constructions offer an additional level of flexiblity to the Filecoin network.

Finally, subnets may choose to also run in their chain Filecoin Expected Consensus. In this case, unless configured otherwise, the power assigned for each miner in the subnet is proportional to their power committed in the root Filecoin chain. Alternatively, some subnets may choose to only consider committed power in the chain accountable for the inner subnet consensus.

2.6 Leaving and killing a subnet

Members of a subnet may leave the subnet at any point by sending a transaction to the *Leave* function in the SA of the subnet in the parent chain. Thus, a miner looking to leave subnet $/root/t01/t12$ needs to trigger this function in actor $t12$ of subnet $/root/t01$. This functions triggers a message to the *ReleaseCollateral* function of the SCA in subnet $/root/t01$ to release and return the corresponding funds to the leaving miner. Miners can recover the stake when leaving the chain, but the amount of collateral recovered is determined by the locking period policy of the subnet (see sections 2.3 ,4. This policy is configurable in each subnet. If a miner leaving the subnet makes the collateral of the subnet to be below $minCollateral_{subnet}$, the subnet gets in an *Inactive* state, and it can't further interact with the rest of chains in the hierarchy or checkpoint to the top chain until the minimum collateral is recovered. If the subnet wants to be revived into an *Active* state and be able to interact again with the hierarchy, new peers will have to join and add new collateral to ensure a collateral larger than $minCollateral_{subnet}$ in the subnet.

Miners of a subnet may choose to kill a subnet by sending a message to the *Kill* function of the SA. In order to kill a subnet, a majority of the staked miners need to send the kill message to SA. Once the killing policy is fulfilled, a new message is sent to the *Kill* function of the SCA to release all the collateral for the subnet and return it to its owners. This function also returns outstanding balances from users to their addresses in the parent subnet.

Handling the state of killed subnets . A subnet may be killed while it is still holding user funds or useful state. If miners leave the subnet and take the collateral below threshold enforced by the hierarchical consensus to allow cross-network communication, users don't have a way to get their funds and state out the subnet. When reaching a killed state, all of the pending state in the subnet (like pending accounts with balance) is migrated back to the parent chain, and the chain and all related assets are removed from the

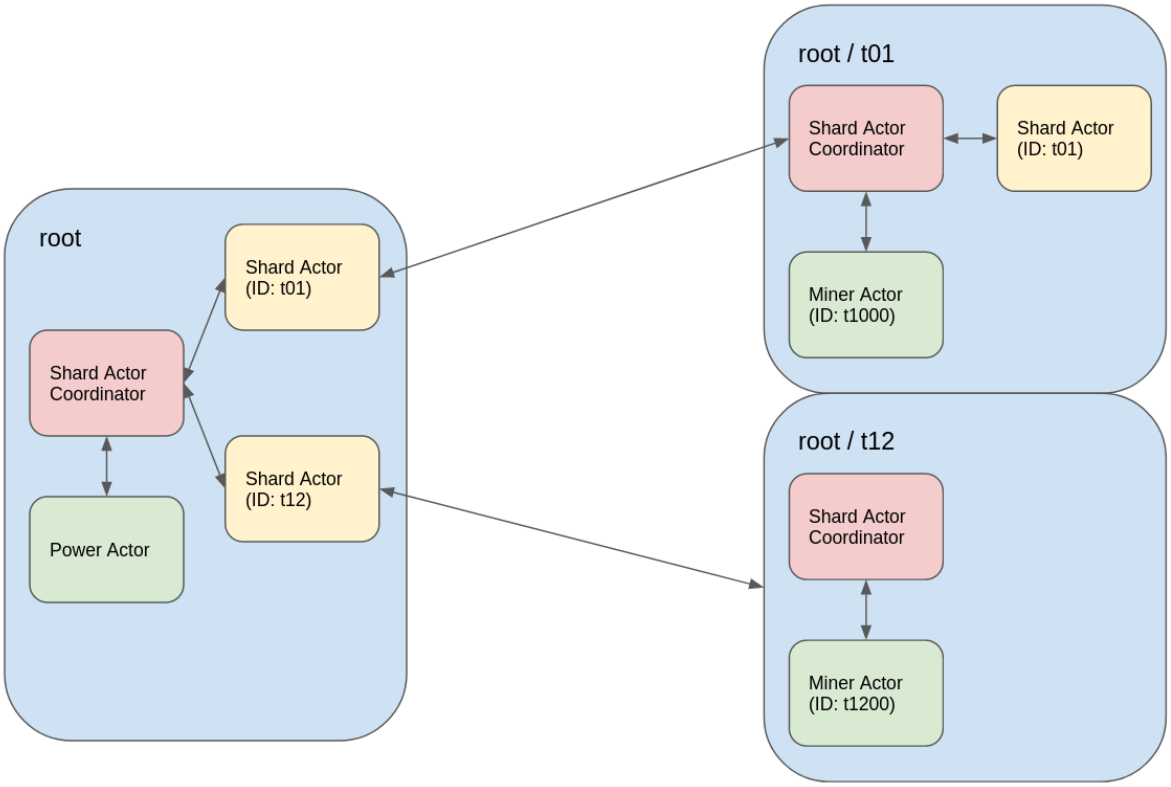

Figure 8: **Power management architecture in hierarchical consensus** The use of subnets enables the decoupling of certain parts of the Filecoin consensus parallelizing its operation. For instance, miners would be able to "live" in subnets while still onboarding power and interacting with the core protocol (mainly running in the root network).

system in the next checkpoint of the parent chain, making effective the removal of the subnet.

To prevent this from happening, SCA includes a *Save* function for any participant in the subnet to be able to persist the state of the subnet[4]. Users may choose to perform this snapshot with the latest state right before the subnet is killed, or perform periodic snapshots to keep track of the evolution of the state. Through this persisted state and the checkpoints committed by the subnet, users are are able to provide proof of pending funds held in the subnet, or a specific part of the state that wants to be migrated back to the parent (subnet state migrations are discussed more in detail in section **??**).

Unfortunately, if a majority in the subnet is malicious, the state snapshots saved for the subnet may hold forged state, however, this is may also be the case for the proofs propagated in the subnet's checkpoints. In any case, due to the *firewall* property of subnets, the impact of an attack in a subnet is limited for the upper levels of the hierarchy, and additional schemes for proving frauds may be introduced in next iterations of the protocol.

2.7 Cross-net transactions and execution

Users in a subnet interact with other subnets through a cross-net transaction. The propagation of a cross-net transaction may slightly differ depending on the location of the subnets in the hierarchy.

In the first implementation of hierarchical consensus, the propagation of cross-net transactions is exclusively done through two type of transactions: top-down and bottom-up transactions. These are the basic primitives used to route any kind of message through the hierarchy. According to if the message needs to move up or down, bottom-up or top-down transactions are used respectively. the hierarchy. Thus, message propagation can be classified into:

Top-down transactions are cross-net transactions originated in a upper subnet that share the same ID prefix to the destination subnet located lower in the hierarchy. This is, for instance, a transaction from $root/t01$ to $/root/t01/t12$. Child subnets need to always sync (i.e., keep the latest state) with their parent chains to be informed about updates in the state of the SCA and SA in the parent chains, $root/t01$ in our example. A top-down transaction is triggered by sending a message to the *Fund* function of the SCA in the destination subnet parent chain, $root/t01$, with the amount of tokens that to be transferred, or by calling *CrossMsg* in SCA with a message whose destination is one of the children of $root/t01$. When a new top-down transaction is triggered, the SCA of the parent increments a unique nonce to each new top-down transaction and stores it in the SCA state. This additionally triggers a state change in the SCA of the parent chain ($root/t01/$) freezing the funds injected to the subnet through the top-down message. This funds are frozen until a down-top transaction releases them back to the parent. Thus, the SCA in $root/t01/$ keeps track of the circulating supply of the subnet.

Nodes in a subnet implement two types of message pools: a message pool that keeps unverified messages originated and targeting the subnet; and a cross-message pool that listens for unverified cross-net messages directed (or traversing) the subnet. Blocks validated by the consensus algorithms of subnets additionally include the new cross-net messages targeting the subnet. Miners of the subnet continuously monitor their cross-net message pool for unverified cross-message. We can think of the cross-message pool as a dedicated mempool responsible for collecting and selecting new cross-net message to be verified in the next block of the subnet. Miner's cross-message pools check the value of *AppliedTopDownNonce* in the SCA of the child subnet, $/root/t01/t12$, to get the latest nonce of a cross-message executed in the child subnet (k). With this, the cross-message pool picks up every top-down transaction stored in the SCA of the parent chain, $/root/t01$, with a nonce greater or equal to k. These top-down transactions selected by the cross-message pool are included in the next block proposal. As it is described below, the cross-message pool is responsible for collecting every cross-net message targeting the subnet, i.e. both, top-down and bottom-up cross-net messages.

[4]The implementation of this scheme will have to be deferred to when FVM is shipped, so actors can have native access to Filecoin storage

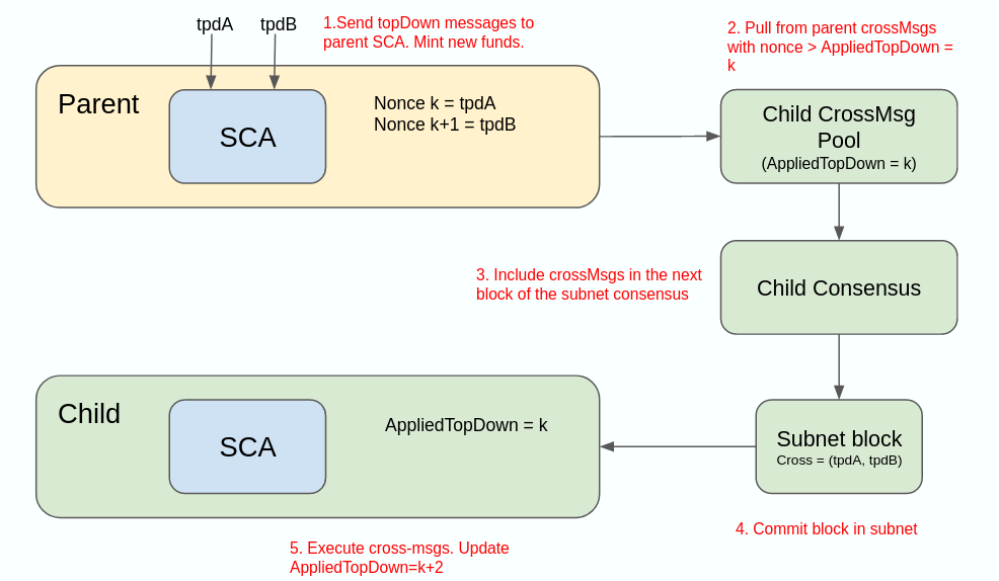

Figure 9: **Commitment of top-down cross-net messages.** When a top-down message is committed by a subnet's parent, it is assigned the next subsequent nonce, and is added to the list of unverified top-down messages for the destination subnet (child). The cross-net message pool of nodes in the child listens to changes in the state of the parent, and pulls any new unverified cross-net message for the subnet. These messages are proposed inside the next block of the consensus and are ordered and verified as any other messages sent from within the subnet. When the block including cross-net messages is committed, they are applied and the corresponding state changes in the subnet triggered.

When a new block including top-down cross-net messages is verified in the subnet consensus, the top-down messages are committed, and every node receiving the new block executes the cross-messages to trigger the corresponding state changes and fund minting in the subnet. Cross-net messages are executed by implicitly calling the *ApplyMsg* function provided by the SCA in every node when a new block including cross-net messages is received. This function also updates *AppliedTopDownNonce* to the highest nonce of the top-down message applied plus one. Thus, the commitment of a simple *Fund* top-down message injecting new funds to an address of the subnet from the parent, translates into the freezing of the funds in the parent SCA, and the minting of new funds that are deposited in the target address when the cross-net message is committed in the subnet (all of this is triggered through the corresponding call to SCA's *ApplyMsg*). An illustration of how top-down transactions are handled by the hierarchical consensus is depicted in Figure 9.

Bottom-up transactions are cross-net transactions originated in a subnet lower in the subnet towards an upper subnet with the same prefix. This is, for instance, a transaction from $/root/t01/t12$ to $root/t01$. A bottom-up transaction function of the SA is triggered by either sending a message to the *CrossMsg* function of the SCA in the child chain ($/root/t01/t12$) with the amount to be sent, or a *Release* message in the SCA with the amount of tokens to be sent from the subnet back to a parent address. Sending these messages to SCA triggers the burning of the funds being sent/released in the subnet (by performing a transaction to the subnet's burning address), and appends the message in the corresponding *CrossMsgMeta* with source the child subnet, $/root/t01/t12$, and either the parent subnet for *Release* messages, or the corresponding ID subnet for *CrossMsgs*.

The subnet SCA additionally keeps a *CrossMsgMetaRegistry* which stores a map with key the CID of *CrossMsgMetas* propagated through checkpoints, and the corresponding *CrossMsgMeta* as value. This data structure is used by miners to fulfill content resolution requests from peers in other subnets looking to resolve CIDs behind checkpoints, cross-net messages and *CrossMsgMetas*. Further details about the content

resolution protocol can be found in section 2.7.1. Every checkpoint period, SCA collects all *CrossMsgMeta* from bottom-up transaction originated in the subnet and MsgMetas propagated from the subnet's child subnets, and include them in the next checkpoint to propagate them up the hierarchy.

When the checkpoint from the child subnet, say $/root/t01/t12$, is committed in the parent chain, $/root/t01$, the SCA of the parent chain inspects all *CrossMsgMeta* in the checkpoint and checks the ones directed to it. Bottom-up messages targeting other subnets are propagated further up the hierarchy in the next checkpoint. On the other hand, each bottom-up *CrossMsgMeta* targeting the current subnet is assigned an increasing nonce for its posterior validation and application by the subnet's consensus algorithm (the same way it is done in child subnet for top-bottom transactions). The cross-message pool for the parent subnet inspects the SCA and collects all the unverified *CrossMsgMeta*, i.e., those with a nonce greater or equal than the value of *AppliedBottomUpNonce* (k). In this case, the cross-message pool only has the CID of the MsgMeta that points to the cross-net messages to be applied. To resolve the cross-msgs behind these CIDs, the cross-message pool makes a request to the content resolution protocol (Sec. 2.7.1) to get the raw messages propagated from child subnets. As messages are resolved successfully by the miners' cross-message pool, they are included in blocks for their validation and commitment by the consensus. Similarly to handling top-down messages, when a block including bottom-up messages is validated and propagated to the rest of the subnet, it is applied by calling SCA's *ApplyMsg* triggering the corresponding fund releases, state changes, and nonce updates in the subnet state.

Cross-net message execution failure . The execution of *ApplyMsg* expects both, top-down and bottom-up cross-msgs, to be applied in order according to their nonce in SCA. If one of the messages for a specific nonce can't be applied and keeps failing when trying to be applied in the subnet's SCA, the subnet consensus could be stalled. This represents an attack vector for a DDoS attacks in subnets. To prevent this from happening, if a cross-msg can't be applied by ApplyMsg in SCA, it is disregarded and *AppliedTopDownNonce* is incremented to the next nonce as if the previous messages was applied successfully. Cross-msgs have to go through several checks before they are stored in SCA and provided to the subnet consensus through the cross-msg pool, but still (and especially for arbitrary messages) the application of these messages may fail. To revert state changes that may have happened in other subnets through the propagation of the failing cross-net message through the hierarchy (like frozen and burnt funds), a new cross-net message is initiated by SCA with the source and destination reverted. As this new cross-net messages propagates through the hierarchy, the corresponding state changes are conveniently reverted to notify the application failure in destination.

Path transactions are cross-net transactions in which source and destination subnet have a common ancestor (e.g., *root*) which is different from the source and destination. In this case, the transaction is handled as a combination of bottom-up and top-down transactions until it reaches its destination. For path transactions the source, with for instance an ID $/root/t01/t12$, sends a message to the *CrossMsg* function of the SCA in the source chain $/root/t01/t12$ with the funds to send as if it was doing a bottom-up transaction indicating its destination, $/root/t02/t23$. This transaction will be propagated through checkpoints subnet by subnet up to the root (or its immediate common parent) as illustrated in figure 12 and detailed bottom-up transactions. As the checkpoint moves up in the hierarchy, funds are conveniently released and burned in each of the subnets as cross-net messages flow through the hierarchy, updating their circulating supply. As no common ancestor was shared up to the root, the transaction is not picked by any of the subnets in the path releasing funds in the root.

At this point of our example, when the checkpoint is committed in the root chain, a top-down transaction is automatically triggered by the SCA in the root chain towards $/root/t02$ when it sees that there are pending cross-net transactions in the checkpoint. This triggers a set of top-down transactions, freezing and minting tokens until the funds from $/root/t01/t12$ reach its destination, $/root/t02/t23$. Figures 12 and 13 illustrates the checkpoint propagation and account updates as different cross-net messages flow through the hierarchy.

According to the route messages need to follow through the hierarchy, and the specific consensus algorithms run by each of the subnets, the propagation of these transactions may be quite slow. To accelerate

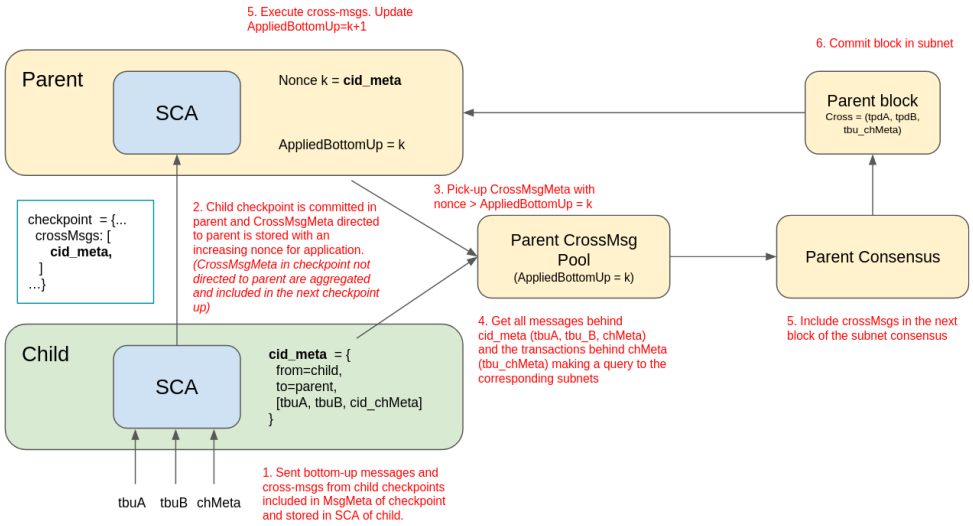

Figure 10: **Commitment of bottom-up cross-net messages** Bottom-up messages are propagated through checkpoints. The child subnet aggregates cross-net messages from within its own subnet or propagated from its childs inside a meta-tree, and include it in the next checkpoint. When the parent receives the checkpoint from the subnet, it checks what meta-trees are directed to itself, and which ones need to be aggregated and propagated further through the hierarchy. Meta-trees to other subnets are included in the next checkpoint of the parent, while the ones targeting the current subnet are assigned the next unique bottom-up nonce and stored for their commitment. The cross-net message pool for nodes in the parent periodically listen for new unverfied meta-trees to be committed and applied in the subnet. When the next unverified meta-tree is picked up, the messages behind it are fetched leveraging the cross-message resolution protocol (section 2.7.1). These messages are ordered and proposed in the next block of the subnet's consensus, and consequently committed and applied to trigger the corresponding state changes (as it was done for top-down messages).

the process, each SA in the path from $/root/t01/t12$ can send a direct message to $/root/t02/t23$ directly certifying that the user is the legitimate owner of the funds. This information can be used by the destination subnet (depending on the finality required for the actions to be performed) as good enough to start operating as if these funds where already settled and available in the subnet.

What is more, as described in Section 2.5, the list of cross-net message meta-trees only include the aggregation of messages triggered from a subnet to a specific destination. Subnets receiving a checkpoint need to explicitly request the list of messages behind the CID that aggregates the list of transactions they are interested in through the content resolution protocol. If we look at the example in figure 12, the checkpoint from subnet E propagated to its parent B, doesn't include every single cross-net message and cross-net message meta-trees, but a digest of all message to specific destinations. Thus, we see that E only propagates $cid_{C-E} = cid(m1, m4)$ in the tree with D as a destination to notify the hierarchy that there are pending transactions for D. When this information gets to B, and sees that there are still pending cross-net transactions that are not directed to her subnet, it recursively aggregates in their checkpoint updating the source and the destination, propagating $cid_{C-B} = cid(cid_{C-F}, cid_{C-E})$ including the CIDs for all the cross-net messages directed to C. In this way, we minimize the amount of information that needs to be propagated to the top of the hierarchy.

2.7.1 Subnet Cross-net message resolution protocol and Data Availability

Data/message availability. To get messages for a specific CIDs, two approaches can be used:

- A *push* approach, where as the checkpoints and *CrossMsgMeta*s move up the hierarchy, miners publish to the pubsub topic of the corresponding subnet the whole DAG belonging to the CID including all the messages targeting that subnet. Content is pushed to other subnets by publishing a *push* message specifying the type of content being pushed (a message, MsgMeta, or checkpoint) along with its CID in the destination subnet's corresponding pubsub topic. When peers in the subnet come across these message, they may choose to pick them up and cache/store it locally to handle resolutions faster in the future or discard them (which would require explicitly resolving the content when its needed.).

- A *pull* approach where upon a subnet receiving a checkpoint with cross-net messages directed to it, miners cross-message pools publish a *pull* message in the source subnet pubsub topic to resolve the cross-net messages for a specific CID found in the tree of cross-message meta and targeting the subnet. These requests are responded by publishing a new *resolve* message in the requesting subnet with the corresponding content resolution. This new broadcast of a content resolution to the subnet's pubsub channels give every cross-message pool a new opportunity to store or cache the content behind a CID even if they don't yet need it. This also prevents from having lots of duplicate messages and ad-hoc communication between subnet miners.

- A *peer-to-peer* approach, where miners resolve content by directly contacting a known peer from the subnet instead of having to broadcast request and pick up responses through the pubsub layer. This approach is more straightforward but requires prior knowledge of members belonging to a subnet, or a complementary discovery protocol to query the participants of a subnet. In the initial implementation, only the pubsub approach will be supported, however, in the future this peer-to-peer protocol will be introduced as a reliable fallback in case the pubsub approach fails.

When the destination subnet receives a new checkpoint with messages directed to it, it is only provided with the CID of the messages (i.e. the CrossMsgMeta) that were sent its way. For the subnet to be able to trigger the corresponding state changes for all the messages, it will need to fetch the list of messages behind that CID as illustrated in Figure 11. Some peers may choose to directly push the corresponding messages behind a CrossMsgMeta to the destination address once a checkpoint has been signed. When the checkpoint is committed in the destination subnet (*blue lines in the figure*), peers check if the corresponding cross-net messages have already been pushed to directly apply them. This is the preferred behavior, as it enables destination subnets to start performing some actions with cross-messages while the checkpoint has not yet

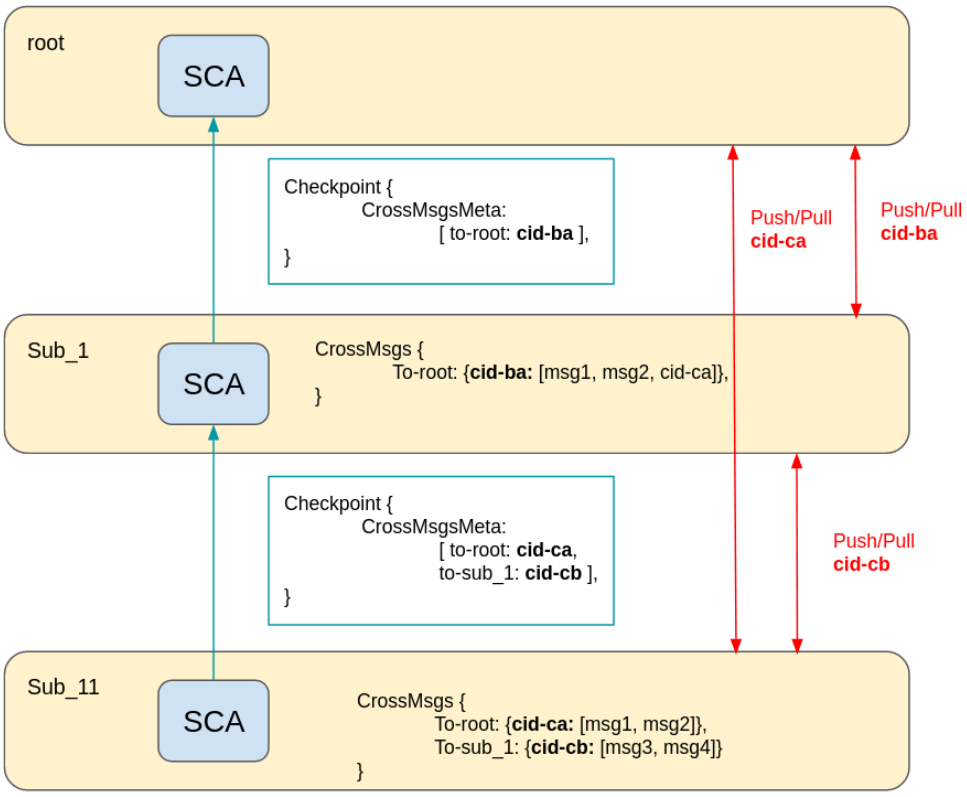

Figure 11: **Content resolution protocol for cross-net messages.** Whenever a subnets submits a new checkpoint to its parent, it pushes the messages behind the CIDs included in the meta-trees of the checkpoint, so they can resolved immediately. If a subnet comes across a CID in the meta-tree that it can't resolve locally (either because it missed the *Push* message from the subnet, or because it recently joined the network), it can resolve the messages behind the CID by sending a *Pull* request to the subnet that originated the meta-tree.

arrived, and to apply cross-net messages as soon as the checkpoint is committed. If when a checkpoint arrives to a subnet, the cross-net messages propagated haven't been pushed, the cross-net message pool needs to proactively pull them using the content resolution protocol the raw cross-net messages propagated in the checkpoint's CrossMsgMeta.

Peers requesting content from others subnets trust that the peers in the other network from which a checkpoint with cross-msgs has been propagated will resolve the content resolution requests sent to the subnet. Intuitively, the node that triggered the cross-net message has no incentive on denying access to this data (as his funds have already been burnt), but data availability is an issue that is worth addressing further. The aforementioned design of the content resolution protocol assumes good behavior from peers in other subnets (i.e., they always answer successfully to content resolution requests), and that the data in the subnet is always available. In a real environment these assumption may not hold, and more complex schemes may be needed to incentivize subnet peers and ensure that every content resolution request between subnets is fulfilled, and a high-level of availability of data.

2.8 Generality of the approach beyond payments

The examples above illustrated cross-net message propagation as token exchanges between accounts in different subnets. However, it is worth noting that this scheme is general enough to accommodate the propagation of any arbitrary message between subnets. Messages would be picked up by the specific destination subnet

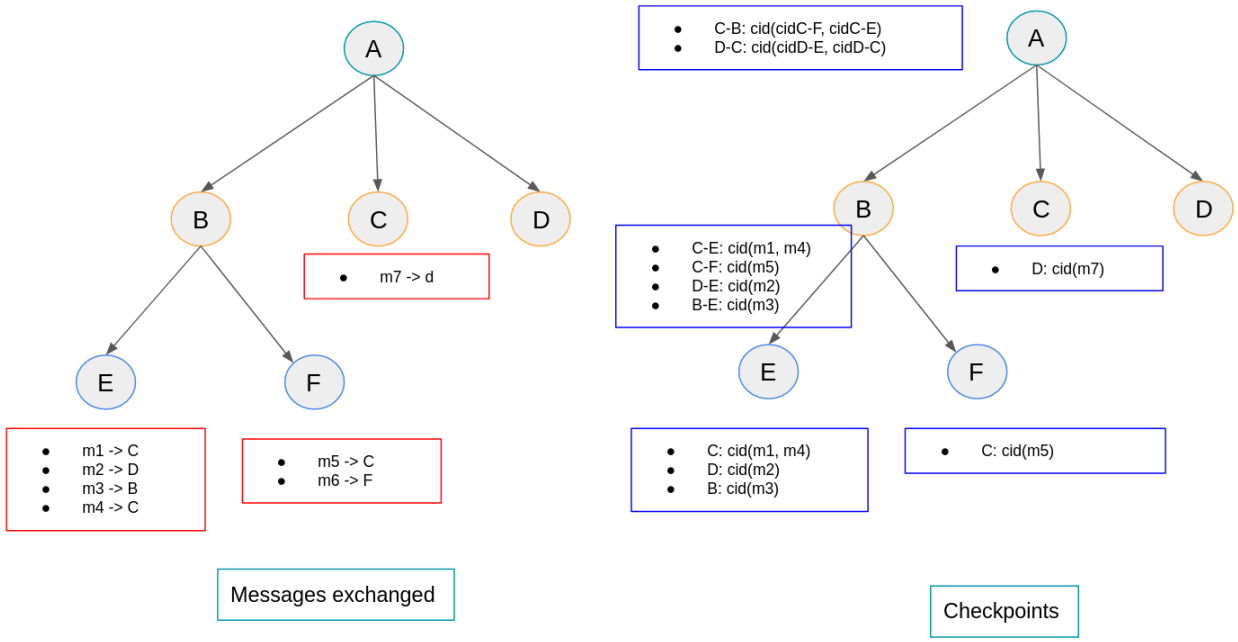

Figure 12: **Transaction propagation through checkpoints.** As cross-net messages are propagated to the top of the hierarchy through checkpoints, they are aggregated inside meta-trees.

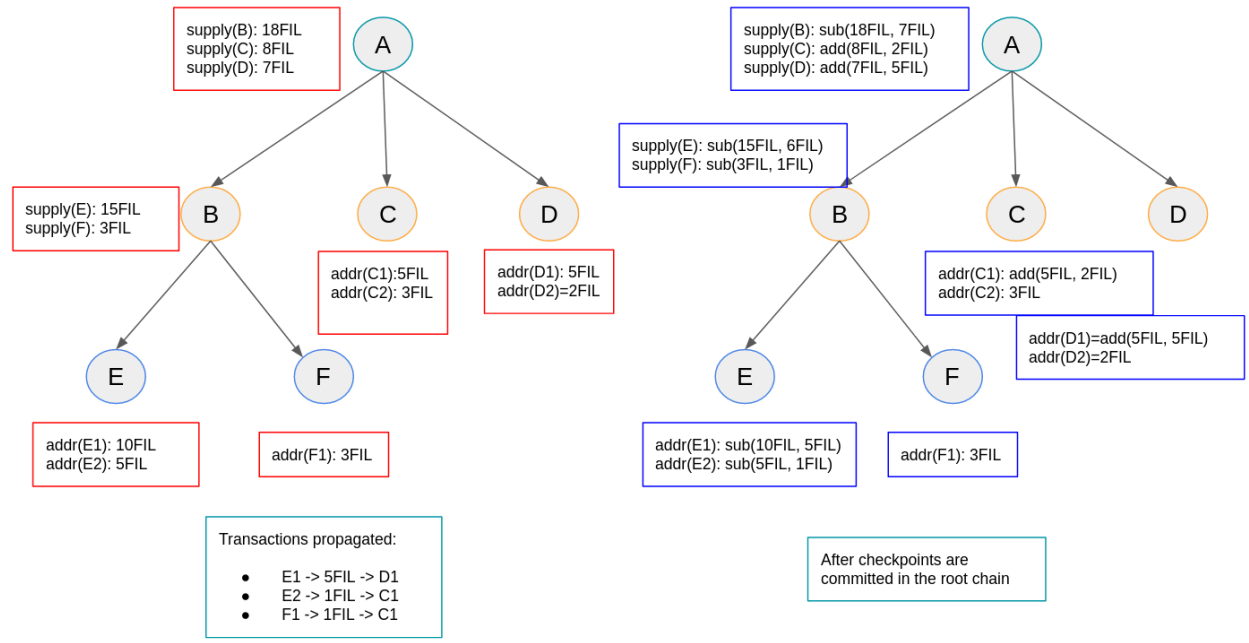

Figure 13: **Cross-net FIL transaction balance updates.** The propagation of cross-net messages through top-down and bottom-up transactions trigger the corresponding freeze and release of funds, and updates in the circulating supply of the subnets traversed according to the value included in the propagated messages. This is handled by SCA.

and trigger the required state changes.

An issue arises when these state changes need to be atomic and require from the state of different subnets. A good example of this is the exchange of ownership of two assets hosted in different subnets. The state change needs to be atomic, and it requires from state that lives in both subnets. To handle this atomic transactions, users in the subnets can choose any subnet in the hierarchy in which they both have certain level of trust to migrate the corresponding state and orchestrate the execution. Generally, subnets will choose the closest common parent as the execution subnet, as they are already propagating their checkpoints to it and leveraging its trust.

2.8.1 Atomic Execution

Let u_i, u_j be two users from different subnets, and let f be a function from a tuple of input states (i_{k1}, i_{kn}) to a tuple of output states (o_{k1}, o_{kn}), where both input states, i_k, and output states, o_k may belong to different subnets (sn_i). In short, a cross-subnet atomic execution of f appears to be executed as a single transaction in which all input/output states $(i_{k1}, i_{kn}, o_{k1}, o_{kn})$ belong to the same subnet. This protocol has the following properties:

- *Atomicity*: If the protocol is executed correctly, all subnets involved have the output state of the execution, o_k, available as part of their subnet state.
- *Timeliness*: The protocol always finishes in a bounded amount of time. The execution can be aborted at any time by any of the users, u_k, involved in it.
- *Unforgeability*: No entity in the system (user or contract) is able to forge the inputs and outputs provided for the execution, or the set of messages orchestrating the protocol.

Finally, the data structures used by the protocol needs to ensure the *consistency* of the state in each subnet, i.e. the output state of the atomic execution can be applied onto the original state (and history) of the subnet without any conflicts.

The atomic execution protocol is implemented in the following stages:

Initialization To signal the start of an atomic execution, the users involved interact off-chain to agree on the specific execution they want to perform and the input state it will involve (for instance, in the case of an NFT exchange, the specific assets to be exchanged). To start the execution, each user needs to lock in their subnet the state that will be used as input for the execution. This prevents new messages from affecting the state and leading to inconsistencies when the output state of the execution is migrated back. The locking of the input state in each subnet signals the beginning of the atomic execution.

Off-chain execution . Each user only holds part of the state required for the execution. In order for them to be able to perform the execution locally, they need to request the state locked in the other subnet. Thus, each user requests from the other subnets the locked input states involved in the execution by CID leveraging the cross-net message resolution protocol. The CID of the input state is shared between the different users during the initialization stage. Once every input state of the execution is received, every user runs locally the execution to compute the output state.

Commit atomic execution in parent subnet As users compute the output state, they commit it in the SCA of the parent subnet. This message includes the CID of the output state, as well as the list of parties involved in the atomic execution. SCA waits for the commitment of the output state computed by every user involved in the execution to mark the execution as successful. To prevent the protocol from blocking if one of the parties disappears half-way, any user is allowed to abort the execution at any time, and to unlock their state by sending an *ABORT* message to SCA.

Termination All subnets involved in the protocol listen to events in the SCA of the execution subnet. When SCA receives the commitment of all the output states computed, if they all match, the execution is marked as successful to notify the subnets that is safe to include the output state in their subnet's state, and to unlock the input state. If instead of a success in the execution, the SCA notifies an *ABORT*, each subnet will unlock their input state without performing any changes in their local state.

Example of atomic execution: NFT exchange TODO:

3 System Evaluation

4 Incentives Model?

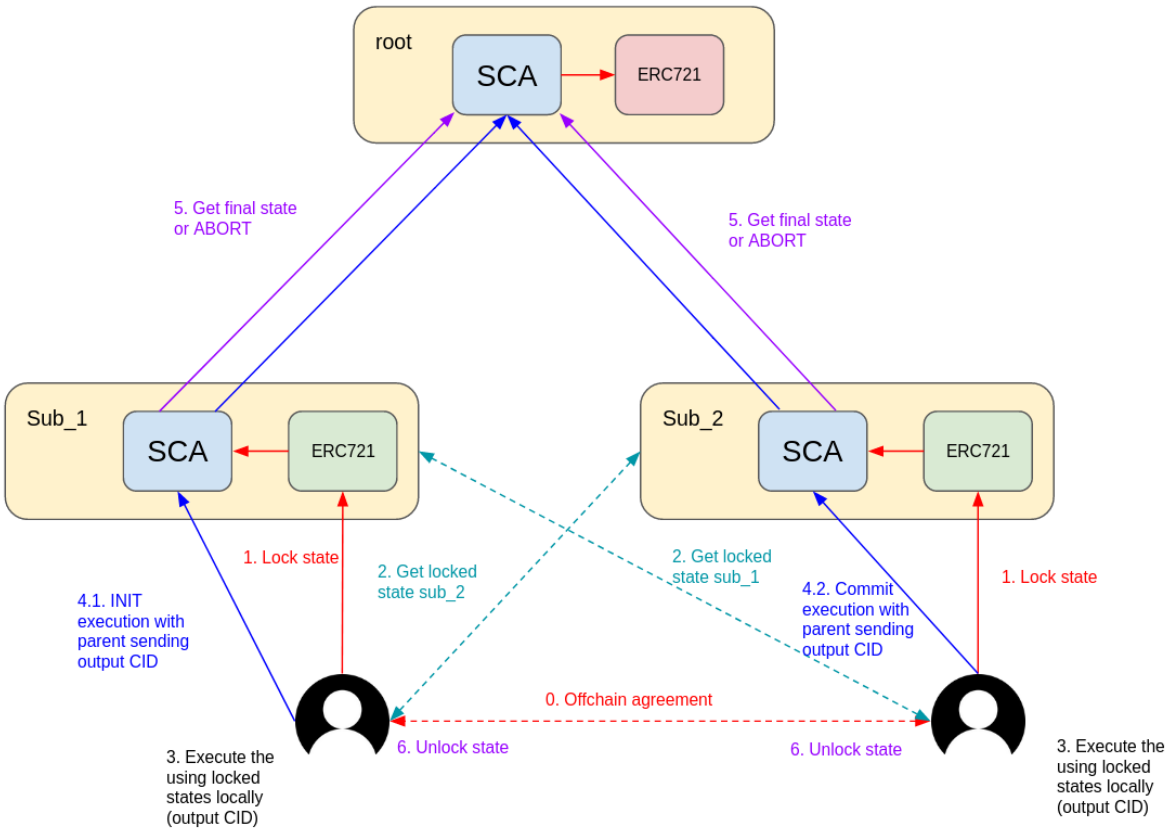

Figure 14: **Atomic execution protocol**. The protocol starts with an offchain agreement between the different users involved. All parties lock the state they want to use as input in the atomic execution and notify the SCA in the parent that they want to start an atomic execution. Once all input states have been locked, all users collect the pending inputs from other subnets to perform the atomic execution. Each perform the execution offchain. Every user submits the CID of the output state of the execution in the SCA of the parent-chain. The SCA waits for all the parties involved to submit the out state, and checks if they all matched. If the state matches, the SCA marks the execution and successful and users are allowed to migrate the output state to their subnet. At any point, users are allowed to abort the execution by sending a message to the SCA of the parent.