Hierarchical Consensus: A Horizontal Scaling Framework for Blockchains

Alfonso de la Rocha
Protocol Labs
alfonso@protocol.ai

Lefteris Kokoris-Kogias
IST Austria
ekokoris@ist.ac.at

Jorge M. Soares
Protocol Labs
jorge@protocol.ai

Marko Vukolić
Protocol Labs
marko@protocol.ai

***Abstract*—We present the Filecoin Hierarchical Consensus framework, which aims to overcome the throughput challenges of blockchain consensus by horizontally scaling the network. Unlike traditional sharding designs, based on partitioning the state of the network, our solution centers on the concept of subnets –which are organized hierarchically– and can be spawned on-demand to manage new state. Child subnets are firewalled from parent subnets, have their own specific policies, and run a different consensus algorithm, increasing the network capacity and enabling new applications. Moreover, they benefit from the security of parent subnets by periodically checkpointing state. In this paper, we introduce the overall system architecture, our detailed designs for cross-net transaction handling, and the open questions that we are still exploring.**

***Index Terms*—blockchain, consensus, distributed systems, P2P, scalability, sharding**

I. Introduction

Consensus, or establishing total order across transactions, poses a major scalability bottleneck in blockchain networks [1]. This is particularly the case when all nodes (often called *validators* or *miners*) are required to process all transactions. Regardless of the specific consensus protocol implementation used, this makes blockchains unable to increase their performance by adding more participants (scale-out).

In traditional distributed computing, 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 operated by its own set of nodes, which keep a subset of the state and are responsible for processing a part of the transactions sent to the system. In existing sharded designs [2], [3], the system often acts as a distributed controller that assigns miners to different shards and attempts to load-balance the state evenly across shards.

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 diluting security when compared to the original single-chain (single-shard) solution. In both proof-of-work and proof-of-stake (PoS) blockchains, sharding may lead to the ability of the attacker to compromise a single shard with only a fraction of the mining power, potentially compromising the system as a whole. Such attacks are often referred to as *1% attacks* [2], [4]. To circumvent them, sharding systems need to periodically reassign miners to shards in an unpredictable way, so as to cope with a semi-dynamic adversary [5], [6]. We believe that this traditional approach to scaling, which considers the system as a monolith, is not suitable for decentralized blockchains due to their complexity and the fact that sharded systems reshuffle state without the consent of its owners.

In our efforts to scale the Filecoin network [7], [8], we depart from the traditional sharding approach to build *hierarchical consensus*[1]. In hierarchical consensus, instead of algorithmically assigning node membership and load balancing the distribution of the state, we follow an approach where users and miners are grouped into *subnets* in which they can freely partake. Users can spawn new child subnets from the one they are operating in accordance with their needs, and become miners there if they fulfill all the requirements set by the protocol. Each subnet can run its own independent consensus algorithm and set its own security and performance guarantees. Subnets in the system are organized hierarchically, each having one parent subnet and any number of child subnets — except for the root subnet (called *root network* or *rootnet*), which has no parent and is the initial anchor of trust. To circumvent the 1% attacks pertinent to traditional sharding, subnets in hierarchical consensus are firewalled [9], in the sense that a security violation in a given subnet is limited, in effect, to that particular subnet and its children, with bounded economic impact on its ancestors. This bounded impact of an attack is, at most, the circulating supply of the parent token in the child subnet. Moreover, ancestor subnets help secure their descendant subnets through *checkpointing*—which helps alleviate attacks on a child subnet, such as long-range and related attacks in the case of a PoS-based subnet [10]. At a high level, hierarchical consensus allows for incremental, on-demand, blockchain scaling and simplifies deployment of new use cases with clearly isolated security domains that provide flexibility for varied use cases. Our design further supports ordinary (e.g., token payment) and atomic transactions across subnets.

This paper is organized as follows: Section II provides a high-level overview of the system and its specifications; Section III describes the life cycle of a subnet and its key operations; Section IV explains the semantics for cross-subnet transactions; finally, Section V surveys the related work and

[1] https://github.com/filecoin-project/eudico/

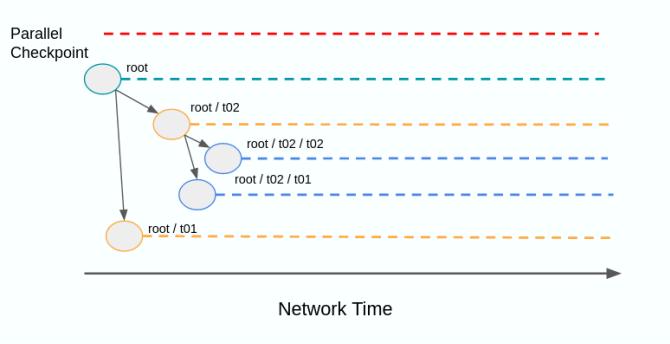

Fig. 1. **System Overview.** Subnets are spawned from the root chain, thereby building a hierarchy of independent networks.

Section VI outlines conclusions and future directions.

II. System Overview

Fig. 1 depicts a high-level overview of a hierarchical consensus system. The system starts with a *rootnet* which, at first, keeps the entire state and processes all the transaction in the system (like present-day Filecoin). At one point, a subset of users requiring lower latency or higher throughput can spawn a new subnet to accommodate their performance requirements. This subnet instantiates a new *chain* with its own state, independent from the root chain, replicated among the subset of participants of the overall system who are members of the subnet. From this point on, the new subnet processes transactions involving the state in the subnet independently from the root chain. Further subnets can then be spawned from any point in the hierarchy.

Subnets are able to interact with the state of other subnets (and that of the rootnet) through *cross-subnet* (or, simply, *cross-net*) messages. Full nodes in a given subnet have trusted access to the state of its parent subnet. We implement this by having nodes sync the chain of its parent (i.e., child subnet nodes also run full nodes on the parent subnet). Miners on parent subnets do not need to sync with child subnets' chains.

In hierarchical consensus, it may be hard to enforce an honest majority of mining power in every subnet, which can result in the subnet chain being compromised or attacked. The system provides a *firewall* security property, introduced in [9]; this guarantees that, for token exchanges, the impact of a child subnet being compromised is limited to, at most, its circulating supply of the token, determined by the (positive) balance between cross-net transactions entering the subnet and cross-net transactions leaving the subnet. Addresses in a subnet are funded through cross-net transactions that inject tokens into the subnet. In order for users to be able to spawn a new subnet, they need to deposit an initial collateral into the new subnet's parent. This collateral offers a minimum level of trust to new users injecting tokens to the subnet and can be slashed in case of misbehavior by subnet validators.

Miners in subnets are rewarded with fees for the transactions executed in the subnet. Subnets can run a consensus algorithm of their choosing to validate blocks and can determine the consensus proofs they want to include for light clients (i.e., nodes that do not synchronize and retain a full copy of the blockchain and thus do not verify all transactions). Subnets periodically commit a proof of their state in their parent through checkpoints. These proofs are propagated to the top of the hierarchy, making them accessible to any member of the system. They should include enough information that any client receiving it is able to verify the correctness of the subnet consensus. Subnets are free to choose a proof scheme that suits their consensus best [e.g., multi-signature, threshold signature or zero-knowledge (ZK) proofs]. 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. Checkpoints are also used to propagate to other subnets in the hierarchy the information pertaining to cross-net messages.

III. Lifecycle of a subnet

A. Spawning and joining a subnet

Creating a new subnet instantiates a new independent state with all its subnet-specific requirements to operate independently. This includes, in particular: a new attack-resilient pubsub [11] topic that peers use as the transport layer to exchange chain-specific messages, a new mempool instance, a new instance of the Virtual Machine (VM), as well as any other additional module required by the consensus that the subnet is running (system actors, i.e., smart contracts in Filecoin terminology; mining power resources; etc. [7]).

To spawn a new subnet, peers need to deploy a new *Subnet Actor* (SA) that implements 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 a new subnet to interact with the rest of the hierarchy, it needs to be registered in the *Subnet Coordinator Actor* (SCA) of the parent chain. The SCA is a system actor 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. And, as SAs are user-defined and untrusted, it also enforces security assumptions, fund management, and the cryptoeconomics of hierarchical consensus. Subnets are identified with a unique ID that is inferred deterministically from the ID of its ancestor and from the ID of the SA that governs its operation. This deterministic naming enables the discovery of and interaction with subnets from any other point in the hierarchy without the need of a discovery service: peers need only send a message to the subnet's specific pubsub topic, identified with the subnet's ID.

B. Checkpointing protocol

Checkpoints are used to anchor a subnet's security to that of its parent chain, as well as to propagate information from a child chain to other subnets in the system. Checkpoints for a subnet can be verified at any point using the state of the subnet chain which can then be used to generate equivocation

proofs (or so-called *fraud proofs*) which, in turn, can be used for penalizing misbehaving entities ("slashing").

Subnet miners need to provide a minimum collateral, $minCollateral_{subnet}$, in their parent's SCA to register the subnet to the hierarchy and be able to interact with other subnets. This collateral is frozen through the lifetime of the subnet and does not become part of its circulating supply. These collateral funds are the ones slashed in the face of a valid fraud proof. If the subnet's collateral drops below $minCollateral_{subnet}$, the subnet enters an *inactive* state, and it can no longer interact with the rest of the hierarchy. To recover its *active* state, users of the subnet need to put up additional collateral.

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. Different signature schemes may be used here, including multi-signatures or threshold signatures among subnet miners.

As an example, consider a checkpoint for subnet $/root/A/B$. Periodically (in terms of subnet block time), miners access the checkpoint template that needs to be signed and populated by calling the SCA in $/root/A/B$. Once signed, checkpoints from $/root/A/B$ are committed to the SA B of the subnet chain $/root/A$. After performing the corresponding checks, this actor triggers a message function to the SCA in $/root/A$, which is responsible for aggregating the checkpoint from $/root/A/B$ with those of other children of $/root/A$ and for generating a new checkpoint for $/root/A$ that is then propagated to its parent chain, $/root$. As checkpoints flow up the chain, the SCA of each chain picks up these checkpoints and inspects them to propagate potential state changes (like balance updates in a monetary transaction) triggered by messages included in the cross-net messages (for brevity, we call these simply *cross-msgs*) that have the SCA's subnet as a destination subnet (see Fig. 2).

Checkpoints are always identified though their Content Identifier (CID)[2], a unique identifier inferred from the checkpoint's hash, and include the corresponding signature from miners in the subnet chain (this can be the signature of an individual miner, a multi-signature, or a threshold signature, depending on the SA policy). Checkpoints include the following data: $< s, proof, prev, children, crossMeta >$ where: (i) s is the source subnet of the checkpoint; (ii) $proof$ is the content identifier CID (roughly corresponding to a hash) of the latest block from the subnet chain being committed in the checkpoint; (iii) $prev$ is a pointer to the CID of the previous checkpoint of the subnet; (iv) $children = (from, cid)$ is a tree which includes the subnet ID and the corresponding checkpoint CID for every child chain; and (v) $crossMeta = (from, to, nonce, msgsCid)$ is a tree of cross-msg metadata including every cross-msg being propagated

[2]https://github.com/multiformats/cid

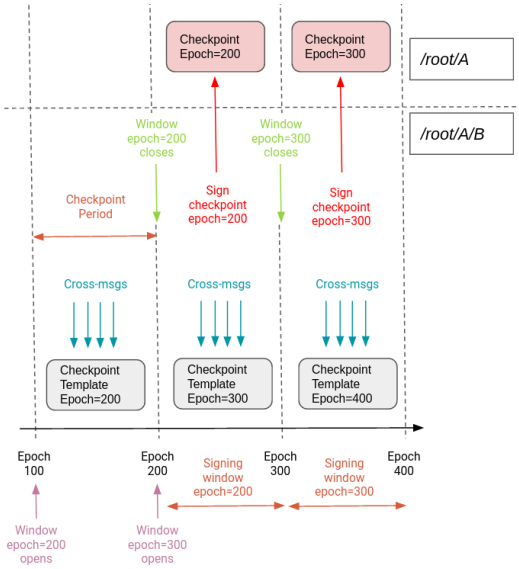

Fig. 2. **Checkpoint template population.** The checkpoint period in the SA determines the window during which cross-msgs are accepted in the current checkpoint. Upon reaching the end of the period, new cross-msgs begin populating the next checkpoint and a signature window is opened for the previous one.

upwards by the subnet and its children. We call this cross-msg metadata *CrossMsgMeta*; it includes information about: the source subnet, $from$, of the group of messages; the destination subnet, to; the cross-msg nonce, $nonce$; and the CID (message digest) of the group of messages, $msgsCid$. This tree of *CrossMsgMeta* gets updated with every new checkpoint on its way up the hierarchy.

C. Leaving and killing a subnet

Members of a subnet can leave the subnet at any point by sending a message to the subnet's SA in the parent chain. If the miner fulfils the requirements to leave the subnet defined in the subnet's SA when it was deployed, a message to the SCA is triggered by the SA to release the miner's collateral. If a miner leaving the subnet brings the collateral of the subnet below $minCollateral_{subnet}$, the subnet gets in an *inactive* state. Miners of a subnet can also kill a subnet by sending a message to the SA. Similar to the previous situation, the SA sends a message to the SCA to release all the collateral for the subnet if all the requirements to kill the subnet are fulfilled.

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 the threshold enforced by the hierarchical consensus to allow cross-net communication, users no longer have a way to get their funds and state out of the subnet. To prevent this from happening, the SCA includes a *save* function that allows any participant in the subnet to persist the state. 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 of a specific part of the state that they want to be migrated back to the parent.

IV. Cross-net transactions and execution

A. Cross-net messages

Users in a subnet interact with other subnets through cross-net transactions. The propagation of a cross-net transaction may slightly differ depending on the location of subnets in the hierarchy, e.g., if moving up or down the hierarchy. In particular, we distinguish: *top-down*, *bottom-up*, and *path* messages.

Top-down messages are cross-msgs directed towards a subnet that is lower in the hierarchy. When a new top-down transaction is triggered, the SCA of the source subnet (parent) increments a nonce that is unique to the top-down transaction directed to each of its child subnets (destination) and stores it in the SCA state. These nonces determine the total order of arrival of cross-msgs to the subnet; without them, different consensus nodes could execute different orderings, leading to nondeterminism. The commitment of a top-down transaction in the SCA also triggers the freezing, in the parent, of the funds included in the message, and updates the circulating supply in the destination subnet. These funds will be frozen until a bottom-up transaction releases them back to the parent. Child subnet miners always sync with their parent chains (i.e., track their latest state) to stay informed of updates to the state of the SCA and SA in the parent, and so are immediately notified when there are new unverified top-down messages directed to them.

Bottom-up messages are cross-msgs directed towards a subnet that is higher in the hierarchy but shares the same prefix. Bottom-up messages are propagated in checkpoints. At every checkpoint period, the SCA collects all *CrossMsgMeta* from bottom-up transactions originated in the subnet and all the *CrossMsgMeta* received from the subnet's child subnets, and includes them in the next checkpoint to be propagated up the hierarchy. Every message leaving the subnet triggers the burn (in the child) and release (in the parent) of the funds included in the message, updating its circulating supply.

When the checkpoint from the child subnet is committed in the parent chain, the SCA of the parent chain inspects all *CrossMsgMeta* in the checkpoint and collects the ones directed to it. Bottom-up messages targeting other subnets are propagated farther up the hierarchy in the next checkpoint, while bottom-up *CrossMsgMeta* targeting the current subnet are assigned an increasing nonce for posterior validation and application by the subnet's consensus algorithm.

Path messages. Every message routed in the hierarchy can be seen as a combination of top-down and bottom-up transactions. Path messages are cross-net messages in which the source and destination subnets are not in the same branch. These are propagated through bottom-up messages (i.e., *CrossMsgMeta* in checkpoints) up to the common parent ($root$, in the worst case), and through top-down messages from there to the destination. As checkpoints move up in the hierarchy, funds are conveniently released and burned in each of the subnets as cross-msgs flow. The opposite happens for top-down messages, where flowing messages trigger the minting of new funds in destination subnets. This updates the circulating supply of subnets as messages flow through the hierarchy.

According to the route that 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 slow. To accelerate the process, each SA in the path can send a direct message to the destination, 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) to indicate a pending payment or even as tentative information to start operating as if these funds were already settled and available in the subnet.

B. Cross-msg pool

Nodes in subnets keep two types of message pools: an internal pool to track unverified messages originating in and targeting the subnet and a cross-msg pool that listens to unverified cross-msgs directed at (or traversing) the subnet. To verify and execute cross-msgs in a subnet, they need to be included by the consensus algorithm in a subnet block.

Miners' cross-msg pools collect unverified cross-net messages by syncing with state changes in the SCA of the parent subnet. Whenever the SCA in the parent receives a new top-down message or collects a new bottom-up *CrossMsgMeta* from a child checkpoint, the cross-msg pool is notified. Top-down messages can be proposed to and applied directly in the subnet. For bottom-up messages, the cross-msg pool only has the CID of the *CrossMsgMeta* that points to the cross-msgs to be applied and, therefore, needs to make a request to the content resolution protocol (Section IV-C) to retrieve the raw messages, so that it may propose and apply them. Blocks in subnets include both messages originated within the subnet and cross-msgs targeting (or traversing) the subnet. When a new block including top-down cross-msgs is verified in the subnet consensus, the cross-msgs are committed, and every node receiving the new block executes the cross-msgs to trigger the corresponding state changes and fund exchanges in the subnet (Fig. 3).

Cross-msgs have to go through several checks before they are stored in the SCA and provided to the subnet consensus through the cross-msg pool, but the application of these messages may still fail. This is especially true for arbitrary messages. If the message for a specific nonce cannot be applied and keeps failing when trying to be applied in the respective SCA, the subnet consensus could stall. This represents a vector for Distributed Denial of Service (DDoS) attacks. To prevent this, a cross-msg that cannot be applied in a subnet triggers a new cross-msg with the subnet where the execution of the message failed as source and the original source of the message as destination. This message is used to revert every intermediate state change that may have been triggered in the original cross-msg route through the hierarchy.

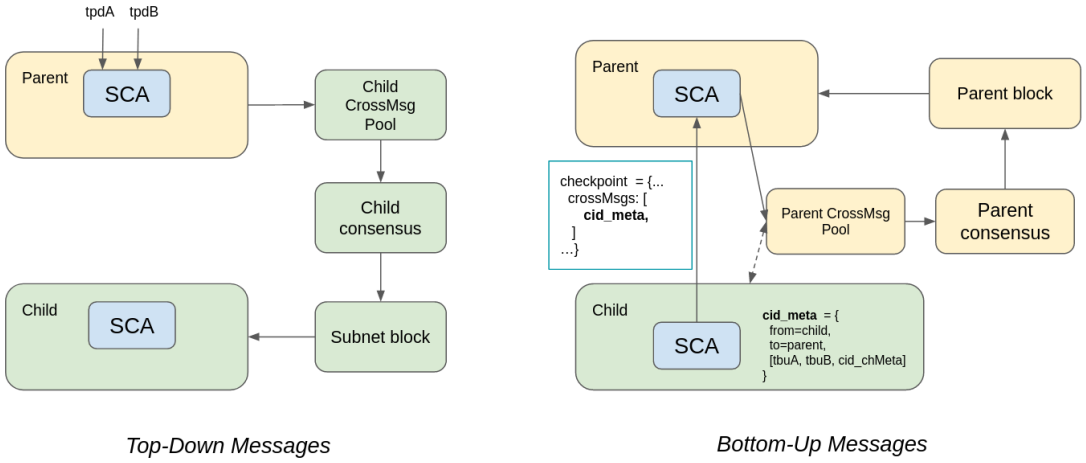

Fig. 3. **Commitment of top-down cross-msgs (left).** When a top-down message is committed by a subnet's parent, it is assigned the next nonce and added to the list of unverified top-down messages for the destination subnet. The cross-msg pool of nodes in the child listens to changes in the parent's state and pulls any new unverified cross-msg to the subnet. These messages are proposed inside the next block of the consensus and are ordered and verified as any other messages sent within the subnet. When the block including cross-msgs is committed, they are applied and the corresponding state changes are triggered. **Commitment of bottom-up cross-msgs (right).** Bottom-up messages are propagated through checkpoints. The child subnet aggregates the cross-msgs from within its own subnet and those propagated from its children inside a meta-tree, and includes this meta-tree in the next checkpoint. When the parent receives the checkpoint, it checks what meta-trees are directed to itself and which need to be propagated further. Meta-trees to other subnets are included in the next checkpoint of the parent, while those targeting the current subnet are assigned the next unique nonce and stored for commitment. When the next unverified meta-tree is picked up, the corresponding messages are fetched, ordered, and proposed in the next block, and consequently committed and applied.

C. Subnet cross-msg resolution protocol

For scalability reasons, when the destination subnet receives a new checkpoint with messages, 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 needs to fetch the payload of messages behind that CID as illustrated in Fig. 4. The subnet SCA where the bottom-up message is generated keeps a registry with all CIDs for *CrossMsgMetas* propagated (i.e., a content-addressable key-value store), which is used to fulfill content resolution requests. The content resolution protocol implements two approaches to resolve content:

(i) A *push* approach, where, as the checkpoints and *CrossMsgMetas* move up the hierarchy, miners publish to the pubsub topic of the corresponding subnet the whole Distributed Acyclic Graph (DAG) belonging to the CID, including all the messages targeting that subnet. Content is pushed to other subnets by publishing in the destination subnet's pubsub topic a *push* message specifying the type of content being pushed along with its CID. Peers may choose to directly push the corresponding messages behind a *CrossMsgMeta* to the destination address once a checkpoint has been signed. When peers in the subnet come across these messages, they may choose to pick them up and cache/store them locally or discard them (in which case, they will need to explicitly resolve the content when required).

(ii) A *pull* approach, where, upon a destination subnet receiving a checkpoint with cross-msgs directed to it, miners' cross-msg pools publish a *pull* message in the source subnet's pubsub topic to resolve the cross-msgs for a specific

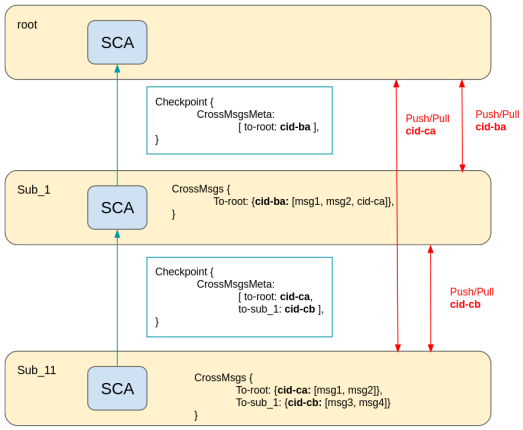

Fig. 4. **Content resolution protocol for cross-msgs.** Whenever a subnet submits a new checkpoint to its parent, it pushes the messages behind the CIDs included in the meta-trees of the checkpoint. If a subnet comes across a CID in the meta-tree that it cannot resolve locally (either because it missed 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 originating subnet.

CID found in the tree of cross-msg meta. These requests are answered 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 gives every cross-msg pool a new opportunity to store or cache the content behind a CID even if they do not yet need it.

D. Generality of the approach beyond payments: Atomic executions

An issue arises when state changes need to be atomic and impact the state of different subnets [12]. A simple example of this is the atomic swap of two assets hosted in different subnets. The state change in the subnets needs to be atomic, and it requires from state that lives in both subnets. To handle these atomic transactions, users in the subnets can choose any subnet in the hierarchy in which they both have a 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 therefore leveraging shared trust.

A cross-net atomic execution takes tuples of input states and returns tuples of outputs states, which may belong to different subnets, but should appear as a single transaction in which all input/output states belong to the same subnet. Our atomic execution protocol has the following properties:

(i) *Timeliness*: The protocol eventually completes by committing or aborting.

(ii) *Atomicity*: If all involved subnets commit and no subnet aborts beforehand, the protocol commits and all subnets involved have the output state available as part of their subnet state. Otherwise, the protocol aborts and all subnets revert to their initial state.

(iii) *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 need to ensure the *consistency* of the state in each subnet, i.e., that the output state of the atomic execution can be applied onto the original state (and history) of the subnet without conflicts.

Our atomic execution protocol consists of the following phases, which, combined, resemble a two-phase commit protocol with the SCA of the least common ancestor/parent serving as a coordinator:

Initialization: To signal the start of an atomic execution, the users interact off-chain to agree on the specific execution they want to perform and the input state it will involve. 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 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 users to be able to execute locally, they need to request the state locked in the other subnet. The CID of the input state is shared between the different users during the initialization stage, and is leveraged by each user to request from the other subnets the locked input states involved in the execution. Once every input state is received, each user runs 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. The 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 halfway, any user is allowed to abort the execution at any time and to unlock their state by sending an *ABORT* message to the SCA.

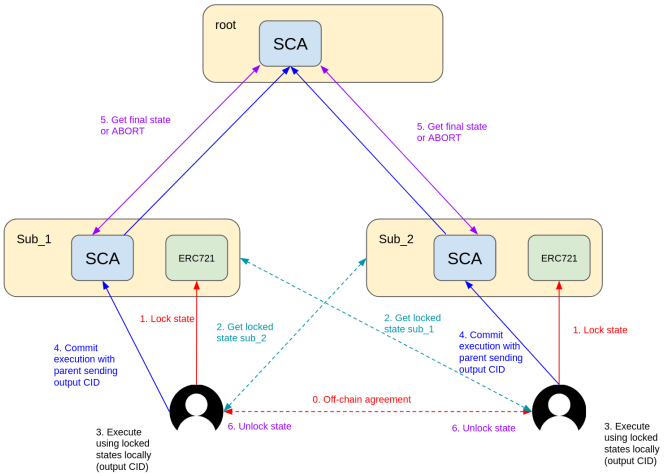

Fig. 5. **Atomic execution protocol**. The protocol starts with an off-chain agreement between the different users involved. All parties lock the state they want to use as input in the atomic execution and one notifies the SCA in the parent that they want to start an atomic execution. Once all input states have been locked, all users collect the pending inputs from other subnets to perform the atomic execution. Each user performs the execution off chain. Every user submits the output state of the execution in the SCA of the parent chain. The SCA waits for all the parties involved to submit the output state, and checks if they all match. If the state matches, the SCA marks the execution as 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.

Termination: All subnets involved in the protocol listen to events in the SCA of the execution subnet. When the SCA receives the commitment of all the computed output states, and if they all match, the execution is marked as successful, possible aborts are no longer taken into account, and subnets are notified, through a cross-net message, that it is safe to incorporate the output state and unlock the input state. If, instead, the SCA receives an *ABORT* message from some subnet before getting commitment from all subnets, it will mark the transaction as aborted and each subnet will be notified (through a cross-msg) that it may revert/unlock their input state without performing changes to the local state.

One open question when moving from fungible assets to general state is whether the firewall property can still hold. This generalized case is problematic since compromised subnets can send seemingly valid but actually corrupt input states to the other subnets involved in the atomic execution. Because subnets are only light clients of other subnets and rely on the security of their consensus, this can be hard to detect without an honest peer in the subnet raising the alert. As part of future work, we are exploring schemes that would

allow the detection of invalid states in the protocol.

V. Related Work

There are several concurrent approaches to the challenging problem of scaling blockchains [1]. We group them coarsely into four categories: sharding, payment/state channels, rollups and sidechains/subnets.

Sharding approaches (e.g., [2], [4]–[6], [12]–[15]) partition blockchain state processing across different groups of nodes, unlike our hierarchical consensus. Security and performance of sharding systems typically requires complex and periodic reassignment of nodes [2] and state across shards [3]. Sharding approaches have been in the focus of most academic work on horizontal scaling to date, they have been deployed in some blockchain systems (e.g., Zilliqa [16] and Near Protocol [17]), and are considered as candidates in others (e.g., Ethereum [18]).

Payment channels [19] are a scaling approach in which a payment is carried out privately among small groups of parties, off chain, with the on-chain communication being reserved only for setup and dispute resolution. Payment channels have been deployed in practice, with the most prominent example being the Bitcoin Lightning Network [20]. This approach was later generalized to general-state channels (e.g., [21]).

Rollups propose a tiered blockchain architecture where, in short, a top tier (L1) blockchain only orders transactions which are then retrieved from other tiers (given a data availability tier) for execution. Separate execution nodes (L2) post back results to L1 so that not everyone needs to execute. This approach is somewhat similar to the tiered approach pioneered by the Hyperledger Fabric permissioned blockchain [22]. Current early rollup implementations include: (i) optimistic rollups (e.g., [23]) in which only a subset of execution nodes execute publicly available transactions in the common case and where transactions are reexecuted on L1 in case of misbehavior of L2 nodes (similar to payment channels) and (ii) ZK rollups (e.g., [24], [25]) which have execution nodes cryptographically prove their execution correct while allowing for fast verification (e.g., [26]).

Our hierarchical consensus is inspired by the PoS sidechain design—to our knowledge first proposed in [9]. In general, sidechains allow a faster and less secure chain to benefit from the security of a more robust, slower chain by writing critical information (e.g., a checkpoint) to it. In our case, subnets are sidechains, hierarchically orchestrated, instantiating the sidechain approach of [9] in a specific and novel way, which allows for general-state cross-net atomic transactions. Furthermore, unlike [9], hierarchical consensus subnets can run any type of consensus and are not limited to PoS-based consensus algorithms, including on the rootnet.

A related approach to ours is that of Avalanche subnets [27], inspired by [28], which, unlike hierarchical consensus, support limited subnet hierarchies and do not consider cross-net semantics. Concretely, the 3 subnets of the Avalanche primary network (rootnet), are split into: an asset exchange chain (X-chain), a platform chain (P-chain), which allows for creation of further non-nested subnets, and a contract chain (C-chain), which supports smart-contracts. In contrast, hierarchical consensus has no such separation, and any subnet can be used for asset exchange, smart contracts, or creating further subnets, while supporting atomic cross-net semantics.

VI. Conclusions

This paper presented a hierarchical consensus framework that enables the horizontal scaling of blockchain systems. It does so by allowing the creation of subnets at any point in the hierarchy, each being able to run a different consensus protocol and set different policies.

Subnets can process internal transactions without going through the main chain, only submitting periodic checkpoints to their parent. They can also take part in cross-net transactions, with support for atomic execution. In addition to increasing chain capacity, our framework enables new use cases by providing highly-customized environments, largely free from the the constraints of the root chain.

While still a work in progress, we plan for this framework to be adopted by the Filecoin blockchain. A prototype implementation of the framework is already available, with ongoing work to integrate different consensus protocols, including Tendermint [29] and MirBFT [30].

References

[1] M. Vukolić, "The quest for scalable blockchain fabric: Proof-of-work vs. BFT replication," in *Int. Workshop on Open Problems in Network Security (iNetSec)*, 2015, pp. 112–125.

[2] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford, "Omniledger: A secure, scale-out, decentralized ledger via sharding," in *2018 IEEE Symp. Secur. Privacy (SP) 2018, Proc., 21–23 May 2018, San Francisco, CA, USA*. IEEE Computer Society, 2018, pp. 583–598. [Online]. Available: https://doi.org/10.1109/SP.2018.000-5

[3] M. Król, O. Ascigil, S. Rene, A. Sonnino, M. Al-Bassam, and E. Rivière, "Shard scheduler: object placement and migration in sharded account-based blockchains," 2021.

[4] L. Luu, V. Narayanan, C. Zheng, K. Baweja, S. Gilbert, and P. Saxena, "A secure sharding protocol for open blockchains," in *CCS '16: Proc. 2016 ACM SIGSAC Conf. Comp. Commun. Secur.*, ser. CCS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 17–30. [Online]. Available: https://doi.org/10.1145/2976749.2978389

[5] G. Yu, X. Wang, K. Yu, W. Ni, J. A. Zhang, and R. P. Liu, "Survey: Sharding in blockchains," *IEEE Access*, vol. 8, pp. 14 155–14 181, 2020.

[6] G. Avarikioti, E. Kokoris-Kogias, and R. Wattenhofer, "Divide and scale: Formalization of distributed ledger sharding protocols," *arXiv preprint arXiv:1910.10434*, 2019.

[7] Protocol Labs, "Filecoin: A decentralized storage network," https://filecoin.io/filecoin.pdf, 2017.

[8] "Filecoin," https://filecoin.io/, 2021.

[9] P. Gazi, A. Kiayias, and D. Zindros, "Proof-of-stake sidechains," in *2019 IEEE Symp. Secur. Privacy, (SP) 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 2019, pp. 139–156.

[10] S. Steinhoff, C. Stathakopoulou, M. Pavlovic, and M. Vukolić, "BMS: Secure decentralized reconfiguration for blockchain and BFT systems," 2021.

[11] D. Vyzovitis, Y. Napora, D. McCormick, D. Dias, and Y. Psaras, "Gossipsub: Attack-resilient message propagation in the Filecoin and ETH2.0 networks," *arXiv preprint arXiv:2007.02754*, 2020.

[12] E. Androulaki, C. Cachin, A. D. Caro, and E. Kokoris-Kogias, "Channels: Horizontal scaling and confidentiality on permissioned blockchains," in *Eur. Symp. Res. Comp. Sec.* Springer, 2018, pp. 111–131.

[13] J. Wang and H. Wang, "Monoxide: Scale out blockchains with asynchronous consensus zones," in *16th USENIX Symp. Networked Syst. Des. Implementation (NSDI 19)*. Boston, MA: USENIX Association, Feb. 2019, pp. 95–112. [Online]. Available: https://www.usenix.org/conference/nsdi19/presentation/wang-jiaping

[14] M. Zamani, M. Movahedi, and M. Raykova, "Rapidchain: Scaling blockchain via full sharding," in *Proc. 2018 ACM SIGSAC Conf. Comp. Commun. Secur.*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 931–948. [Online]. Available: https://doi.org/10.1145/3243734.3243853

[15] M. Al-Bassam, A. Sonnino, S. Bano, D. Hrycyszyn, and G. Danezis, "Chainspace: A sharded smart contracts platform," in *25th Annu. Netw. Distrib. Syst. Secur. Symp., NDSS 2018, San Diego, CA, USA, February 18–21, 2018*. The Internet Society, 2018. [Online]. Available: http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018_09-2_Al-Bassam_paper.pdf

[16] The ZILLIQA Team, "The ZILLIQA Technical Whitepaper," https://docs.zilliqa.com/whitepaper.pdf, 2017.

[17] A. Skidanov and I. Polosukhin, "Nightshade: Near protocol sharding design," https://near.org/downloads/Nightshade.pdf, 2019.

[18] Ethereum Wiki, "On sharding blockchains FAQs," https://eth.wiki/sharding/Sharding-FAQs, 2021.

[19] L. Gudgeon, P. Moreno-Sanchez, S. Roos, P. McCorry, and A. Gervais, "SoK: Layer-two blockchain protocols," in *Financial Cryptogr. Data Secur. - 24th Int. Conf., FC 2020, Kota Kinabalu, Malaysia, February 10-14, 2020 Revised Selected Papers*, ser. Lecture Notes in Computer Science, J. Bonneau and N. Heninger, Eds., vol. 12059. Springer, 2020, pp. 201–226. [Online]. Available: https://doi.org/10.1007/978-3-030-51280-4_12

[20] Joseph Poon and Thaddeus Dryja, "The Bitcoin Lightning Network," http://lightning.network/lightning-network-paper-DRAFT-0.5.pdf, 2016.

[21] S. Dziembowski, S. Faust, and K. Hostáková, "General state channel networks," ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 949–966. [Online]. Available: https://doi.org/10.1145/3243734.3243856

[22] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. D. Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolić, S. W. Cocco, and J. Yellick, "Hyperledger fabric: A distributed operating system for permissioned blockchains," in *Proc. 13th EuroSys Conf., EuroSys 2018, Porto, Portugal, April 23-26, 2018*, R. Oliveira, P. Felber, and Y. C. Hu, Eds. ACM, 2018, pp. 30:1–30:15. [Online]. Available: https://doi.org/10.1145/3190508.3190538

[23] H. Kalodner, S. Goldfeder, X. Chen, S. M. Weinberg, and E. W. Felten, "Arbitrum: Scalable, private smart contracts," in *27th USENIX Secur. Symp. (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 1353–1370. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/kalodner

[24] "Polygon Hermez," https://hermez.io/, 2021.

[25] "StarkWare," https://starkware.co/, 2021.

[26] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev, "Scalable, transparent, and post-quantum secure computational integrity," *IACR Cryptol. ePrint Arch.*, p. 46, 2018. [Online]. Available: http://eprint.iacr.org/2018/046

[27] Avalanche, "Subnet FAQ," https://docs.avax.network/build/tutorials/platform/subnets/subnet-faq/, 2021.

[28] A. E. Gencer, R. van Renesse, and E. G. Sirer, "Service-oriented sharding with Aspen," 2016.

[29] E. Buchman, J. Kwon, and Z. Milosevic, "The latest gossip on BFT consensus," *CoRR*, vol. abs/1807.04938, 2018. [Online]. Available: http://arxiv.org/abs/1807.04938

[30] C. Stathakopoulou, T. David, M. Pavlovic, and M. Vukolić, "Mir-BFT: High-throughput robust BFT for decentralized networks," *Journal of Systems Research*, 2022, to appear.