Filecoin Hierarchical Consensus Specification (aka Project B1)

ConsensusLab
Protocol Labs

January 11, 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 the chain. Every subnet can run its own independent consensus algorithm and have its own security 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 its ancestor subnets. Moreover, ancestor subnets *help secure* their descendant subnets — for instance, checkpointing a Proof-of-Stake subnet into its parent may help alleviate notorious 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 very 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.

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 miners (or 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). Mining rewards in the rootnet are proportionately distributed among all subnets in the hierarchy, and in order for validators in a subnet to unlock their rewards, they need to commit the corresponding checkpoint in their parent chain. Subnets may optionally define additional miner rewards, in a token different from the parent token. Tokenomics and incentives are detailed in Section 4.

We follow the approach in which a miner in a given subnet s has trusted access to state of all ancestor subnets of s. We implement this by having a miner on subnet s itself sync with the chain of all ancestor subnets (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. All subnet validators in the system run full nodes in the rootnet preventing the dilution of security of the rootnet which may result from creating new subnets. At any given point in time a miner can obtain rewards from the rootnet, and from the child subnets in which it is participating as a miner **TODO: make sure this is as intended**. Nodes are allowed to spawn new subnets from a child chain as depicted in the figure for $Subnet_{21}$ and $Subnet_{22}$.

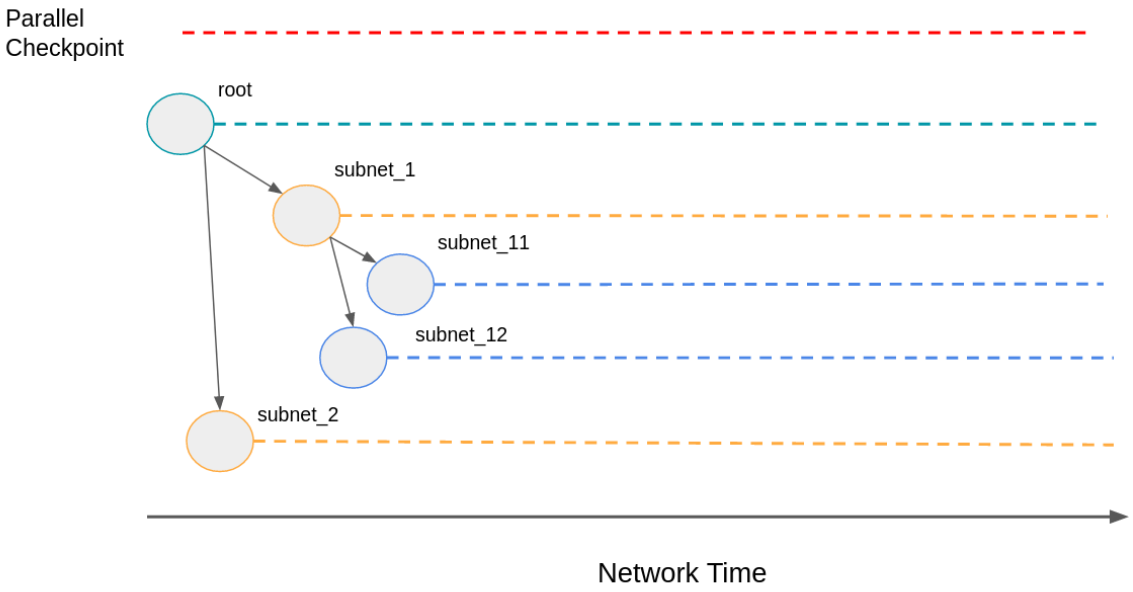

Figure 1: System Overview

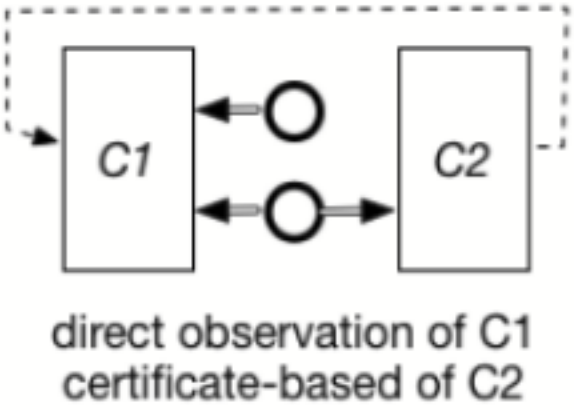

Figure 2: Cross-chain transactions approach

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 want 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 supply in the child chain. The circulating supply is determined by the balance between cross-chain transactions entering the subnet and cross-chain 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-chain 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 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.

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 1. The list of arguments of this function depends on the specific implementation of the 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 miners participating in the subnet. This kind of requirements ensures

[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

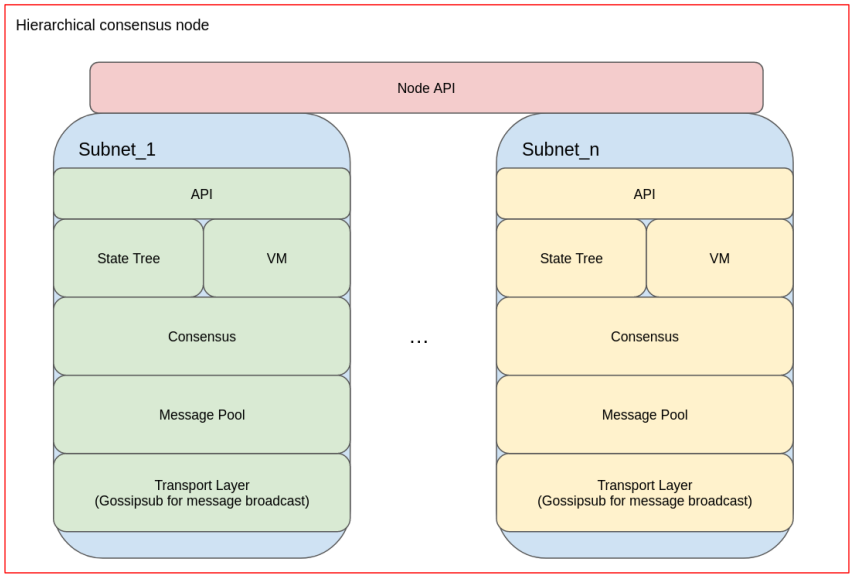

Figure 3: Hierarchical Consensus node and subnet stack

that new participants joining a subnet (that may be used for a very 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 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 *minStake* 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 *AddStake* including the amount of funds to be added to the subnet's stake.

- *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 a 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-chain transactions.

- *Leave*: This function can be used by miners and users looking to leave the subnet and recover their stake. 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 *ReleaseStake* function in the SCA to recover the corresponding part of the stake

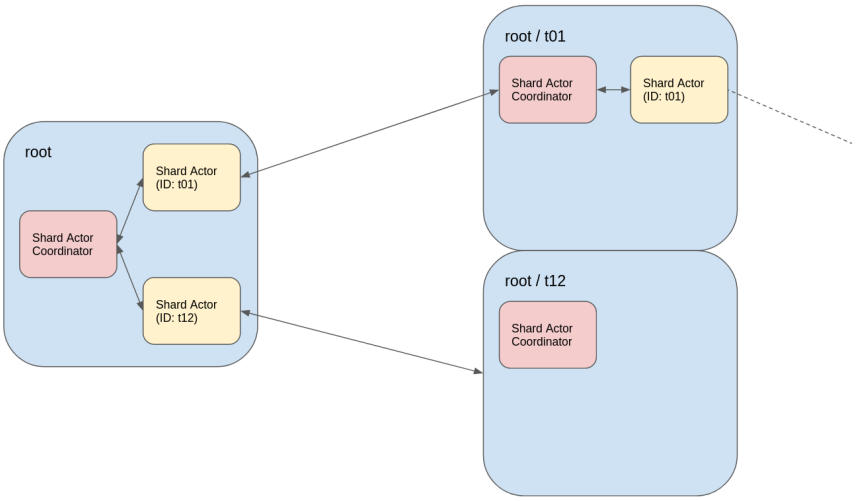

Figure 4: Subnet Actor and Subnet Coordinator Actor architecture

in the coordinator. Some additional checks may be required in the actor to ensure that the minimum stake 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 stake of the subnet 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.

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.

Algorithm 1 Subnet Actor

```
1: Struct SubnetActorState contains
2:     ID                          ▷ ID to uniquely identify the subnet
3:     name                        ▷ Human readable name of the subnet
4:     parent                      ▷ ID of the parent subnet
5:     consensus                   ▷ Consensus algorithm running in the subnet
6:     miners                      ▷ List of miners in the subnet
7:     totalStake                  ▷ Total amount staked in the subnet
8:     stakeMap                    ▷ Map with the amount staked by each miner
9:     status                      ▷ Status of the subnet (Instantiated, Active, Terminating, Killed)
10:    checkpoints                 ▷ Checkpointing state for the subnet
11:    checkPeriod                 ▷ Number of epochs of checkpointing period
12:
```

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 (algorithm 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 stake 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 stake is fulfilled. This stake will have a key role for the security model in subnets. Finally, if the subnets runs a consensus algorithm that uses the same mining resource as the *rootnet*, all the power that the miner poses in the root is delegated to the child chain to prevent a power dillution in the root chain. This would be the case if Filecoin is the rootnet, and a new subnet chooses to run Filecoin Storage-based consensus. A miner joining the subnet would bring its storage power to the new subnet.

- *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 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. In the next checkpoint tick, this aggregated checkpoint is propagated to the top of the chain.

- *RawCheckpoint*: It constructs and returns the raw checkpoint that needs to be signed by miners and committed to the corresponding SA before the next tick. 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.

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

- *ReleaseStake*: *AddStake*'s counterpart, it releases an amount of stake to a specific address from the subnet's stake.

- *SubmitFraud*: Submits a fraud proof for a misbehaving miner from a subnet. A valid fraud proof for miner slashes its stake and distributes the slashed stake proportionately to all the users impacted by the attack. The fraud protocol is detailed in section 2.8.

- *Fund*: Use to send top-down transactions to inject new funds in a spefific 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 stake 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 the consensus algorithm to execute cross-messages from other subnet trigger the corresponding state changes in the subnet. See section 2.7 for a detailed description of how this function works.

Algorithm 2 Subnet Coordinator Actor

```
1: Struct subnetActorCoordinatorState contains
2:     networkName                          ▷ Name of the network
3:     totalsubnets                         ▷ Total number of subnets in the network
4:     minStake                             ▷ Minimum stake required to register new subnet
5:     subnets                              ▷ List of active child subnet
6:     checkpoints                          ▷ Map of checkpoints committed in the network.
7:     checkPeriod                          ▷ Number of epochs of checkpointing period
8:     checkMsgMetaRegistry                 ▷ Registry of msg metadata propagated up in the hierarchy from checkpoints
9:     bottomUpMsgMetaRegistry              ▷ Registry of msg metadata from bottom-up messages
10:    appliedTopDown                       ▷ Next nonce of cross topDown message to apply
11:    appliedBottomUp                      ▷ Next nonce of cross bottomUp message to apply
12:    Struct subnet contains
13:        ID                               ▷ CID to uniquely identify the subnet
14:        status                           ▷ Status of the subnet (Instantiated, Active, Terminating, Killed)
15:        prevCheckpoint                   ▷ Previous checkpoint for the subnet
16:        totalStake                       ▷ Total amount staked in the subnet
17:        funds                            ▷ Balance table with funds frozen by address
18:        circSupply                       ▷ Circulating supply in subnet
19:        topDownMsgs                      ▷ List of topDown messages indexed by nonce
20:
21:
```

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, checkpointing, 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, the subnet needs to be registered in the SCA of the parent chain. For a subnet to be registered in the SCA, the actor 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. To perform this transaction, the SA should have enough funds available, other peers need to have joined and staked in the SA according to its policy to fund the register transaction. For a subnet to be registered, at least $minStake_{subnet}$ needs to be staked in the SCA.

Subnets can run any consensus algorithm to validate blocks of the chain. If the child subnet's mining resource is fungible and of the same nature of the one used in the parent subnet, the mining power in the child subnet for a miner will be equal to the one it is currently committing in the top chain to ensure that power is not diluted in the upper levels of the hierarchy. If for instance, the parent and child subnets run Filecoin Storage consensus, the mining power of the miner will be determined by the power committed by the miner in the top chain. Miners can only commit their power (i.e., fungible resource) from the top chain in a single subnet. On the other hand, if the child chain runs a consensus where the mining resources is not related with the one used in the parent chain, miners will dedicate new resource to mine in the chain. This is the case, e.g., when the parent chain runs a Filecoin Expected consensus and the subnet runs a Nakamoto consensus.

Miners of the root in *hierarchical consensus* will mine in two chains in parallel: in the root chain, which is the root trust anchor responsible for the security of the system; and in the subnet the miner belongs to. The reason for this is that a *hierarchical consensus* must support the creation of new subnets without diluting in any way the security guarantees of the rootnet.

Optionally, 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 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 use 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[2]. This will be key for every cross-chain 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 stake 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

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

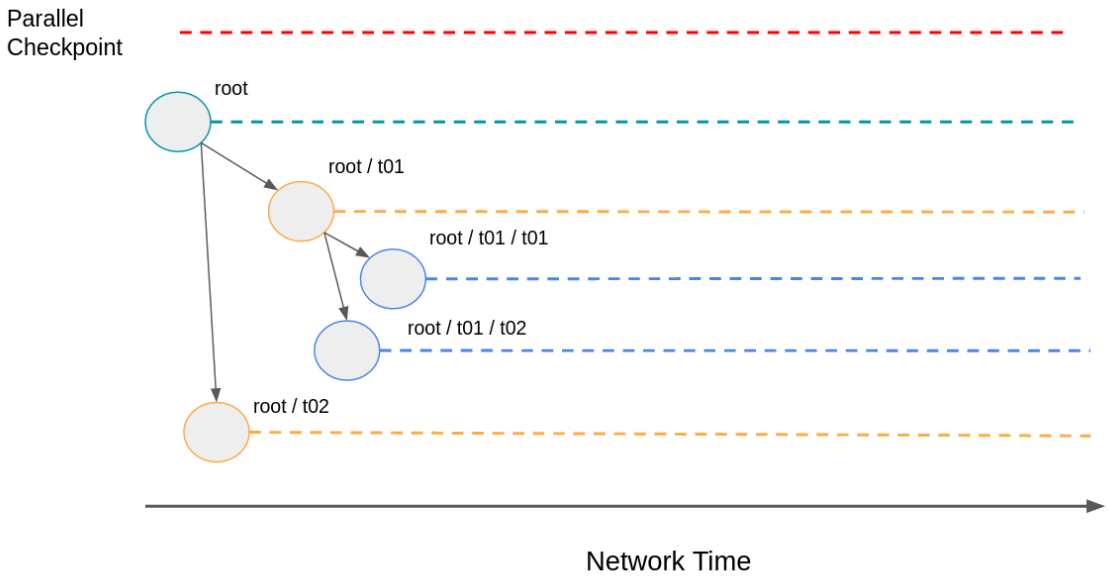

Figure 5: Subnet Naming Convention

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.

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. The low-level details of the checkpointing protocol are described in ConsensusLab Project B2.[3] 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, 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

[3]See `https://www.overleaf.com/project/611d37cf28642370db6ceeb1`

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 stake gets below $minStake_{peer}$ after being slashed, it will lose mining rights in the subnet. If the fraud proof includes enough information, the slashed stake of the miner is proportionately distributed in the parent chain to the addresses of the parties impacted in the attack. Further details on slashing and fraud proofs can be found in section 2.8.

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. Let's illustrate this using the Filecoin network as an example: in Filecoin new tipsets are effectively mined every 30 seconds. Each of these tipsets unlock a reward in the form of X new FIL minted in the network. A hierarchical consensus over Filecoin would distribute this X FIL every 30 seconds among the subnets of the hierarchy. Thus, the rewards subnets miners get are: (i) the transaction fees of the transactions that involved the subnet chain and (ii) certain amount of the total tokens minted in the root chain. Child chains can unlock their rewards from minted FILs of the root chain by committing checkpoints in the parent chain. This incentivizes the commitment of checkpoints from child chains. The specific distribution and economics of rewards is fleshed out in section 4. 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.

Miners can access 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-messages field and with its subnet as a destination. See figure 6 An extended description of this protocol can be found in section 2.7.

Algorithm 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 (algorithm 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 $height$.

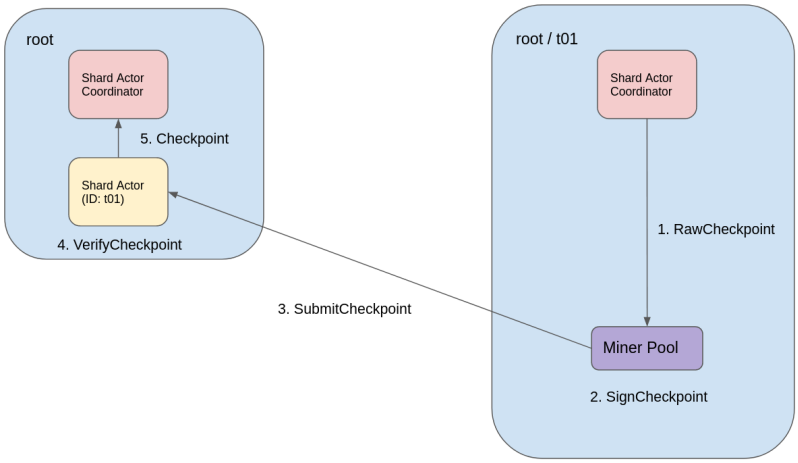

Figure 6: Checkpointing process

- A pointer to the CID of the previous checkpoint generated by the subnet.
- A tree of cross message metadata including of 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-message nonce, and the cid (message digest) of the message. This tree of CrossMsgMeta gets updated with every new checkpoint in its way up the hierarchy. 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-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-chain message exchanges is depicted in figure 10.
- 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-chain message list of trees, this data is updated by every new checkpoint in its way up the hierarchy.

Algorithm 3 Checkpoint data structure

```
1: Struct Checkpoints contains
2:     checkpointData                          ▷ Struct including the checkpoint data
3:     signature                               ▷ Signature for the checkpoint
4:     Struct CheckpointData contains
5:         lastBlockCid                        ▷ Cid of the latest block being committed by the checkpoint
6:         height                              ▷ Height of the chain for the last block included in the checkpoint
7:         prevCheckpoint                      ▷ Cid of the previous checkpoint generated in the chain
8:         CrossMsgs                           ▷ Tree of CrossMsgMeta being propagated
9:         checkCids                           ▷ Tree of checkpoints from child chains
10:
11:
```

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-chain message. Miners interact with their miner actor in their subnet, which triggers the corresponding cross-chain 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 7 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. Storage and retrieval markets are extremely interwined in the current implementation of Filecoin. By modularizing all storage-related functionalities (sector bookkeeping, miner operation, power actors, retrieval deals, payment channels, etc.) we open the door to the ability to host each of these modules in a different subnet. Thus, specific subnets can be spawned to handle payment channels, retrieval deals, etc. distributing the load, spawning specialized subnets for each of these components, and consequently increasing the performance and flexibility of the Filecoin protocol.

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 *ReleaseStake* 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 stake 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 stake of the subnet to be below $minStake_{subnet}$, 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 stake 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 stake to ensure a stake larger than $minStake_{subnet}$ in the subnet.

Miners of a subnet may choose to kill a subnet by sending a message to the *Kill* function of the SA. The

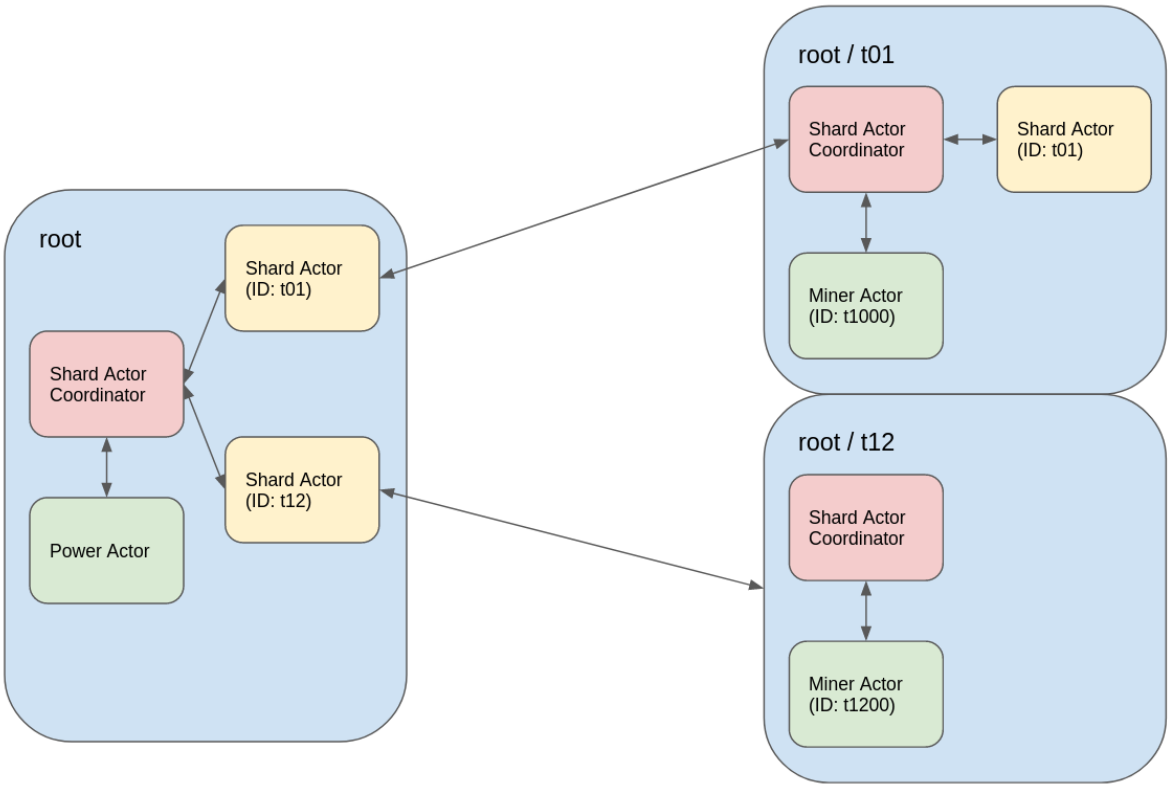

Figure 7: Power management architecture in hierarchical consensus

specific policy in order to be able to kill a subnet is up to the subnet founders. It may require a majority of the staked miners to send the kill message; or even that all miners agree on killing the subnet. Once the killing policy is fulfilled, a new message is sent to the *Kill* function of the SCA to release all the stake for the subnet and return it to its owners. This function also returns outstanding balances from users to their addresses in the parent 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 system in the next checkpoint of the parent chain, making effective the removal of the subnet.

TODO: Elaborate on state migrations when a subnet is killed
TODO: How do we handle killing a subnet when its state still holds user funds.

2.7 Cross-chain transactions and execution

In order for users to be able to have funds available in a subnet, they need to perform a transaction to fund an owned address in the subnet. This can be done through a cross-chain transaction. The propagation of a cross-chain transaction may slightly differ according to the location of the subnets in the hierarchy.

2.7.1 Stage I: Asymmetric Propagation

In the first stages of the implementation of *hierarchical consensus*, transaction actors and checkpoints orchestrate the propagation of cross-chain transactions between the different subnets involved. According to the specific location of the subnets, and the path that the propagation of the transaction needs to traverse, the commitment of the cross-chain transaction may differ.

- *Top-down transactions* are cross-chain 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, $root/t01$. 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 token that want to be transferred, or by calling *CrossMsg* in SCA with a message whose destination is one of the childs of $root/t01$. When a new top-down transaction is triggered, the SCA in the parent assigns a unique and incremental nonce to each new top-down transaction and stores it in the SCA state. This also triggers a state change in the SCA of the parent chain ($root/t01/$) freezing in the actor the funds injected to the subnet the through the top-down message. This funds are frozen until any down-top transaction releases it back to the parent. Thus, the SCA in $root/t01/$ keep track of the circulating supply in the subnet.

  Blocks validated by the consensus algorithms of subnets not only include internal messages initiated in the subnet, but also new cross-msgs directed to the subnet. Accordingly, miners proposing new blocks make a request to a cross-msg pool to check if there are unverified messages cross-msgs. The cross-msg pool checks the value of *AppliedTopDownNonce* in the SCA of the child subnet, $/root/t01/t12$, to get the latest nonce of a cross-msg executed in the child subnet (k). With this, the cross-msg 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-msg pool are included in the next block proposal.

  When a new block including top-down cross-msgs is verified in the subnet consensus, the top-down messages are committed, and every node receiving the new block executes the cross-msgs triggering the corresponding state changes and fund minting in the subnet by implicitly calling the *ApplyMsg* function provided by the SCA in their nodes. This function also updates *AppliedTopDownNonce* to the highest nonce of the top-down cross-msg applied plus one. Thus, the commitment of a simple *Fund* top-down message to inject new funds to an address of the subnet from the parent is translated in 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-msg is committed in the subnet (all of this is peformed by the call to SCA's

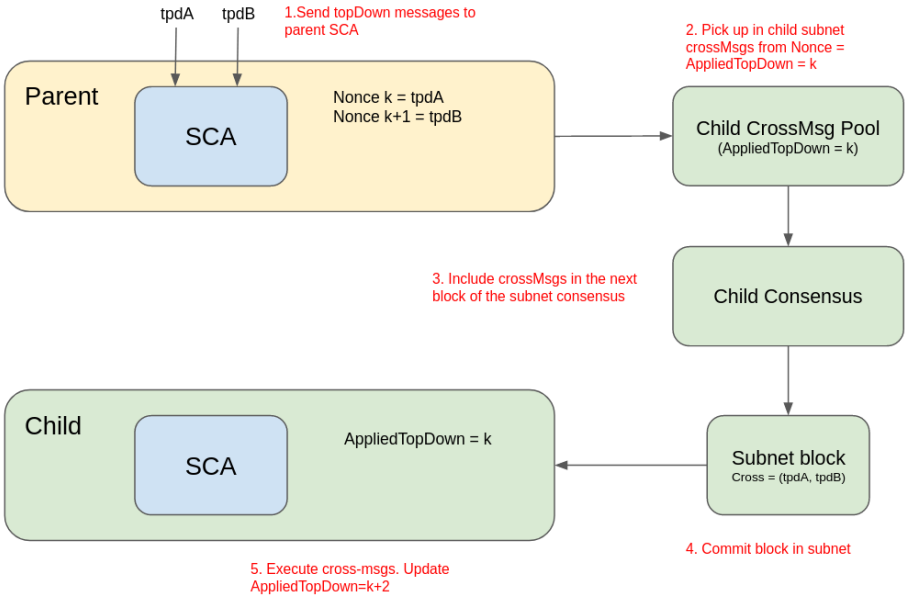

Figure 8: Commitment of top-down cross-msgs

ApplyMsg). An illustration of how top-down transactions are handled by the hierarchical consensus is depicted in Figure 8.

The execution of *ApplyMsg* expects top-down cross-msgs to be applied in order. If one of the messages for a specific nonce can't be applied and keeps failing when trying to be applied in the subnet, the subnet consensus could be stalled. This represents an attack vector to DDoS attacks. To prevent this from happening, if a cross-msg can't be applied by ApplyMsg, it will be 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. *TODO: We will need a way to inform back the parent that the message execution wasn't successful to unfreeze the funds and revert the corresponding state changes triggered by this message.*

- *Bottom-up transactions* are cross-chain 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 sending a message to the *CrossMsg* function of the SCA in the source chain ($/root/t01/t12$) with the amount that wants to be sent, or a *Release* message in the SCA specifying the amount of tokens that want to be sent from the subnet back to a parent address. Sending these transactions to SCA triggers the burning of the funds being sent/released in the subnet, and appends the message in the corresponding *CrossMsgMeta* storing cross-msgs with source the child subnet, $/root/t01/t12$, and destination the parent subnet if it is a *Release* message, or the corresponding ID subnet for *CrossMsgs*. The subnet SCA also 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 resolution requests from peers in other subnets for checkpoint, cross-message and cross-msg meta.

  The commitment of this message triggers the burn of these funds in the source chain (by performing

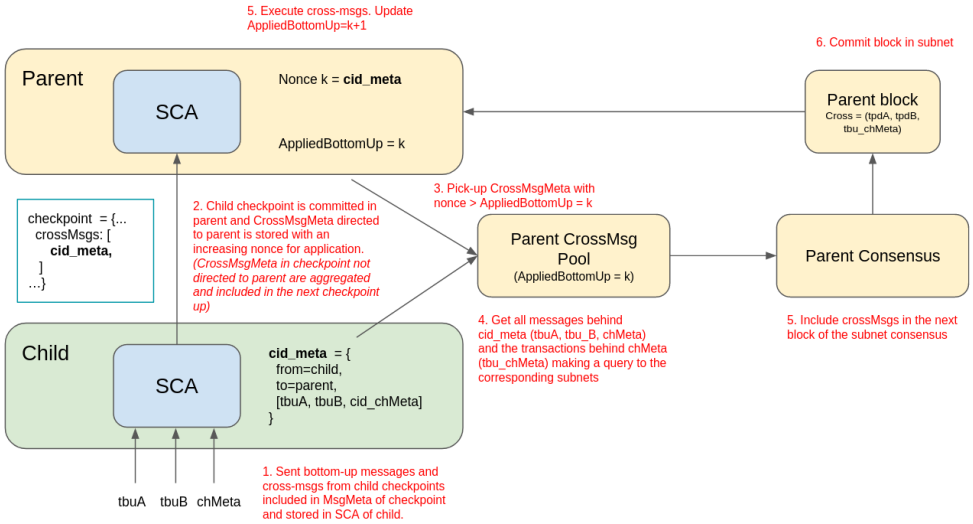

Figure 9: Commitment of bottom-up cross-msgs

a transaction to the burning address in the subnet). This transaction is buffered and included as part of the cross-transaction list of trees in the next checkpoint of the subnet chain. When the source chain commits this checkpoint in the parent chain $root/i/$, all the funds being burned and leaving a subnet are tracked in order to release frozen funds from the SA for the cross-chain transaction, and update the circulating supply in the subnet. The commitment of the checkpoint in the SCA of $/root/t01$ inspect the field of the checkpoint including the cross-chain transactions, it collects the transactions directed to itself, and automatically triggers the transaction from the frozen funds to the destination address of the destination subnet specified in the cross-chain transaction.

- *Path transactions* are cross-chain 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 *xChainTransaction* 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 as illustrated in figure 10. As the checkpoint moves up in the hierarchy, funds are conveniently released and burned in each of the subnets, 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-chain 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 10 and 11 illustrates the checkpoint propagation and account updates as different cross-chain 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 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-chain message 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. If we look at the example in figure 10, the checkpoint from subnet E propagated to its parent B, doesn't include every single cross-chain message, 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-chain transactions that are not directed to her subnet, it recursively aggregates in their checkpoint updating the source and the destination, propagating $cid_{C-B} = cid(cid_{C-F}, cidC - E)$ including the CIDs for all the cross-chain transactions directed to C. In this way, we minimize the amount of information that needs to be propagated to the top of the hierarchy.

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

- a *push* approach, where as the checkpoints 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; or,
- a *pull* approach where upon a subnet receives a checkpoint with cross-chain transactions directed to it, miners publish a request in the source subnet pubsub topic to request the DAG (list of messages) for a specific CID found in the cross-chain list of trees.

Coming back to our path transaction example, when C sees that a checkpoint is committed to *root* including pending transactions to its subnet, it gets its cross-chain transaction tree and start reading every branch to fetch and apply every cross-chain transaction sent to it. Thus, when inspecting its tree, it sees $cid_{C-B} = cid(cid_{C-F}, cidC - E)$. To get and apply the messages for this branch, if the push approach was used, C only needs to fetch $cid_{C-B} = cid(cid_{C-F}, cidC - E)$ locally, and recursively fetch cid_{C-F}, and cid_{C-F} until it gets all the messages and can apply them. If cid_{C-B} is not found locally, C needs to fetch cid_{C-B}, cid_{C-F}, and cid_{C-F} from their corresponding subnet.

Generality of the approach beyond payments. The examples above illustrated cross-chain 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 and trigger the required state changes. Handling arbitrary message that trigger fund exchanges between subnet may be a bit more complex, as it may require partial execution as the message propagates to determine the funds that needs to be burnt and released in each subnet. However, only the subnets being traversed by the checkpoint need to perform this execution, and we can come up with ways to include as part of the tree a proof of the funds that need to be mobilize preventing from having to execute the state change in each step.

2.7.2 Discussion on data availability and efficiency

An issue that arises with this checkpoint-based approach for cross-chain message propagation is the availability of cross-chain messages. Checkpoints only include commitments that aggregate the list of messages being propagated from one subnet to another. When the destination subnet receives a new checkpoint with messages directed to it, it is only provided with the CID of the messages 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, and it needs to trust that nodes in the subnet the checkpoint belongs to will provide the subnet with the message. Intuitively, the node that triggered the cross-chain 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.

In the meantime, the two ways in which a subnet may access the list of messages behind a CID are:

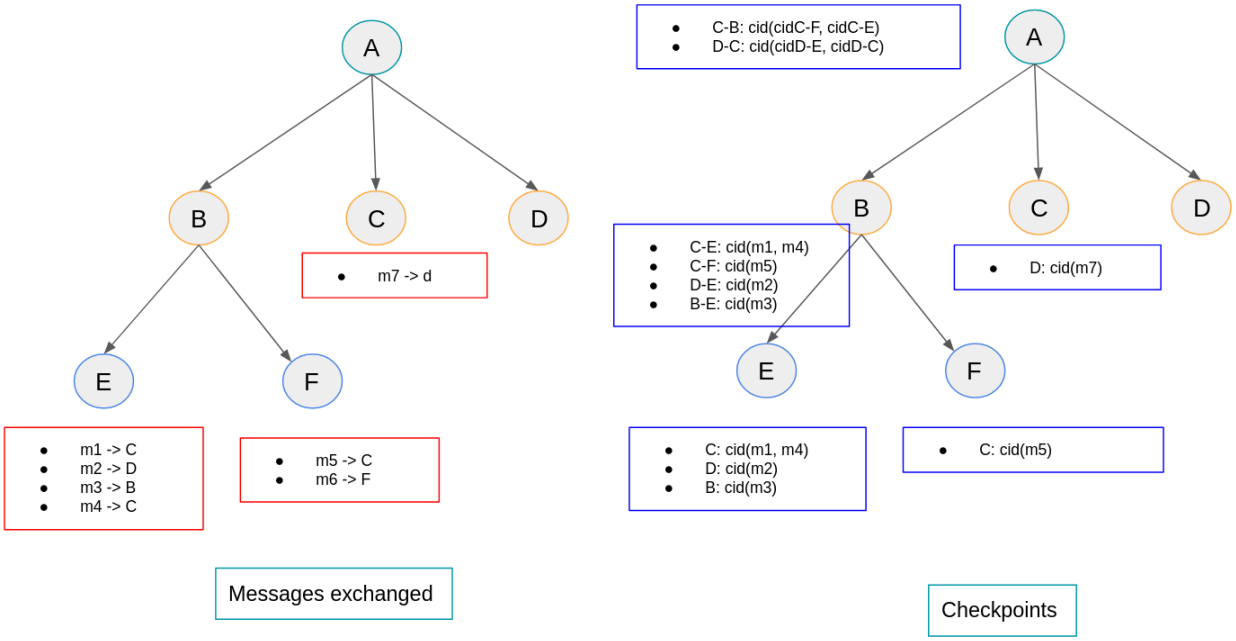

Figure 10: Transaction propagation through checkpoints

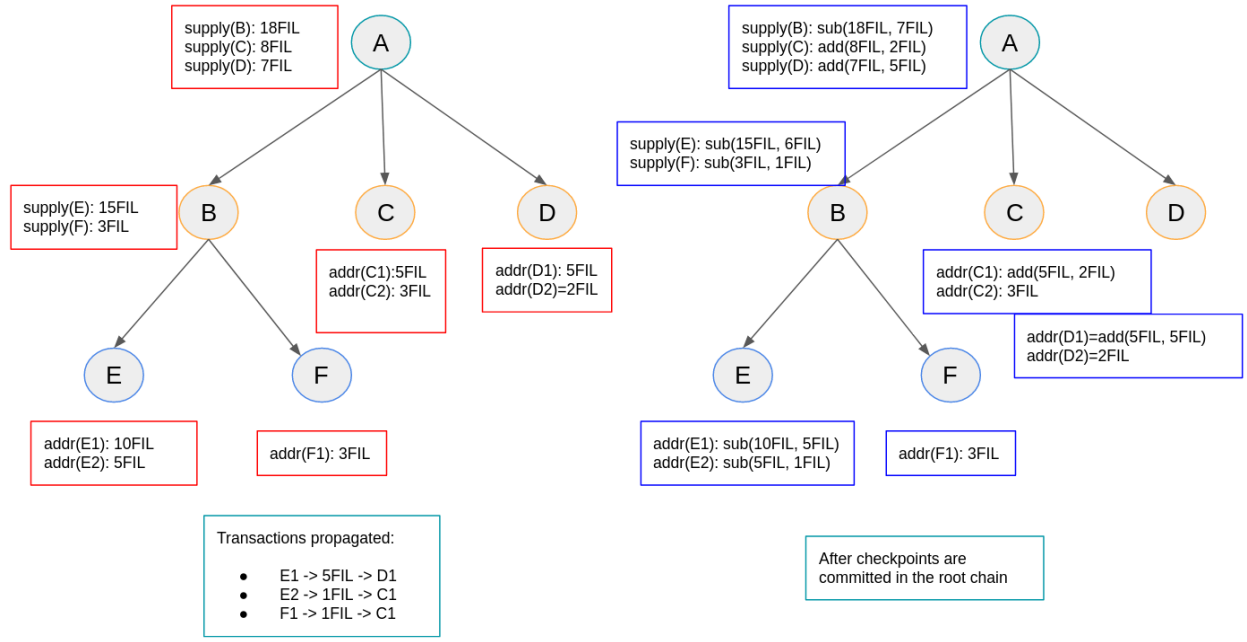

Figure 11: Cross-chain FIL transaction balance updates

- Push approach: Every time a node sees that a new subnet has been committed to the subnet, it publishes the list of message for that CID in all the subnets that require the CID (i.e. the parents from the root of the destination chain that need to trigger their fund function, and the destination subnet in path transactions. In top-down transaction, only the parents of the destination subnet need this message. This scheme is more efficient and should be the default.
- Pull approach: If the push message for a checkpoint is missed, or additional validations need to be made, peers can publish a request for the list of messages behind the CID to the subnet that generated the checkpoint.

Both approaches have its drawbacks. The push approach is faster but introduces a communication overhead due to the number of messages being exchanged; while the pull approach is slower (it requires a full RTT to recover the data), and has the data availability problem, as if the checkpoints are removed in the subnet chain, or the chain is killed, there is no way of recovering what was behind the CID.

2.7.3 Stage II: Lurk/ ZK cross-chain certificates/proofs?

Work in progress...

2.8 Fraud Proofs

3 System Evaluation

4 Incentives Model?