Interplanetary Consensus (IPC)*

ConsensusLab†

Abstract

Totally ordering transactions constitutes a significant scalability bottleneck standing in the way of massive adoption of blockchain systems, as all participants need to replicate and execute sequentially every single transaction. Layer 2 (L2) protocols aim at resolving this scalability limitation by off-loading state and processing to a loosely coupled sub-system.

We present Interplanetary Consensus (IPC), a new blockchain architecture design that enhances the scaling capabilities of L2+ protocols. Users of IPC can dynamically spawn new blockchain subsystems (subnets) as children of any existing subnet. IPC is based on the design principles of on-demand horizontal scaling. Child subnets leverage the security of their parent subnets by periodically checkpointing their state in the parent's state. IPC provides native communication across subnets within the IPC framework.

We believe that IPC addresses several issues with existing L2 approaches, including limited throughput capacity, isolation from each other, centralized components, or monolithic architectures. In the following, we introduce the overall system architecture, native functionality, and design decisions of our reference implementation for child subnets based on the BFT Trantor consensus protocol, with Filecoin as rootnet.

1 Introduction

Consensus, or establishing total order across transactions, poses a major scalability bottleneck in blockchain networks [61], such as Bitcoin [48], Ethereum [64], or Filecoin [7]. In short, the main challenge with consensus is that it requires all replicas (sometimes called *nodes*, *validators*, or *miners*) to process all transactions. Regardless of the specific consensus protocol implementation used, even if we use the most scalable and efficient consensus protocol theoretically feasible, this makes blockchain performance limited to that of a single replica at best.

This observation makes it practically impossible to process all transactions pertaining to a use case that a particular blockchain network aims at addressing on its base network, also often denoted as *mainnet* or a Layer-1 (L1) network. For instance, if we put all monetary transactions of the entire world on a single L1, every replica of this L1 would need to process all transactions in the world, which seems infeasible and certainly limits us only to vertical scaling of the replicas. Even if this was somehow possible, the hardware requirements for such replicas would be prohibitive, which would hamper decentralization of a blockchain system [62] and, in particular, the ability of people to participate.

In the last few years, this has been well understood by decentralized systems designers and the quest for decentralized system scalability has moved to Layer 2 (L2) and beyond. Here, roughly, an L2 network is a network that anchors certain critical information in an L1 but processes transactions off L1, according to its proper protocol. At the same time an L1 ideally remains umodified, for security and governance reasons. Prominent examples of L2s started

*`https://ipc.space`

†`https://consensuslab.world`

with the Lightning Network (LN) payment channels [51] that serve today as the main L2 for Bitcoin transactions.

After LN, the number of L2 scalability solutions exploded to the extent that it became challenging to even track their number and to navigate the area. For instance, [38] gives an academic overview of state-of-the-art as of 2020 with over 160 references, while [43] offers an informal yet often updated overview of Ethereum scaling proposals listing more than 20 independent efforts which are being deployed as production networks. Other blockchain ecosystems have their own scaling approaches, e.g., Cosmos, Avalanche or the Internet Computer.

There are several glaring issues with existing approaches, of which we list just a few:

- bi-lateral payment and state channels, e.g., in LN, require an L1 transaction to open (and another to close) a channel between every two parties that wish to transact. This poses a scalability challenge which is arguably possible to address to a certain extent by multi-hop payment routing, yet some challenges remain, notably when it comes to generalization of multi-hop payment channels to general state (smart contract) processing;

- existing L2 scaling solutions are often deployed *in isolation*. As a result, interoperability relies either on custom bridges across L2s, or using different protocols to transfer assets and state between different L2s with the help of a common L1, which hampers security and/or usability.

- many L2 scaling designs are fundamentally limited in data throughput capacity of an L1 they are building on (e.g., so-called roll-up L2 solutions which publish transaction data on their L1);

- many L2 scaling efforts have, in their current designs, centralized components (e.g., sequencers [60]) whose decentralization is non-obvious.

With Interplanetary Consensus (IPC), we take up a challenging task of hitting a sweet design spot of L2+ scaling, addressing the above issues. With IPC, we aim at a blockchain scaling architecture that provides considerable *performance*, *decentralization* and *security*, which are known to be conflicting goals [30], while aiming at usability at relatively short time scales. In a nutshell, IPC is based on the following design principles.

Web-scale on-demand horizontal scalability. IPC allows L2+ networks, called *Subnets*, to be spawned on-demand in a permissionless way. Anyone can spawn a subnet and define rules for joining and leaving a subnet (which allows for both permissionless and permissioned membership of subnets).

The basic idea behind subnets in IPC is that they are dynamically deployed, separate and loosely coupled blockchain subsystems, which can communicate seamlessly among each other with augmented security compared to stand-alone blockchain systems thanks to a to hierarchical subnet communication structure used in IPC.

Subnets are meant to be used in different ways. For instance, subnets can host different (sets of) applications, or can be used to shard a single application.

L2+ child subnets leverage security of their parent subnets. IPC is organized in a hierarchical fashion, where each subnet, except for one that we call the *rootnet*, is associated with exactly one other subnet called its *parent*. Conversely, one parent can have arbitrarily many subnets, called *children*, associated with it.

This yields a tree structure of subnets, in which child subnets periodically checkpoint critical information, e.g., their state, or membership information, to their parent subnet. Checkpointed

history of a child subnet cannot be reverted even if there is a trust assumption violation in a child subnet, as long as a parent subnet operates as expected. Checkpoints are propagated in a recursive way all the way to the rootnet (L1), which makes child subnets benefit from security of their ancestor subnets.

This tree of subnets expresses a hierarchy of trust. All components of a subnet and all Participants using it are assumed to fully trust their parent; in the IPC trust model, this is accomplished by having each participant in a subnet either run a full node in said subnet and all its ancestor subnets, or it trusts some other participant which runs a full node. Note that, in general, trust in all components of the parent subnet is not required, but the parent system as a whole is always assumed to be correct (for some definition of correctness specific to the parent subnet) by its child.

Lightweight communication across subnets and IPC hierarchy. Conceptually, IPC has built-in interoperability across subnets leveraging the IPC tree hierarchy, without relying on custom bridges across subnets. This feature, which appears unique to IPC, allows Participants of the IPC framework to seamlessly communicate with other subnets belonging to the same IPC tree.

Communication across subnets in IPC is facilitated by two *actors* (equivalent to smart-contracts in Ethereum terminology): the IPC Gateway Actor (IGW) and the IPC Subnet Actor (ISA), as well as a lightweight process we call the *IPC agent* which submits the transactions involved in inter-subnet interaction. Concretely, IPC agents read the replicated state of one subnet and submit transactions on its behalf to another subnet. Participants running those IPC agents get rewarded for such mediation.

The IPC architecture directly provides several primitives for cross-subnet interaction, such as:

1. Transfer of funds between accounts residing in different subnets.
2. Saving checkpoints (snapshots) of a child subnet's replicated state in the replicated state of its parent.
3. Submitting transactions to a subnet by the application logic of another subnet.

Out-of the box support for decentralized subnets. IPC has been designed to avoid centralization points, unless users explicitly want to deploy single-replica subnets. IPC runs decentralized consensus (total-order broadcast) protocols across replicas belonging to individual subnets.

Configurable consensus protocols and governance rules for subnets. In IPC, both consensus (total-order) protocols and the rules for value deposits and withdrawals from subnets can be customized. Customizable consensus in IPC allows for use of low latency and high throughput protocols at the leaves of IPC hierarchy.

Our implementation. Our implementation of a subnet node, called Eudico[1] is a fork of the Lotus Filecoin L1 node[2]. Eudico modularises Lotus in a way which allows for pluggable consensus protocols based on the *Mir* framework for developing distributed protocols[3] which our

[1] `https://github.com/consensus-shipyard/lotus/`
[2] `https://github.com/filecoin-project/lotus`
[3] `https://github.com/filecoin-project/mir`

group also developed. Currently, Eudico features implementation of a modern Byzantine fault-tolerant (BFT) protocol, that we call *Trantor*[4]. Trantor is a multi-leader BFT protocol which draws inspiration from recently proposed high-throughput BFT consensus protocols, namely ISS [57] and Narwahl [32]. Additionally, Eudico also supports Filecoin's Expected Consensus and Nakamoto's Proof-of-Work, and the Mir framework allows pluggability of other consensus protocols. In addition to Eudico, we are also currently exploring plugging subnets based on Tendermint Core[5] into IPC.

Outline. In the rest of this document, we describe IPC in detail. Section 2 introduces a sample use case of IPC that we use throughout as a running example. In Section 3 we define preliminary concepts and assumptions used by IPC. Sections 4 and 5 respectively describe IPC's functionality in terms of the interaction between a parent and a child subnet and summarize the high-level implementation of the two actors IPC uses to enable this interaction. We discuss related work in Section 6. Finally, our implementation of IPC and its deviations from the general design are laid out in Section 7.

2 Example Use Case: On-Line Gaming Platform

To better understand how IPC works and how it is useful, let us imagine an example application of a distributed on-line gaming platform. Consider a platform where registered players meet and play games against each other, while the platform maintains player rankings. Tournaments can be organized as well, where each participant pays a participation fee and the winner(s) obtain prize money (both in form of coins). We now describe how IPC could be used to build this hypothetical application in a fully distributed fashion.

Rootnet with all users' funds (`L1`). The rootnet is used as a financial settlement layer. Most users' coins are on accounts residing in the rootnet's replicated state. A robust established blockchain system like Filecoin could be a reasonable candidate for use as the rootnet. Its relatively higher latency and lower throughput (that is often the price for security and robustness) is not a practical issue, as users will rarely interact directly with it.

Gaming platform as a subnet (`L2`). The functionality of the gaming platform (such as maintaining score boards, recommending opponents to players, or organizing tournaments) is implemented as a distributed application on a dedicated subnet. This subnet uses a significantly faster BFT-style consensus protocol (such as Trantor) since the application needs to be responsive for the sake of user experience, and deals, in general, with fewer funds than the rootnet (only as much as users dedicate to playing). The replicas constituting this subnet are run by gaming clubs or even some (not necessarily all) individual players (who do not necessarily trust each other, e.g. to not manipulate the score boards). To have a replica in the `L2` subnet, the club (or the player) needs to lock a certain amount of funds as collateral that can be slashed by the system if the replica misbehaves.

Individual games (`L3`). For each individual game, a new child of the `L2` subnet is created (Section 4.1) and acts as a (distributed) game server. Since not much is usually at stake in a single game and only few players are involved, the whole `L3` subnet may even be implemented by a single server the players trust. However, this decision is completely up to the players

[4] `https://github.com/filecoin-project/mir/tree/main/pkg/systems/trantor`
[5] `https://github.com/tendermint/tendermint/`

and they may choose a different implementation of the L3 subnet when starting the game (by submitting the corresponding transactions to the L2 subnet). When the game finishes, its result is automatically reported to the L2 subnet (Section 4.5), which updates the players' ratings accordingly, and the L3 subnet is disposed of (Section 4.6).

Player accounts. Each player has an account on the L2 subnet where they deposit funds (Section 4.2) from the rootnet by submitting a corresponding L1 transaction[6]. They use these funds to pay transaction fees on the L2 subnet and tournament registration fees. A player can transfer funds back to their L1 account through a withdraw operation (Section 4.3) by submitting an L2 transaction.

Tournaments. Tournaments can be organized using the platform, where each player registers by submitting a corresponding L2 transaction. When the tournament finishes, the winner receives the prize money (obtained through the registration fees) on their L2 account. One can also easily imagine that only part of the collected fees transforms to the prize, while the rest can remain in the platform and be used for other purposes, such as rewarding the owners of the replicas running the subnet hosting the platform (i.e., the L2 subnet). To stretch the example even further, one could imagine a tournament being implemented as an L3 subnet, while the tournament's individual games are its children (L4).

This simple use case utilizes most of IPC's features. Throughout the rest of the document, we will use the on-line gaming platform as a running example when describing IPC's functionality in more detail.

3 Preliminaries

The vocabulary used throughout this document is described in the Glossary [58] (e.g., *subnets*, *actors*, *accounts*, *users*, and *IPC agents*). The reader is assumed to be familiar with the terminology defined there. [**js:** Despite this note, the vocabulary seems to be defined throughout the body. If we're defining in the body anyway, the appendix seems redundant – this isn't a book.] [**mv:** If the glossary is stable, I suggest bringing it to the main body of the paper by defining concepts inline, as we go. Glossary can additionally stay in the appendix, for quick reference.]

Basic abstractions. A *subnet* consists of multiple *replicas*, yet we abstract a subnet as a single entity which maintains an abstraction of *replicated state* (of which each replica maintains a copy and that all replicas agree on) that can only be modified through *transactions* submitted either by *users* or by an *IPC agent*. A sequence of transactions can be batched into a *block* to amortize the ordering overhead. We further abstract away the concrete mechanism of transaction submission and execution, as it is specific to the implementation of each particular subnet.

For example, the replicated state of a subnet representing a game of chess would consist of the players' identities, a flag indicating which player's turn it is, and positions of the individual pieces on the board, while players' moves would be performed by submitting transactions to the subnet.

[6] We call a transaction submitted to the L1 subnet an "L1 transaction".

Interaction between subnets. In IPC, the replicated state (or, simply, state) of one subnet often needs to react to changes in the state of another subnet. E.g., after a game hosted in a subnet finishes, the implementation of the game logic might need to update the involved players' rankings in its parent subnet based on the result of the game. As the state of every subnet evolves independently of the state of other subnets, *IPC establishes a protocol for interaction between the states of different subnets.*

At the basis of the protocol, IPC relies on *Proofs of Finality (PoF s).* In a nutshell, a *PoF* is data that proves that a subnet irreversibly reached a certain replicated state. Regardless of the approach to *finality* that the *ordering protocol* of a subnet uses (e.g., immediate finality for classic BFT protocols [37], or probabilistic finality in PoW-based systems [48]), a *PoF* serves to convince the verifier that the replicated state the *PoF* refers to will not be rolled back. This helps IPC establish the partial ordering between the states of two subnets.

For example, for a subnet using a BFT-style ordering protocol, a quorum of signatures produced by its replicas can constitute a *PoF*. To prove the finality of the state of a subnet based on a longest-chain-style protocol, a *PoF* might consist of signatures of a committee of processes considering the state deep enough (for some parametrized notion of "deep enough") in the chain. This committee can be, for example, a quorum of replicas of the very subnet that is to verify the *PoF*. If the verifying subnet itself is a longest-chain one, a *PoF* can be as simple as a hash of the proving subnet's block, with every replica of the verifying subnet deciding locally about the *PoF*'s validity (potentially leading to forks in the worst case).

If a *PoF* is associated with subnet `A`'s replicated state at *block height* $h_{\mathtt{A}}$, and the *PoF* is included in subnet `B`'s replicated state at block height $h_{\mathtt{B}}$, then subnet `B`'s replicated logic will consider all `A`'s state changes up to $h_{\mathtt{A}}$ to have occurred at `B`'s height $h_{\mathtt{B}}$. (Unless, of course, another *PoF*' of $h'_{\mathtt{A}}$ has been included by `B` at $h'_{\mathtt{B}}$, in which case `B` consideres only the state changes between $h'_{\mathtt{A}}$ and $h_{\mathtt{A}}$ to have occured at $h_{\mathtt{B}}$).

In the following, for some representation of a subnet's replicated state (e.g., its full serialization or a handle that can be used to retrieve it in a content-addressable way, such as an IPFS content identifier (CID)), we denote by $PoF(state)$ the proof that a subnet reached *state*. We also denote by $PoF(tx)$ the proof that a subnet reached a state in which transaction tx already has been applied to the replicated state.

Naming subnets. We assign each subnet a name that is unique among all the children of the same parent. Similarly to the notation used in a file system, the name of a child subnet is always prefixed by the name of its parent. For example, subnets `P/C` and `P/D` would both be children of subnet `P`.

Representing value. For each pair of subnets in a parent-child relationship, we assume that there exists a notion of *value* (measured in *coins*) common to both subnets.[7] We represent this value by associating some number of coins (also referred to as funds) with accounts and actors in a subnet's replicated state. The number of coins associated with an account is the account's *balance*. Each user is assumed to have an account in each subnet the user interacts with. All the transactions spending coins from an account must be signed by the corresponding user's private key. For ease of presentation, we do not explicitly include these signatures in the further description of IPC.

We also assume that the submission, ordering, and application of transactions is associated with a cost (known as transaction fees, or *gas*). Each subnet client (user wallet or IPC agent)

[7] One can easily generalize the design to decouple the use of value between a parent and its child, but we stick with using the same kind of value in both subnets for simplicity.

submitting a transaction to a subnet must have an account in that subnet, from which this cost is deducted. If the funds are insufficient, the subnet may fail to execute the transaction.

Note that the operation of IPC requires the submission and processing of transactions that are not easily attributed to a concrete user. This is the case with transactions that an IPC agent submits on behalf of a whole subnet. [**mp:** Move this part after IPC agent and actors, as it already uses those terms. On the other hand, it would be nice to have it before the trust model. Maybe we could move the trust model to be last...]

Notation. We refer to an account `a` in the replicated state of subnet `S` as `S.a`. To denote a function of an actor in the replicated state of a subnet, we write `Subnet.Actor.`*`Function`*. E.g., the *IPC Gateway Actor* (`IGA`) function *`CreateChild`* in subnet `P` is denoted `P.IGA.`*`CreateChild`*. We also use this notation for a transaction tx submitted to subnet `P` that invokes the function, e.g., $tx =$ `P.IGA.`*`CreateChild`*$(\texttt{P/C}, params)$.

Trust model. IPC can be deployed in an adversarial environment, where some participants might be malicious (Byzantine) and actively try to subvert the system. For each subnet, the assumptions under which the subnet's implementation is guaranteed to operate correctly may differ. While some may rely on honest participants controlling a certain fraction of the overall involved resource such as computing power, storage capacity, or staked collateral, others may depend on a simple majority of honest replicas. In case of a violation of these assumptions, there are no guarantees about the replicated state of the affected subnet. The users of a subnet must bear this risk and choose the subnets to use accordingly.

IPC as a system is based on two fundamental principles:

1. *Hierarchical trust*: Whatever is part of the replicated state of a subnet is considered a ground truth by all children of that subnet. For example, if a child subnet's membership (i.e., set of replicas) is maintained in the state of its parent (as is the case for IPC-native PoS-based subnets, see Section 4.7), long-range attacks within the child subnet can be easily ruled out. If a parent subnet fails as a whole, there are no guarantees about the correct behavior of its children. However, if the parent stops being available, the child subnet can continue to work, albeit disconnected from the rest of the IPC hierarchy until the parent becomes available.

2. *Subnet firewall property*: In case a whole subnet fails (e.g., if the underlying assumptions made by the protocol it is based on are violated), at most as many coins can be impacted by the failure (e.g., double-spent or permanently lost) as have been deposited to the subnet from its parent. That is, a user is able to manage the risk of using a potentially non-robust subnet through the amount of funds they deposit and/or accept to receive in it.

The above principles enable child subnets to "inherit" some of their parents' robustness through *checkpointing*, where a child subnet regularly includes its replicated state (or a reference to it) in its parent's replicated state. In case the child subnet fails, there exists a record of the evolution of its state in the parent. This enables participants (e.g., former users of the failed subnet) to agree on picking up an older version of the child subnet's state from before the occurrence of the failure and, say, use that version as the initial state of a new, more robust subnet.

We envision subnets higher up in the hierarchy to be more robust than their descendants, in the sense that it should be harder / less probable to violate the assumptions their correctness is based on. For example, a robust public system such as Filecoin, backed by a substantial amount of resources, can serve as a the rootnet, while its descendant subnets, where less is at stake in

case of a failure of the whole subnet, can more easily sacrifice a part of their robustness, e.g., for the sake of performance.

IPC actors. For inter-subnet communication, IPC relies on two special types of actors: the IPC Gateway Actor (`IGA`) and the IPC Subnet Actor (`ISA`). In a nutshell, their functions are as follows.

1. The `IGA` is an actor that contains all IPC-related information and logic associated with a subnet that needs to be replicated *in the subnet itself*. Each subnet contains exactly one `IGA`. We describe the `IGA` in detail in Section 5.1

2. The `ISA` is the `IGA`'s parent-side counterpart, i.e., it is an actor in a parent subnet's replicated state, containing all the data and logic associated with a particular child subnet – we say the `ISA` "governs" that child subnet. A subnet contains as many IPC Subnet Actors as the number of its children. We refer to the `ISA` governing child subnet `C` as $\texttt{ISA}_{\texttt{C}}$. We describe the `ISA` in detail in Section 5.2.

IPC agent. The IPC agent is a process that mediates the communication between a parent and a child. It has access to the replicated states of both subnets and acts as a client of both subnets. When the replicated state of subnet `A` indicates the need to communicate with subnet `B`, the IPC agent constructs a *PoF* for `A`'s replicated state and submits it as a transaction to `B`.

IPC does not prescribe who must run an IPC agent, nor how the *PoF* is to be constructed, nor which IPC agent must submit the transaction with the *PoF*. All this is specific to the implementation of the subnets involved. The IPC agents may run a protocol for choosing which of them submit(s) the transaction, or even resort to a simplistic approach where each IPC agent submits an identical transaction. In our reference implementation (Section 7), for example, we construct the *PoF* directly in the verifying subnet's (`B`'s) replicated state, using one IPC agent per replica of the proving subnet (`A`). When describing the functionality of IPC and referring to "the IPC agent" submitting a *PoF*, we assume such a subnet-specific mechanism for choosing one (or multiple) IPC agent(s) to perform the actual submission.

The interaction between subnets through IPC agents is depicted in Fig. 1.

Incentives. In general, submitting transactions (and their subsequent execution by the subnet) is associated with a *cost* (often referred to as "gas"). We refer to the cost associated with a transaction as the *transaction fee*, measured in coins. A participant running an IPC agent is not necessarily interested in participating in such a costly protocol without incentives. Moreover, the replicas of a subnet might need to cooperate with IPC agents during the construction of Proofs of Finality. Even though certain deviations from the protocol can be detected and penalized (see Section 4.7.3), participants running subnet replicas might also need positive incentives to participate in the creation of a *PoF*.

The key to providing incentives for IPC agents and replicas is that the `ISA` and the `IGA` can, as actors, hold funds that their logic can distribute among other accounts or actors on their respective subnets. Thus, the actors can be configured to reimburse an account associated with the IPC agent submitting a transaction (potentially adding an extra reward), as well as to reward or penalize accounts.

The source of funding for the `IGA` and / or `ISA` is subnet-specific. For example, a subnet's implementation can require a certain part of each transaction fee to be sent to the subnet's `IGA`. An `ISA` can be funded, for example, through transfers (withdrawals) of funds from the child subnet, or by charging fees for propagating cross-net transactions.

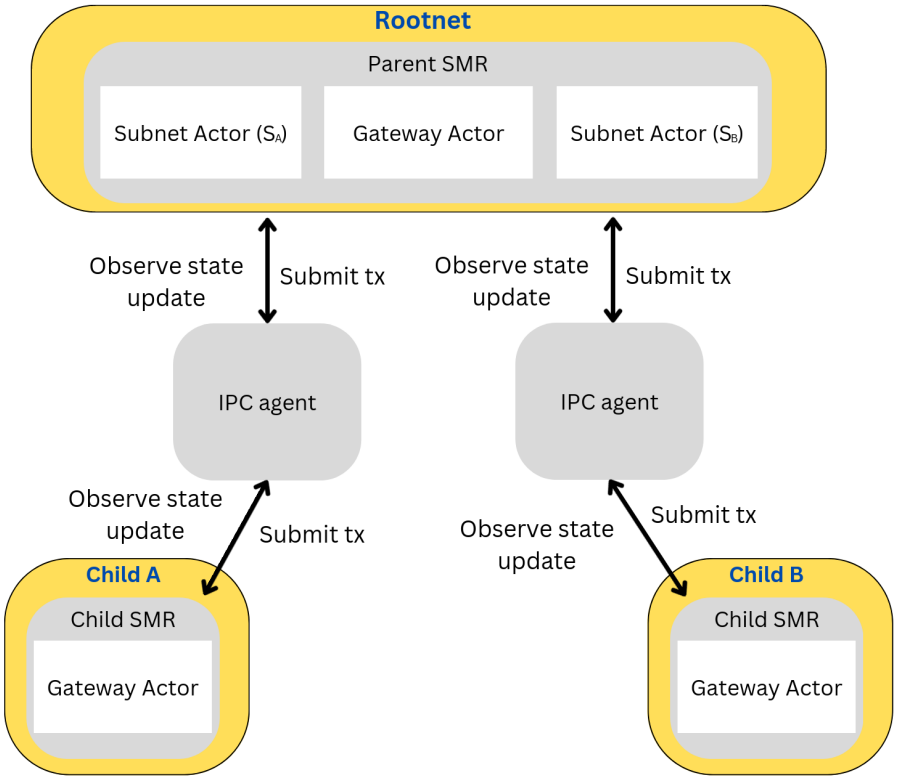

Figure 1: The basic IPC components and their interfaces in an example with one parent and 2 child subnets (A and B).

To incentivize the replicas of a subnet to collaborate with an IPC agent on the creation of Proofs of Finality, a similar mechanism can be deployed. For example, a valid *PoF* would include metadata, where the replicas that participated in its creation could insert an address to receive a reward when the *PoF* is accepted.

Example 1. *Our gaming platform from Section 2 could use the following approach. Each replica of the* `L2` *subnet running the gaming platform is registered in the* `IGA` *of the subnet itself and has an associated account controlled by the participant operating the replica. (Such a participant could be, for example, a local club of players who bought a server and are paying for internet connection.) The* `L2` *subnet is configured to pay a small fee to the* `IGA` *for each transaction executed by the subnet. The* `IGA` *periodically (e.g., every 1000 blocks) re-distributes a part of the collected fees to the accounts associated with the replicas.*

Each participant also controls an account in the rootnet (`L1`*) and operates an IPC agent (e.g., running on the same machine as the replica) to mediate the interactions between* `L1` *and* `L2` *(e.g., periodic checkpointing of* `L1`*'s state to* `L2`*). The definition of* $\texttt{ISA}_{\texttt{L2}}$ *requires a valid PoF to contain a withdrawal transaction transferring a certain amount of funds (e.g., from the* `IGA` *in* `L2`*) to the* `L1` *account associated with the IPC agent submitting the PoF. This way, the participant operating the IPC agent has an incentive to submit the PoF. To prevent too many or repeated submissions, the* $\texttt{ISA}_{\texttt{L2}}$ *adjusts the reward based on the most recent PoF already submitted.*

4 IPC parent-child interactions

We now focus on the interaction between two subnets in a parent-child relation, which is the basic building block of the recursive IPC hierarchy. The IPC interface exposes the following functionalities:

1. Creating child subnets in the IPC hierarchy.
   E.g., when starting a new game on the gaming platform.
2. Depositing funds from an account in a subnet to an account in its child.
   E.g., when a player tops up the balance of their account on the gaming platform.
3. Withdrawing funds from an account in a subnet to an account in its parent.
   E.g., when a player withdraws money they won in a tournament.
4. Checkpointing a subnet's replicated state in the replicated state of its parent.
   E.g., after every 100 blocks of transactions applied to the gaming platform's subnet.
5. Invoking actor functions across subnets, i.e., the replicated logic of one subnet acting as a client of another subnet.
   E.g., when a game finishes and players' rankings are automatically updated.
6. Removing child subnets from the IPC hierarchy.
   E.g., when a game fihishes, rankings have been updated, and the state of the game can be disposed of.
7. Managing proof-of-stake subnets, exposing functions for adding and removing replicas, managing the associated collateral, and slashing of provably misbehaving replicas.
   E.g., when mutually distrusting players play together.

In the following, we describe each functionality in detail, introducing the functions of the `IGA` and `ISA` through which this functionality is exposed and the patterns in which the users and the IPC agent invoke them via transactions.

4.1 Creating a child subnet

To create child subnets, the `IGA` exposes the following function.

$$\texttt{IGA}.\mathit{CreateChild}(\mathit{subnetName})$$

Any user or actor of a subnet `P` can create a new child subnet `P/C` by

1. creating a new instance of the IPC Subnet Actor $\texttt{ISA}_\texttt{C}$ and
2. submitting a transaction $\texttt{P.IGA}.\mathit{CreateChild}(\texttt{C})$.

The new actor $\texttt{ISA}_\texttt{C}$ must be configured with all the subnet-specific parameters relevant for governing the new subnet. These would usually include the used consensus protocol, rules for joining the subnet, definitions and evaluation logic for PoMs and PoFs, and slashing policies. From the perspective of the IPC hierarchy, the subnet is considered created as soon as $\texttt{ISA}_\texttt{C}$ is created. The subnet itself need not necessarily be operational at this moment, as the parent subnet always has a passive role when it comes to interacting with it.

Example 2. *Imagine that a player wants to create a game subnet to play against 3 opponents, such that each player will run their own replica in the game subnet (i.e., their own copy of the game server). As an incentive for honest behavior, the player decides that the child subnet will be PoS-based (see Section 4.7), where each replica must be backed by a minimal collateral of 10 coins that would be slashed (and, say, redistributed to the other players) if the replica is caught misbehaving (see Section 4.7.3).*

To achieve this, the player creates a subnet actor governing a child subnet that allows a replica to join only if at least 10 coins of collateral are associated with it, and stops accepting new collateral after 4 replicas (the player and 3 opponents) reach the threshold of 10 coins. The player then registers the new subnet through a `P.IGA.CreateChild` *transaction, starts their own replica of the game server (we assume the game server is implemented such that it can be replicated and run as a subnet) and waits for other players's replicas to join (see Section 4.7.1).*

Note that, since the subnet actor is created by the user, its initial state and logic can be configured arbitrarily. For example, the `ISA` *could easily be configured with other admission policies (only players with a game ranking within a defined range) or slashing policies (penalize misbehaving replicas by the backing player loosing not just coins, but also their position in a game-specific ranking system).*

4.2 Depositing funds

A deposit is a transfer of funds from an account in the parent subnet to an account in the child subnet. The following functions are exposed by the IPC actors to enable deposits.

`ISA.`*`Deposit`*(*amount, account*)
`IGA.`*`MintDeposited`*(*amount, account, PoF*)

The *amount* is the amount of funds to be deposited, *account* is the destination account in the child subnet, and *PoF* is a proof of finality proving that `ISA.`*`Deposit`*(*amount, account*) has been applied to the parent subnet's replicated state and that state is final (i.e., cannot be rolled back).

Depositing *amount* coins from an account `P.a` in the parent subnet `P` to an account `P/C.b` in the child subnet `P/C` involves the following steps.

1. The owner of `P.a` submits, using their wallet, a transaction
   tx = `P.ISA`$_{\texttt{C}}$`.`*`Deposit`*(*amount*, `b`).

2. `P` orders and executes *tx*, transferring *amount* coins from `a` to `ISA`$_{\texttt{C}}$.

3. When `P`'s replicated state that includes *tx* becomes final (for some subnet-specific definition of finality provable to `P/C.IGA`, which contains the *PoF* verification logic), the IPC agent constructs a *PoF*(*tx*).

4. The IPC agent submits a transaction[8]
   tx' = `P/C.IGA.`*`MintDeposited`*(*amount*, `b`, *PoF*(*tx*)).

5. `P/C` orders and executes *tx'*, which results in minting *amount* new coins and adding them to the balance of `P/C.b`.

[8]In a practical implementation, instead of submitting a separate transaction for each deposit, the IPC agent may submit multiple deposits batched in a single transaction, with a single *PoF* proving the finality of the child state in which all the corresponding funds have been transfered to `ISA`$_{\texttt{C}}$. This optimizes both performance and cost (transaction fees).

After all the above steps are performed, the newly minted *amount* of coins at the child is backed by the analogous locked amount at $\texttt{P.ISA}_\texttt{C}$. However, those coins can effectively only be used by the owner of `P/C.b`, since $\texttt{P.ISA}_\texttt{C}$ will not transfer its coins within `P` until they are burned in `P/C` during a withdrawal operation (see below).

Example 3. *Imagine that a player wants to start using the gaming platform (running on the subnet `P/C`), but only has funds in an account `P.a` in the parent subnet. To be able to join games (L3) that require a collateral (as in Example 2), the player decides to fund their account `P/C.b` (L2) with 20 coins. Thus, the player submits a $\texttt{P.ISA}_\texttt{C}$`.Deposit(20, b)` transaction, starts an IPC agent process that performs steps 3 and 4 above, and waits until the funds appear on `P/C.b`.* *[**gg:** Regarding colllateral, this example is a bit confusing to me. I thought collateral is to be held at the parent (at least for the suggested implementation).][**arp:** I think that in this example, games that require collateral in this example are L3 not L2, but yes, we should maybe not use collateral here, but instead betting on games or something that is less confusing (chipping in a prize for the winner). I added L3 and L2 for clarity]*

4.3 Withdrawals

A withdrawal is a transfer of funds from an account in the child subnet to an account in the parent subnet. The following functions are exposed by the IPC actors to enable withdrawals.

$$\texttt{IGA.}\textit{Withdraw}(\textit{amount, account})$$
$$\texttt{ISA.}\textit{ReleaseWithdrawn}(\textit{amount, account, PoF})$$

The *amount* is the amount of funds to be withdrawn, *account* is the destination account in the parent subnet to which the withdrawn funds are to be credited, and *PoF* is a proof of finality proving that $\texttt{IGA.}\textit{Withdraw}(\textit{amount, account})$ has been applied to the child subnet's replicated state and that state is final (i.e., cannot be rolled back).

Withdrawing *amount* coins from an account `P/C.b` in the child subnet `P/C` to an account `P.a` in the parent subnet `P` involves the following steps.

1. The owner of `P/C.b` submits, using their wallet, a transaction $tx = \texttt{P/C.IGA.}\textit{Withdraw}(\textit{amount}, \texttt{a})$.
2. `P/C` orders and executes *tx*, burning *amount* coins from `b`.
3. When `P/C`'s replicated state that includes *tx* becomes final (for some subnet-specific definition of finality provable to $\texttt{P.ISA}_\texttt{C}$), the IPC agent constructs a $PoF(tx)$.
4. The IPC agent submits a transaction[9] $tx' = \texttt{P.ISA}_\texttt{C}\texttt{.}\textit{ReleaseWithdrawn}(\textit{amount}, \texttt{a}, PoF(tx))$.
5. `P` orders and executes *tx'*, which results in $\texttt{P.ISA}_\texttt{C}$ transferring *amount* coins to account `P.a`.

The above procedure ensures that the locked *amount* at the parent is not released until the child has already burned the minted *amount* of coins. The $\texttt{P.ISA}_\texttt{C}$ actor ensures (by verifying the associated *PoF*) that the coins have been burned in `P/C` before releasing the corresponding *amount* back into circulation in `P`.

[9] Like with deposits, withdrawals can also be batched for better performance and lower cost. Our implementation applies this optimization to both deposits and withdrawals, further combined with checkpoints (see Section 7).

Example 4. *A player might want to stop using the gaming platform and withdraw all the funds back to the parent subnet, in order to spend them on something else. They still have 20 coins on their account* `P/C.b` *that they want to transfer back to* `P.a`*. The player performs the withdrawal by submitting a transaction* `P/C.IGA.Withdraw(20, a)`*, starts an IPC agent process to perform the necessary inter-subnet communication, and waits until the coins arrive at* `P.a`*. (The player might need to spend a part of the 20 coins on fees for both the* `Withdraw` *and the* `ReleaseWithdrawn` *transactions.)*

4.4 Checkpointing

Checkpointing is a method for a parent subnet to keep a record of the evolution of its child subnet's replicated state by including snapshots of the child's replicated state (called checkpoints) in the parent's replicated state. If, for some reason, the child subnet misbehaves as a whole (e.g., by a majority of its replicas being taken over by an adversary), agreement can be reached in the parent subnet about how to proceed. For example, which checkpoint should be considered the last valid one, and potentially used as the initial checkpoint (equivalent in concept to a "genesis block") for a new sanitized subnet. The following function is exposed by the `ISA` to enable checkpointing.

$$\texttt{ISA.}\textit{Checkpoint}(\textit{snapshot}, \textit{PoF})$$

A checkpoint can be triggered by predefined events (e.g., periodically, after a number of state updates, triggered by a specific user or set of users, etc.). The IPC agent is configured with the (subnet-specific) checkpoint trigger, monitors the child subnet's replicated state, and takes the appropriate action when the trigger condition is satisfied by the child subnet's state. A checkpoint of subnet `P/C` to its parent `P` is created as follows:

1. When the predefined checkpoint trigger is met in the replicated state of `P/C`, the IPC agent retrieves the corresponding snapshot of `P/C`'s replicated state (*state*) from the child subnet and constructs the proof of its finality $PoF(state)$.

2. The IPC agent submits a transaction
   $tx = \texttt{P.ISA}_\texttt{C}\texttt{.}\textit{Checkpoint}\,(state,\, PoF(state))$.

3. `P` orders and executes *tx*, which results in $\texttt{P.ISA}_\texttt{C}$ including *state* (i.e., the checkpoint of `P/C`'s replicated state) in its own actor state.

4.5 Propagating cross-net transactions

Cross-net transactions are a means of interaction between actors located on different subnets. Unlike a "standard" transaction issued and submitted to a subnet by a user's wallet, a cross-net transaction is issued by actors of another subnet.

Since those actors themselves are not processes (but mere parts of a subnet's replicated state), they cannot directly submit transactions to other subnets. IPC therefore provides a mechanism to propagate these transactions between subnets using the following functions of the `IGA`.

$$\texttt{IGA.}\textit{Dispatch}(\textit{tx, src, dest})$$
$$\texttt{IGA.}\textit{Propagate}(\textit{tx, src, dest, PoF})$$

In a nutshell, if an actor's logic in subnet $\mathtt{S}_1$ produces a transaction for a different subnet $\mathtt{S}_2$, it calls $\mathtt{S}_1$.IGA.*Dispatch*, which saves the transaction in $\mathtt{S}_1$'s IGA buffer that we call the *postbox*. IPC agents, monitoring the postbox, then iteratively submit the transaction to the appropriate next subnet along the path from $\mathtt{S}_1$ to $\mathtt{S}_2$ using IGA.*Propagate*.

Since, in general, we only rely on an IPC agent to be able to submit transactions to the parent or children of a subnet whose state it observes, an IPC agent only propagates the transaction to the parent or child, depending on which is next along the shortest path from $\mathtt{S}_1$ to $\mathtt{S}_2$ in the IPC hierarchy. After such "one hop", the transaction is again placed in the postbox of the parent / child, and the process repeats until the transaction reaches its destination subnet.

More concretely, we illustrate the propagation of a cross-net transaction using an example where an actor in subnet P/A is sending a cross-net transaction tx to its "sibling" subnet P/B. tx is first propagated from P/A to its parent P, which, in turn, propagates it to its other child P/B. We use the function IGA.*Dispatch* in a subnet to announce that the transaction is ready to be propagated and the function IGA.*Propagate* to notify a subnet about a cross-net transaction to be passed on (or delivered, if the destination has been reached).

1. An actor P/A.ActorA constructs a transaction
   tx = P/B.ActorB.*SomeFunction*($someParams$)
2. P/A.ActorA invokes the funcion P/A.IGA.*Dispatch*(tx, P/A, P/B) (note that no additional transactions are necessary here).
3. The implementation of P/A.IGA.*Dispatch* adds tx along with the routing metadata to a P/A.IGA.postbox.
4. Let $state_{\mathtt{A}}$ be the state of subnet P/A where tx is already included in P/A.IGA.postbox. When the IPC agent responsible for the interaction between P/A and P detects that $state_{\mathtt{A}}$ is final, it constructs a $PoF(state_{\mathtt{A}})$ and submits a transaction
   $tx_{\mathtt{A}}$ = P.IGA.*Propagate*(tx, P/A, P/B, $PoF(state_{\mathtt{A}})$).
5. Subnet P orders and executes $tx_{\mathtt{A}}$, verifying $PoF(state_{\mathtt{A}})$ and (internally) invoking P.IGA.*Dispatch*(tx, P/A, P/B). This, in turn, adds tx along with its routing metadata to P.IGA.postbox.
6. Analogously to step 4, the IPC agent submits a transaction
   $tx_{\mathtt{P}}$ = P/B.IGA.*Propagate*(tx, P/A, P/B, $PoF(state_{\mathtt{P}})$), where $state_{\mathtt{P}}$ is the state of P with tx already included in P.IGA.postbox.
7. Upon ordering and executing $tx_{\mathtt{P}}$, P/B.IGA.*Propagate* verifies $PoF(state_{\mathtt{P}})$. Detecting that the destination is the own subnet, the implementation of P/B.IGA.*Propagate* executes tx instead of propagating it.

Example 5. *In our gaming example, imagine that a game (running in its own subnet that is a child of the gaming platform's subnet) has finished and the ranking of the involved players needs to be updated. The game server is implemented as an actor on the game's own subnet, while the gaming platform (storing the player ranking tables) is an actor of its parent. To update the ranking, the game actor would use a cross-net transaction to inform the platform actor about the results of the game and the platform actor would update the rankings accordingly.*

4.6 Removing a child subnet

To remove child subnets, the IGA exposes the following function.

IGA.*ToRemove*()

IGA.*RemoveChild*($subnetName$, PoF)

The function `IGA.ToRemove()` marks the subnet to be removed in the `IGA` of the subnet, while `IGA.RemoveChild(`*subnetName*,*PoF*`)` effectively deregisters the subnet from the IPC hierarchy. Removing a child subnet `P/C` from the IPC hierarchy is performed in the following steps:

1. A replica of `P/C` submits a transaction *tx*=`P/C.IGA.ToRemove()`. The validity of this transaction to be included in the subnet's replicated state is subnet-specific, and may require coordination among the replicas to validate the removal of the subnet, as well as additional parameters to the function.

2. When `P/C`'s replicated state that includes *tx* becomes final, the IPC agent constructs a $PoF(tx)$.

3. The IPC agent submits a transaction
   tx' = `P.IGA.RemoveChild(P/C,` $PoF(tx)$`)`.

4. `P` orders and executes *tx'*, which results in `P.IGA` deregistering `P/C` from the IPC hierarchy.

As the deposited funds are in the `ISA`, the subnet can still keep operating for all accounts to withdraw their funds back at the parent.

4.7 Proof-of-stake subnets

In order to disincentivize replicas of a subnet from misbehaving, IPC provides a mechanism for conditioning a replica's participation in the child subnet on *collateral* in a proof-of-stake fashion. To this end, the `ISA` can associate each replica of the child subnet with a collateral. Replicas must transfer this collateral to the `ISA`, and the `ISA` only releases the collateral back once the corresponding replica stops participating in the subnet. The way in which the collateral associated with replicas impacts the functioning of the child is subnet-specific.

If a child replica provably misbehaves, the proof of such misbehavior can be submitted as a transaction to the `ISA` (invoking its *Slash* function). The `ISA` then decreases the amount of collateral associated with the offending replica in accordance with its (subnet-specific) slashing policy.

Note that collateral is different from funds deposited for use in the child subnet. Unlike the deposited funds, collateral is not made available in the child subnet and stays in the parent's `ISA` until the associated replica stops participating in the subnet, either by leaving or by being slashed.

The advantage of this approach is that a child subnet can leverage funds in the parent subnet to serve as collateral. In case of provable misbehavior even of replicas that have gained complete power over the child subnet (e.g., by having staked most of the collateral), those replicas can still be slashed at the parent. It also prevents long-range [27] and similar attacks by maintaining membership information (sometimes also referred to as power table) of the child subnet at the parent. The membership saved in the `ISA`'s state defines the ground truth about what replicas the child subnet should consist of, as well as their relative voting power (proportional to the staked collateral) in the underlying ordering protocol. The child subnet observes the state of the `ISA` in the parent and, when the membership information changes, reconfigures accordingly.

Since we expect PoS-based child subnets to become very common, IPC provides the following native functionality for managing PoS-based subnets:

- Manipulate the membership by staking and releasing collateral associated with replicas

- Slashing, where IPC permanently removes / redistributes a part of the collateral associated with a provably misbehaving replica (e.g., one that sends conflicting messages in the ordering protocol)

Example 2 provides a concrete case of where and how PoS-based subnets can be used.

4.7.1 Staking collateral

To increase a replica's collateral in a PoS-based subnet, IPC uses the following functions.

`ISA.StakeCollateral`(*account, replica, amount*)
`IGA.UpdateMembership`(*membership, PoF*)

Concretely, to increase the amount of collateral associated with a *replica* in subnet `P/C`, collateral must be staked for *replica* in the parent `P`'s IPC Subnet Actor $\texttt{P.ISA}_\texttt{C}$. Let the the account from which the collateral is transfered to $\texttt{P.ISA}_\texttt{C}$ be `P.a`. (Any user with sufficient account balance, i.e. at least *amount* and the transaction fee, can perform this operation.)

The child subnet, holding its own local copy of the target membership, is then informed (through an IPC agent) about the membership change and reconfigures to reflect it. Note that it is required for the child subnet to hold a copy of the membership in its replicated state, so that all its replicas observe it in a consistent way. In fact, the child subnet replicated state contains two versions of membership information:

- The *target membership*, which is a local copy of the membership stored in $\texttt{P.ISA}_\texttt{C}$ and designates the desired membership of the child subnet to which the subnet must reconfigure.
- The *current membership*, which is the actual membership currently being used by the subnet to order and execute transactions. Since the process of reconfiguration to a new membership is usually not immediate, the current membership may "lag behind" the target membership.

The whole staking procedure is as follows.

1. The owner of `P.a` submits the transaction
   tx = $\texttt{P.ISA}_\texttt{C}$.*StakeCollateral* (`P.a`, *replica*, *amount*).
2. After ordering *tx*, $\texttt{P.ISA}_\texttt{C}$ increases the collateral associated with *replica* by *amount*, which is deducted from the submitting user's account in `P`. If no collateral has been previously associated with *replica*, this effectively translates in *replica* joining the subnet.
3. The IPC agent, upon detecting the updated *membership* in $\texttt{P.ISA}_\texttt{C}$ through the application of *tx*, constructs a $PoF(tx)$ and submits the transaction
   tx' = `P/C.IGA.`*UpdateMembership* (*membership*, $PoF(tx)$).
4. Upon ordering *tx'* and successfully verifying $PoF(tx)$, `P/C.IGA` updates its target membership.
5. Subnet `P/C` reconfigures (the reconfiguration procedure is specific to the implementation of the subnet and its ordering protocol) to use the new *membership* as its current membership.

4.7.2 Releasing collateral

Releasing collateral works similarly (but inversely) to staking, with one significant difference. Namely, once the child subnet `P/C` reconfigures to reflect the updated membership, the parent's IPC Subnet Actor $\texttt{P.ISA}_\texttt{C}$ does not release the staked funds until it is given a proof that the child

subnet P/C finished the (subnet-specific) reconfiguration procedure. The following functions are involved in releasing collateral:

ISA.RequestCollateral(*replica, amount, account*)
IGA.UpdateMembership(*membership, PoF*)
ISA.ReleaseCollateral(*membership, PoF*)

Let P.a be an account that has staked collateral for a *replica* in a PoS-based subnet P/C. The procedure for releasing collateral is as follows.

1. The owner of P.a submits the transaction
   tx = $\texttt{P.ISA}_\texttt{C}$.*RequestCollateral* (*replica*, *amount*, a),
   where *amount* is the amount of funds the owner of P.a wants to reclaim.
2. After ordering *tx* and checking that P.a has indeed previously staked at least *amount* for *replica*, $\texttt{P.ISA}_\texttt{C}$ decreases the collateral associated with *replica* by *amount*. Note that P.ISA does not yet transfer *amount* back to P.a
3. The IPC agent, upon detecting the updated *membership* in $\texttt{P.ISA}_\texttt{C}$ through the application of *tx*, constructs a $PoF(tx)$ and submits the transaction
   tx' = P/C.IGA.*UpdateMembership* (*membership*, $PoF(tx)$).
4. Upon ordering *tx'* and successfully verifying $PoF(tx)$, P/C.IGA updates its target membership.
5. Subnet P/C reconfigures (the reconfiguration procedure is specific to the implementation of the subnet and its ordering protocol) to use the new *membership* as its current membership.
6. The IPC agent detects that the current membership of P/C has changed. Let *state* be the the replicated state of P/C where the current membership has already been updated.
7. The IPC agent constructs a $PoF(state)$ and submits the transaction
   tx" = $\texttt{P.ISA}_\texttt{C}$.*ReleaseCollateral* (*membership*, $PoF(state)$)
8. Upon ordering *tx"* and successfully verifying $PoF(tx)$, $\texttt{P.ISA}_\texttt{C}$ verifies that the received *membership* reflects the requested change in the collateral of *replica* and, if this is the case, transfers *amount* to P.a.

4.7.3 Slashing a misbehaving replica

Slashing is a penalty imposed on provably malicious replicas in PoS-based subnets. When a replica of a child subnet provably misbehaves, IPC agents can report the misbehavior to its parent subnet, which can take an appropriate (configured) action (e.g., confiscate a part of the replica's collateral). The definition of what constitutes a provable misbehavior is subnet-specific. An example of such misbehavior is sending equivocating messages in the subnet's ordering protocol, such as two conflicting proposals for the same block height. IPC exposes the following function to enable slashing:

ISA.*Slash*(*replica, PoM*)

Slashing a misbehaving *replica* of a PoS-based subnet P/C proceeds as follows:

1. The *replica* provably misbehaves, e.g., by sending two signed contradictory messages that would not have been sent if *replica* strictly followed its prescribed distributed protocol.

2. An IPC agent is informed of this misbehavior, e.g., by the replicas that received the contradictory messages, constructs a Proof of Misbehavior (PoM) (e.g., a data structure containing the two contradictory messages signed by *replica*), and submits the transaction $tx = \texttt{P.ISA}_\texttt{C}.\textit{Slash}(\textit{replica}, PoM)$

3. Upon ordering tx, $\texttt{P.ISA}_\texttt{C}$ evaluates the PoM against *replica* and adapts its associated collateral accordingly, resulting in a new membership for `P/C`.

4. Updating the membership of `P/C` and subsequent reconfiguration proceeds exactly as in Section 4.7.1, from Item 3 on.

Example 6. *Imagine the setting from Example 2, where a player creates a PoS-based subnet for a game of 4 players with a minimal collateral of 10 coins. Suppose that one of the players makes a move in the game, but later realizes that it would be beneficial to revert that move. What is more, the majority of other players would also benefit from the game state being reverted to the state just before the move occurred. Those players might collude and agree off-band to revert their replicas of the game server to an older state, and propose (in the child's ordering protocol) a different transaction to be ordered instead of the one containing the original game move.*

However, the system (concretely, the IPC Subnet Actor) is configured such that the initial proposal of the original transaction, together with the new proposal of the "replacement" transaction (both signed by the proposing replica), constitute a PoM that can be verified by the `ISA`. Any honest player that locally logs all proposals can thus submit a `P.ISA.Slash` transaction, with the offending replica and the PoM as arguments, receiving (in the parent subnet) a part of the collateral associated with the offending replica. Moreover, if the child subnet's protocol allows to prove that a replica supported (in some protocol-specific way – imagine signed PBFT prepare messages) more than one proposal for the same block height, the other colluding replicas can also be slashed.

5 IPC Actors

This section describes the state and functions of the two IPC actors: the `IGA` and the `ISA`.

5.1 IPC Gateway Actor (`IGA`)

The `IGA` is an actor that exists in every subnet in the IPC hierarchy and contains all information and logic the subnet itself needs to hold in order to be part of IPC. The functionality of the `IGA` described in Section 4 is summarized in Algorithm 1. The `IGA` holds:

- The names of its own, its parent's and its children's subnets
- The predicate used to evaluate the validity of Proofs of Finality. This predicate will be applied to PoFs from both the parent subnet and the child subnets. It is specific to the subnets (and the protocols they use) involved in interactions with this subnet.
- The postbox storing all the outgoing cross-net transactions, along with their routing metadata (original source and ultimate destination subnets). We model the postbox as an infinitely growing set, from which the appropriate IPC agents select only those elements that need to be submitted to other subnets. A garbage-collection mechanism for deleting delivered outgoing cross-net transactions from the sender subnet's state is out of the scope

of this document. One can imagine, however, a garbage-collection mechanism based on acknowledgments (that are themselves cross-net transactions).

- In a PoS-based subnet whose membership is managed by its parent, the `IGA` also contains the target membership that the subnet must reconfigure to (if the subnet is not using it yet). This membership is the subnet's local copy of the membership stored in its corresponding IPC Subnet Actor in the parent. It must be part of the subnet's replicated state, so that its replicas have a consistent view of it and can correctly reconfigure. Since reconfiguration does not happen immediately, the actual membership (also part of the subnet's replicated state) lags behind the target membership.

Algorithm 1: IPC Gateway Actor (`IGA`)

```
1  ownSubnetName: name of the subnet the IGA resides in
2  parentSubnetName: name of the parent subnet
3  childSubnets: set of subnet names, initially empty
4  valid: predicate over a PoF defining its validity criteria
5  postbox: set of tuples (transaction, source, destination), initially empty
6  targetMembership: the membership this subnet should reconfigure to if it is not yet using it (PoS only)
7
8  CreateChild(name)
9      childSubnets = childSubnets ∪ {name}
10 RemoveChild(name)
11     childSubnets = childSubnets \ {name}
12 MintDeposited(amount, account, PoF)
13     if valid(PoF) then
14         mint amount new coins
15         transfer minted coins to account
16 Withdraw(amount, account)
17     if account.balance ≥ amount then
18         Burn amount coins from account
19 Dispatch(tx, src, dest)
20     postbox = postbox ∪ {(tx, src, dest)}
21 Propagate(tx, src, dest, PoF)
22     if valid(PoF) then
23         if dest = ownSubnetName then
24             execute tx
25         else if ∃s ∈ childSubnets ∪ {parentSubnetName} : s is part of dest then
26             Propagate(tx, src, dest)
27 UpdateMembership(membership, PoF)
28     if valid(PoF) then
29         targetMembership = membership
```

5.2 IPC Subnet Actor (`ISA`)

The IPC Subnet Actor (`ISA`) is the actor in the parent subnet's replicated state that governs a single child subnet. It stores all information about the child subnet that the parent needs and logic that manipulates it. The `ISA` is registered in the IPC hierarchy by invoking the parent's `IGA.CreateChild`(*subnetName*) function (see Section 4.1). The functionality of the `ISA` described in Section 4 is summarized in Algorithm 2. The `ISA` holds:

- The predicate (*valid*) used to evaluate the validity of Proofs of Finality of the child subnet's replicated state. It is specific to the child subnet and the protocol it uses, and its definition is part of *params* passed to `IGA.CreateChild` when the `ISA` is created.
- The amount of funds that are locked for use in the child subnet (*lockedFunds*). Deposits increase and withdrawals decrease this value accordingly. Keeping track of this value is only necessary for enforcing the firewall property, since a misbehaving child subnet might claim to withdraw more than has been deposited in it. Thus, before withdrawing, the `ISA` consults this value to make sure that the total amount of withdrawals never exceeds the amount previously deposited.
- Snapshots of the child subnet's replicated state obtained through invocations of the `Checkpoint` function (*checkpoints*).
- If the child subnet is a PoS-based one, the `ISA` also contains state required for managing the subnet's membership and the associated collaterals. The high-level implementation presented in Algorithm 2 presents a simplified view of this state and the associated logic, as it conveys the mechanisms involved without getting lost in details. In particular, the presented description neglects some corner cases arising from concurrent handling of multiple staking, releasing, and/or slashing procedures. In a real-world implementation, however, these corner cases can easily be addressed.

  The `ISA` stores information on which child replica has how much collateral (*childMembership*), how much collateral (and for which replica) is staked from which account (*collateral*), and which accounts requested the withdrawal of how much collateral (*collateralRequests*). Moreover, the `ISA`'s state contains a predicate for checking the validity of proofs of misbehavior (*validPom*) and a procedure to execute when a valid PoM is received through the `Slash` function.

6 Related Work

We categorize related work in the following subcategories of approaches to scale blockchains: novel consensus protocols, rollups, channel networks, and subnet networks (like IPC). Note however that these categories are not inherently competing, but rather potentially orthogonal solutions that, carefully combined, could scale even beyond what one of them, in isolation, is capable of.

6.1 Consensus protocols

Recent developments in asynchronous consensus protocols have significantly increased throughput by decoupling data dissemination from metadata ordering [41, 55, 32, 46, 44, 45, 2, 8, 17, 57]. These advances are inherently restricted by the need to replicate all state updates across all members of the consensus protocol. Furthermore, their improvements are orthogonal to the horizontal scaling proposed by IPC, in that each subnet can benefit from these advances at their consensus level.

6.2 ZK and optimistic rollups

Rollups scale blockchains by having third parties (sequencers) locally order and execute batches of transactions and submitting only the result of the batch in the blockchain. Optimistic rollups [40, 4] rely on the result of a sequencer allowing for a predefined dispute period. If

Algorithm 2: IPC Subnet Actor (`ISA`)

```
1  valid: predicate over a PoF defining its validity criteria
2  lockedFunds: total amount of funds circulating in the child subnet
3  checkpoints: set of checkpoints of the child's replicated state
4  childMembership: map of replica identities to their respective staked collaterals
5  collateral: map of accounts to replica identities, to staked collaterals
6  collateralRequests: set of received but unsatisfied requests for releasing collateral
7  validPoM: predicate over a PoM defining its validity criteria
8  slashingPolicy: procedure to execute on reception of a valid PoM
9
10 Deposit(amount, account)
11     lockedFunds += amount
12 ReleaseWithdrawn(amount, account, PoF)
13     if valid(PoF) ∧ lockedFunds ≥ amount then
14         lockedFunds −= amount
15         transfer amount to account
16 Checkpoint(snapshot, PoF)
17     if valid(PoF) then
18         checkpoints = checkpoints ∪ {snapshot}
19 StakeCollateral(account, replica, amount)
20     childMembership[replica] += amount
21     collateral[account][replica] += amount
22 RequestCollateral(replica, amount, account)
23     if collateral[account][replica] ≥ amount then
24         childMembership[replica] −= amount
25         collateralRequests = collateralRequests ∪ {(amount, account)}
26 ReleaseCollateral(membership, PoF)
27     if valid(PoF) ∧ membership = subnetMembership then
28         for (amount, account) ∈ collateralRequests do
29             transfer amount to account
30 Slash(replica, PoM)
31     if valid(PoM) then
32         slashingPolicy(PoM)
```

disputed, the sequencer is not punished only if it proves within a limited amount of time, via executing the batch in the blockchain, that the result of its batch matches its submitted result. ZK rollups [49, 1, 59, 56, 15, 11, 1, 6] cryptographically generate a succinct proof of the correctness of the result for fast verification. Unfortunately, most ZK rollups proposed to date rely on centralized sequencers, with some novel works working on decentralizing sequencers [6]. In addition, rollups that post transaction input data on L1 inherit throughput scalability limitations of an L1. Some alternative approaches such as Validium [21], combine ZK rollups with input data storage off-L1, for better scalability.

Compared to rollups, IPC to a certain extent resembles a combination of Validium and optimistic rollups in a sense that IPC subnets leverage parent subnets for increased security, while storing transaction data off parent subnet, i.e., in a child subnet itself, replicated with a BFT protocol. IPC currently does not use ZK technology for transaction execution in subnets; however, such support is not precluded in future. In particular, our group aims at exploring integration of Lurk ZK execution framework [24] in IPC subnets.

6.3 Sharding

Sharding [66, 42, 47, 65, 26, 25, 63, 23, 39] approaches periodically partition blockchain state across different groups of replicas. While sharding has shown promise in increasing scalability, it also introduces new challenges such as ensuring cross-shard communication and preventing data fragmentation. Unlike subnets, shards do not conceptually differ from each other, leading to a potential increase of cross-shards communication compared to subnets, where users and services partition the state on demand.

6.4 State and payment channels

State channels [34] scale blockchains by having participants lock their state in the blockchain (on-chain) among them in order to update locally the state (off-chain) with which to release the funds back on-chain. Payment channels [51, 33, 29] are the analogous solutions for channels where the state are coins. Channels can form network graphs in which nodes can relay payments (or general state) in exchange of a fee [54, 53, 20, 35, 22]. Though many works, particularly in state channels, are addressing significant problems specific to channels, IPC subnets are a generalization of channels. This is because channels require all parties to sign in order to update the channel's state off-chain, whereas subnets (like the ones we show in this document) allow for a variable quorum size, or even different types of proofs of finality.

6.5 Subnet and sidechain networks

There are a number of previous and concurrent works that are conceptually similar to IPC subnets.

Single-layer networks. Polkadot's relay-chain connects Polkadot's parachains [13] with native cross-net transactions, in a topology that also resembles a single scaling layer of direct child subnets of a rootnet. They also offer specialized bridges to other blockchains, like that of Bitcoin or Ethereum. Polkadot's parachains rely on anchoring their compressed state to the parent for total ordering and cross-net interactions, which limits the capabilities of horizontal scaling: parachains need to constantly lease an auctioned slot on the relay-chain for a specific period, a problem that is exacerbated in a single layer of child subnets checkpointing to the same parent, compared to IPC's tree topology. Polygon [14, 18] scales the Ethereum blockchain by using a network of subnets (sidechains in the Polygon terminology) connected to the Ethereum mainnet, with a developing framework for services to create their own subnet. Yet, Polygon PoS offers one subnet, and Polygon does not offer a native cross-net communication protocol. Avalanche offers a similar solution for its rootnet with Avalanche subnets [16], a network of subnets that are connected to their rootnet for further scaling and customized applications. Internet Computer Protocol's (ICP) subnets [9] are independent blockchains with the ICP rootnet as the common parent, and whose committees are subcommittees of the rootnet's validator set. ICP subnets are tightly integrated with its rootnet and dedicated to execute smart contracts seamlessly. Unfortunately, none of these works consider a structure beyond a single layer of direct child with their respective rootnet as the only parent.

Topology-agnostic networks. The concept of sidechains [28, 36] precedes that of subnets. In particular, IPC parent-child interaction design and IPC trust model, draws a lot of inspiration from PoS sidechain formalization [36]. A sidechain targets crosschain payments not necessarily in the parent-child hierarchy. Cosmos zones [5] are independent blockchains that interact in a general graph not necessarily in the topology of a tree, with a reference implementation that

spawns new zones running Tendermint as the consensus protocol. Cosmos relies on the single finality of Tendermint for inter-blockchain communication (IBC). Cosmos zones can alternatively use the Cosmos hub, a zone that specializes in facilitating cross-net interactions, similarly to the Polkadot relay-chain. The flexibility for any topology between subnets comes with the need to have further requirements for IBC, like having nodes of a blockchain run light clients of each other blockchain they want to interact with.

UTXO-based subnets. Plasma chains [50] horizontally scale payments via multiple subnets (childchains) organized in a tree-like structure. It is, one of the first works to consider subnet-like interactions between blockchains. Given the topology, many of the challenges that IPC subnets face are present in Plasma chains. However, Plasma chains are tree-based subnets focused on payments. In this sense, IPC subnets generalize the problem that Plasma chains are trying to solve, allowing state to be transacted in subnets, and not just payments. Also, Plasma chains rely on synchrony for withdrawals offering a dispute resolution period, similarly to channels and optimistic rollups, posing an irreconcilable trade-off between finality latency and security [52], with Platypus [52] approaching the PoF shown for IPC subnets running Trantor as the first fix to this trade-off. IPC subnets also generalize withdrawal mechanisms by allowing subnets to define what constitutes a PoF, a slashing rule and a fraud proof.

7 IPC's reference implementation

The reference implementation of IPC differs slightly from the description of the functionality and implementation shown in previous sections, partly due to the rapidly changing development of the system, and partly due to its concurrent implementation simultaneous with the design and improvement of the system. In this section, we describe the particular implementation choices for the reference implementation of IPC. It is not the purpose of this section to comprehensively describe the reference implementation, but to list the relevant differences of the current reference implementation compared with the description of IPC made in previous sections, as well as to list the pertinent implementation decisions of abstractions such as the PoF or the consensus protocol used by the reference implementation.

7.1 Preliminaries

The current implementation considers Filecoin [7] as the rootnet and Trantor [19] running in child subnets, both running as the consensus layer of the Lotus blockchain client [10]. Filecoin is a Proof-of-Storage, longest-chain-style protocol with probabilistic finality. For our purposes, we note that Trantor is a BFT-style protocol that iterates through instances of PBFT [31] with immediate finality. Every decided block in Trantor contains an ordered list of decided transactions and a certificate for verification, with every Δ-th block containing a checkpoint of the state. At the moment, the checkpoint being generated by Trantor is a certificate of the latest decided block provided by the Lotus client.

The reference implementation makes use of IPFS-style content addressing, in that data is stored where relevant and referred to with a Multiformats-compliant [12] content identifier (CID) elsewhere. In particular, CIDs that address information of a specific child's subnet can be used to retrieve the content through BitSwap [3] from any of the participants running full nodes of the subnet. This means that if a subnet only has faulty participants, the content referred to by this CID may not be available. However, this is not a problem, as IPC still preserves the subnet firewall property (Section 3).

The reference implementation uses the Filecoin Virtual Machine (FVM) as the runtime environment in which the `IGA` and the `ISA` are deployed. Both actors are user-defined, in that any user can deploy their own modifications of the provided actors, and use them to interact with the rest of the IPC hierarchy.

7.2 Components

The IPC reference implementation preserves all the components described in Section 5 without additions. We however list here implementation decisions concerning these components. The two main design decisions are (*i*) to have one IPC agent manage the interactions across all subnets, and not one per parent-child pair, and (*ii*) to make the `IGA` the entry point for all IPC functionality, with the possibility to augment the default functionality for a specific subnet with the `ISA`.

7.2.1 IPC agent

In the previous sections, we considered that every parent-child pairing had an independent IPC agent process. In fact, the implementation manages to execute one single IPC agent for the entire tree of subnets that may be of relevance to a participant. This process can be executed either as a daemon or as a command-line tool. In the latter case, the IPC agent cannot participate in either checkpointing or propagating cross-net transactions. We refer to the participants running the IPC agent as a command-line tool as the IPC clients.

While a participant only runs one IPC anget, it also runs a full node for each subnet that the participant is involved in. The IPC agent process along with all the full nodes relevant to a participant conform the participant's *trust domain*. All processes within the same trust domain assume each other's correctness. As a result, the IPC agent can be notified of changes to the state of each of the full nodes locally run by the participant.

7.2.2 `IGA` as entry point

In the reference implementation, the entry point for all functionality is the `IGA`, unlike in the high-level functionality described in Section 4. For any of the provided functionalities, the IPC agent submits transactions to the `IGA` (e.g. `IGA.`*`Deposit`*`(C, `*`amt`*`, ...)`). For bottom-up transactions, the child's `IGA` communicates with the `ISA` of the child at the parent. For top-down transactions, the `IGA` of the parent directly communicates with the `IGA` at the child. Nonetheless, the IPC agent never communicates with the `ISA` directly, but indirectly through an `IGA`.

As a result, in the reference implementation, the `IGA` contains the locked funds of each subnet, i.e. the subnet's *circulating supply* (unlike in Section 5, where the `ISA` held the locked funds). The circulating supply of each child subnet is stored in a map at the `IGA`, where the key is the subnet ID. As withdrawals contain a PoF, the circulating supply suffices for the firewall property.

7.2.3 `ISA` for subnet customization

In the reference implementation, $\texttt{ISA}_{\texttt{S}}$ holds the state specific to subnet `S`. However, the aforementioned entry point for all functionality is the `IGA`, and not the `ISA`. A subnet can augment the default functionality of the `IGA` in the `ISA`. In particular, the `ISA` can include conditions for

the validation of proofs of finality, releases of funds and stake, and slashing rules[10].

7.3 Functionality

In this section, we describe the implementation of the functionality. In the reference implementation, cross-net transactions are the cornerstone of interactions between IPC subnets. Deposits, withdrawals, staking and releasing collateral, and state changes across subnets are all implemented with cross-net transactions.

7.3.1 Deposits

The main difference of deposits in the reference implementation compared to the high-level description is that $\texttt{ISA}_{\texttt{C}}$ does not hold the funds being deposited to subnet `C`. Additionally, the reference implementation explicitly addresses incentives by requiring an IPC fee in each cross-net transaction. In particular, depositing *amount* coins from an account `P.a` in the parent subnet `P` to an account `P/C.b` in the child subnet `P/C` is performed in the following steps:

1. The owner of `P.a` submits a transaction tx = `P.IGA.`*`Deposit`*(`P/C`, *b, amount, IPCfee*).
2. tx is ordered and executed at `P`. The ordering and execution of tx is as follows:
   - `IGA` checks that *IPCfee* is above a hard-coded minimum IPC base fee. The parameter *IPCfee* is an amount of coins to be paid to the child replicas to incentivize them to participate in the validation of top-down transactions for the child[11] (see Section 7.3.4).
   - *amount* is deposited in the `IGA` of the parent subnet.
   - `IGA` creates a top-down transaction tx' = `P/C.IGA.`*`MintDeposited`*(*b, amount, IPCfee*) and stores it in the top-down registry (see Section 7.3.4).
   - The top-down transaction tx' is ordered and executed at the child subnet, resulting in the minting of *amount* sent to account `P/C.b`. We detail further the ordering and execution of top-down transactions in Section 7.3.4.

7.3.2 Withdrawals

Analogously to deposits, withdrawals must carry an IPC fee to be paid to the child replicas, and the funds to be released back at the parent subnet `P` are being held at `P.IGA`. Otherwise, the procedure is analogous to the one described in Section 4. More concretely, withdrawing *amount* coins from an account `P/C.b` in the child subnet `P/C` to an account `P.a` in the parent subnet `P` involves the following steps:

- The owner of `P/C.b` submits a transaction tx = `P/C.IGA.`*`Withdraw`*(*amount, a, IPCfee*).
- tx is ordered and executed at `P/C`. The ordering and execution of tx is as follows:
  - `IGA` checks that *fee* is above a hard-coded minimum IPC base fee. The parameter *IPCfee* is an amount of coins to be paid to the child replicas to incentivize them to participate in the validation of bottom-up transactions for the child (see Section 7.3.4).

[10] the reference implementation does not provide any slashing rule at the time of writing, but provides the mechanism to define slashing rules in the `ISA`.

[11] The IPC fee is different from and in addition to transaction fees for replicas to order and execute transactions in a subnet, which we mention in Section 3 but otherwise omit in each transaction throughout the document.

- *amount* is burned from `b`.
- `IGA` creates a bottom-up transaction tx' = `P.ISA`$_C$.*`ReleaseWithdrawn`*(*amount, b, IPCfee*) and stores it in the bottom-up registry (see Section 7.3.4).
- The bottom-up transaction tx' is ordered and executed at the parent subnet. This results in `ISA`$_C$ calling `IGA` to release *amount* and send it to account `P.a`. We detail further the ordering and execution of bottom-up transactions in Section 7.3.4.

7.3.3 Checkpointing

A checkpoint of a subnet `P/C` is triggered every Δ blocks decided at the child subnet. If the latest block decided meets this condition, and if the participant's full node is a replica according to the state stored at the parent, then the IPC agent starts computing the checkpoint as follows:

1. The IPC agent obtains a state snapshot from the child's subnet. The state snapshot is a CID *chkpCID* of the latest decided block of the child's subnet that contains a checkpoint certificate as *PoF* (recall that every Δ-th block contains a checkpoint certificate in Trantor). The *PoF* of the checkpoint is the certificate of the block.

2. The IPC agent obtains the CIDs of all new grandchildren's checkpoints stored at `P/C.IGA.gcChkps` and of the bottom-up transactions in the bottom-up registry `BUpTxs` (see Section 7.3.4). Explicitly checkpointing children checkpoints recursively bubbles up the security anchor from lower levels of the hierarchy.

3. The IPC agent submits *tx* =`P.ISA`$_C$.Checkpoint(`P/C`, *chkpCID, gcChkps, BUpTxs*, *PoF*).

4. The `ISA`$_C$ verifies the validity of the *PoF*, saves the checkpoint CID in its state and calls on `IGA` to save all checkpoints' CIDs (those of `P/C`'s children and of `P/C`) and to execute all bottom-up transactions attached to the checkpoint. If the *PoF* is not valid according to the state at `ISA`$_C$, then the entire checkpoint will fail.

7.3.4 Propagating cross-net transactions

Similarly to the description in Section 4, the current reference implementation uses the postbox in each `IGA` to propagate cross-net transactions to a subnet not immediately adjacent by submitting multiple cross-net transactions in the parent-child hierarchy (one in each subnet along the path to the destination subnet). As shown in Section 4, a cross-net transaction *tx* is propagated to each immediately adjacent subnet along the path to its destination by traversing through the postbox of all intermediate subnets, via cross-net transactions $tx'(tx)$ containing *tx* as payload.

However, once tx' is ordered and executed at an intermediate subnet `S`, an IPC agent must pay for the cost to pay the fees for ordering and executing another transaction $tx''(tx)$ in `S` so as to move the state to the next subnet along the path. For this reason, the reference implementation leaves *tx* in the postbox of that intermediate subnet until the account that originally triggered the cross-net transaction creates tx'' and pays for the fees required to execute it[12]. This process is repeated until *tx* reaches its destination subnet.

As a result, a cross-net transaction that has been paid for leaves the postbox to join a FIFO queue, known as either *the bottom-up registry* or *top-down registry*. All three, postbox, bottom-up registry and top-down registry, contain cross-net transactions and are part of the state of `IGA`, but only those transactions in either top-down registry or bottom-up registry are

[12] a whitelist of accounts that are allowed to create and pay can be provided.

propagated. Transactions in the top-down registry (resp. bottom-up registry) at `P.IGA` are top-down (resp. bottom-up) transactions.

Top-down transactions. In order to prevent inconsistencies across replicas, the IPC agent does not immediately submit a top-down transaction to the child subnet. Instead, IPC agents consistently broadcast the top-down transactions they consider as final at the parent subnet. Also, as an optimization, the IPC agent batches top-down transactions in a *top-down checkpoint* that is consistently broadcast to other replicas every Δ_T blocks. In this consistent broadcast, an IPC agent broadcasts batches of top-down transactions that it locally considers as valid, and it in turn signs a received batch if it considers all transactions of the batch as valid. As such, the *PoF* of a top-down transaction tx is obtained once enough signatures to form a certificate[13] are received for a batch containing tx.

A participant running a straggling parent full node that receives a certificate for a batch as *PoF*, but that does not locally see all transactions of the batch as valid, can instead verify the *PoF*. Once the IPC agent verifies a certificate for a batch of transactions, the IPC agent submits the batch to the child subnet for ordering and execution.

Bottom-up transactions. The child subnet aggregates bottom-up transactions attaching their corresponding CIDs to the next checkpoint, along with an increasing nonce value per CID that is unique for each parent-child pair[14]. The CIDs of these bottom-up transactions are placed in the `IGA` of the child. Bottom-up transactions are stored in the `IGA` of the child until it is time to checkpoint to the parent (see Section 7.3.3). This way, the `IGA` serves as the single location for the CIDs of bottom-up transactions and the IPC agent only needs to monitor the `IGA` to get all necessary information from the child subnet. As shown in Section 7.3.3, since the CIDs of bottom-up transactions are attached to the checkpoint, the propagation of bottom-up transactions depends on the validity and execution of the checkpoint transaction.

7.3.5 Creating and removing a child subnet

Analogously to Section 4, subnets are created by instantiating a new `ISA` and registering the `ISA` in `IGA`. When the `IGA` contains a minimum amount of collateral stored associated with `ISA` (where enough is defined in the `IGA`), the subnet can be registered in `IGA`.

A subnet `P/C` is removed from IPC in three steps. First, all users must withdraw their funds to set the circulating supply to zero. Second, all validators leave and release their collateral. Third, any account sends a `P.IGA.`*`kill`*`(`*`C`*`)` transaction to the `IGA` that marks the subnet as removed.

7.3.6 Staking and releasing collateral

Contrary to Section 4, staking collateral does not require a explicit call to update any state in the `IGA` of the child subnet. A replica simply increases its stake with account `P.a` by submitting a transaction tx = `P.IGA.`*`StakeCollateral`*`(P.a,` *replica*, *amount*`)`. This transaction stakes the collateral in the `IGA` of the parent `P`. Notwithstanding, it does not incur a reconfiguration in the weights of the replica set running the Trantor protocol at the child subnet `P/C`. Instead, it is up to the current replica set at `P/C` to decide when and how to update their membership to reflect this change of the relative voting power, and notify the new reconfiguration to `P.IGA` with a bottom-up transaction.

[13] In Trantor, a certificate for the batch consists of at least enough replicas containing a supermajority of the voting power sign the batch.

[14] the nonce value is necessary to prevent replay attacks

A reconfiguration transaction $tx = \texttt{P.ISA}_\texttt{C}\texttt{.}\textit{UpdateMembership}\,(membership, PoF)$ being submitted to the parent can also trigger a release of collateral $tx = \texttt{P.IGA.}\textit{ReleaseCollateral}\,(membership)$ to reflect this update in the weighted voting of the child subnet. It is possible that a reconfiguration triggers a release of collateral that drops the staked collateral below the minimum threshold for subnet creation. IPC subnets of the reference implementation must always hold this minimum amount of collateral per subnet (defined in the `IGA`). If, after it has been created and registered, the subnet's collateral drops below the required minimum, then the subnet enters an *inactive* state. This means that the subnet can no longer interact with the rest of the active subnets registered in the `IGA`, or even withdraw funds back to the parent subnet. In this case, though, users and remaining replicas can stake enough collateral to reactivate the subnet.

7.4 Incentives

In the current reference implementation, replicas get rewarded for executing the checkpoint algorithm and participating in the top-down checkpoint by charging an IPC fee on all cross-net transactions. This incentivizes replicas to participate, even if that costs them a fee to be paid for the transaction at the parent. All cross-net transactions must contain a fixed amount known as IPC fee (on top of the standard transaction fee required for ordering and execution of the transaction in the corresponding subnet). The IPC fee is only paid once the checkpoint (or top-down checkpoint) is ordered and executed. However, no specific incentives are given at the moment to the submitter(s) of the checkpoint transaction, or to the signers of the certificate, in that the sum of IPC fees of the batch is evenly distributed proportionally to the stake of each replica in the membership. This can lead to equilibria in which some participants are incentivized neither to submit checkpoints nor participate in generating a PoF. We are currently working in more complex incentive-compatible mechanisms that ensure rational participants will follow the protocol for the reference implementation, via rewards and slashing.

8 Conclusion

The scalability challenge posed by the need of consensus in blockchain networks has led to the exploration of L2 and subsequent layers. While there has been an explosion of L2 solutions in recent years, most of these lack of a non-monolithic design that allows for native communication across state partitions, on-demand horizontal scaling, decentralization, and recursive security anchoring with the root layer. We presented in this document Interplanetary Consensus (IPC), a blockchain architecture based on web-scale on-demand horizontal scalability through a hierarchical subnet structure. IPC leverages the security of parent subnets to benefit child subnets and allows subnets to be used in different ways, such as hosting different applications or sharding a single application, or even user-motivated state partitions that are not necessarily immediate at the time of writing. As a result, we believe IPC represents a significant step forward in the quest for L2+ scaling, providing for a sensible, flexible design that balances decentralization, scalability, security, and usability at relatively short timescales.

References

[1] Applied zkp. `https://appliedzkp.org/`. Accessed: March 24, 2023.

[2] Aptos labs. `https://aptoslabs.com/`. Accessed: March 24, 2023.

[3] Bitswap — ipfs docs. `https://docs.ipfs.tech/concepts/bitswap/`. Accessed on March 24, 2023.

[4] Celestia documentation - rollmint. https://docs.celestia.org/developers/rollmint/. Accessed: March 24, 2023.

[5] Cosmos network documentation. https://docs.cosmos.network/main. Accessed: March 24, 2023.

[6] Espresso. https://www.espressosys.com/. Accessed: March 24, 2023.

[7] Filecoin. https://filecoin.io/. Accessed: March 24, 2023.

[8] Hedera hashgraph. https://hedera.com/. Accessed: March 24, 2023.

[9] How it works - internet computer. https://internetcomputer.org/how-it-works. Accessed: March 24, 2023.

[10] Lotus. https://lotus.filecoin.io/. Accessed on March 24, 2023.

[11] Matter labs. https://matter-labs.io/. Accessed: March 24, 2023.

[12] Multiformats. https://multiformats.io/. Accessed on March 24, 2023.

[13] Parachains — polkadot. https://polkadot.network/parachains/. Accessed: March 24, 2023.

[14] Polygon pos. https://polygon.technology/solutions/polygon-pos. Accessed: March 24, 2023.

[15] Sin7y. https://sin7y.org/. Accessed: March 24, 2023.

[16] Subnets — avalanche. https://www.avax.network/subnets. Accessed: March 24, 2023.

[17] Sui. https://sui.io/. Accessed: March 24, 2023.

[18] Supernets. https://polygon.technology/supernets. Accessed: March 24, 2023.

[19] Trantor: Modular consensus for (not only) filecoin ipc. https://hackmd.io/P59lk4hnSBKN5ki5OblSFg?view. Accessed: March 24, 2023.

[20] The raiden network, 2019.

[21] Validium, 2023.

[22] ACINQ. Eclair github repository, 2008.

[23] Mustafa Al-Bassam, Alberto Sonnino, Shehar Bano, Dave Hrycyszyn, and George Danezis. Chainspace: A sharded smart contracts platform, 2017.

[24] Nada Amin, John Burnham, François Garillot, Rosario Gennaro, Chhi'mèd Künzang, Daniel Rogozin, and Cameron Wong. Lurk: Lambda, the ultimate recursive knowledge. Cryptology ePrint Archive, Paper 2023/369, 2023. https://eprint.iacr.org/2023/369.

[25] Elli Androulaki, Christian Cachin, Angelo De Caro, and Eleftherios Kokoris-Kogias. Channels: Horizontal scaling and confidentiality on permissioned blockchains. In Javier Lopez, Jianying Zhou, and Miguel Soriano, editors, *Computer Security*, pages 111–131, Cham, 2018. Springer International Publishing.

[26] Georgia Avarikioti, Eleftherios Kokoris-Kogias, and Roger Wattenhofer. Divide and scale: Formalization of distributed ledger sharding protocols, 2021.

[27] Sarah Azouvi and Marko Vukolić. Pikachu: Securing pos blockchains from long-range attacks by checkpointing into bitcoin pow using taproot. In *Proceedings of the 2022 ACM Workshop on Developments in Consensus*, ConsensusDay '22, page 53–65, New York, NY, USA, 2022. Association for Computing Machinery.

[28] Adam Back, Matt Corallo, Luke Dashjr, Mark Friedenbach, Gregory Maxwell, Andrew Miller, Andrew Poelstra, Jorge Timón, and Pieter Wuille. Enabling blockchain innovations with pegged sidechains, 2014.

[29] Conrad Burchert, Christian Decker, and Roger Wattenhofer. Scalable funding of bitcoin micropayment channel networks. In *Royal Society open science*, 2018.

[30] Vitalik Buterin. Why sharding is great: demystifying the technical properties, 2021.

[31] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OsDI*, volume 99, pages 173–186, 1999.

[32] George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. Narwhal and tusk: a dag-based mempool and efficient BFT consensus. In Yérom-David Bromberg, Anne-Marie Kermarrec, and Christos Kozyrakis, editors, *EuroSys '22: Seventeenth European Conference on Computer Systems, Rennes, France, April 5 - 8, 2022*, pages 34–50. ACM, 2022.

[33] Christian Decker and Roger Wattenhofer. A fast and scalable payment network with bitcoin duplex micropayment channels. In *Symposium on Self-Stabilizing Systems*, 2015.

[34] Stefan Dziembowski, Sebastian Faust, and Kristina Hostáková. General state channel networks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 949–966, New York, NY, USA, 2018. Association for Computing Machinery.

[35] Elements Project. c-lightning – a lightning network implementation in C. Accessed on March 24, 2023.

[36] Peter Gaži, Aggelos Kiayias, and Dionysis Zindros. Proof-of-stake sidechains. In *S&P*, 2019.

[37] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling Byzantine agreements for cryptocurrencies. In *SOSP*, 2017.

[38] Lewis Gudgeon, Pedro Moreno-Sanchez, Stefanie Roos, Patrick McCorry, and Arthur Gervais. Sok: Layer-two blockchain protocols. In Joseph Bonneau and Nadia Heninger, editors, *Financial Cryptography and Data Security*, pages 201–226, Cham, 2020. Springer International Publishing.

[39] Zicong Hong, Song Guo, Peng Li, and Wuhui Chen. Pyramid: A layered sharding blockchain system. In *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications*, pages 1–10, 2021.

[40] Harry Kalodner, Steven Goldfeder, Xiaoqi Chen, S. Matthew Weinberg, and Edward W. Felten. Arbitrum: Scalable, private smart contracts. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1353–1370, Baltimore, MD, August 2018. USENIX Association.

[41] Idit Keidar, Eleftherios Kokoris-Kogias, Oded Naor, and Alexander Spiegelman. All you need is DAG. In Avery Miller, Keren Censor-Hillel, and Janne H. Korhonen, editors, *PODC '21: ACM Symposium on Principles of Distributed Computing, Virtual Event, Italy, July 26-30, 2021*, pages 165–175. ACM, 2021.

[42] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. Omniledger: A secure, scale-out, decentralized ledger via sharding. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 583–598, 2018.

[43] L2BEAT Team. L2beat: Ethereum layer 2s analytics. `https://l2beat.com/`, 2021. Accessed: March 30, 2023.

[44] Yuan Lu, Zhenliang Lu, and Qiang Tang. Bolt-dumbo transformer: Asynchronous consensus as fast as the pipelined BFT. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, pages 2159–2173. ACM, 2022.

[45] Yuan Lu, Zhenliang Lu, and Qiang Tang. Bolt-dumbo transformer: Asynchronous consensus as fast as the pipelined BFT. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, pages 2159–2173. ACM, 2022.

[46] Yuan Lu, Zhenliang Lu, Qiang Tang, and Guiling Wang. Dumbo-mvba: Optimal multi-valued validated asynchronous byzantine agreement, revisited. In Yuval Emek and Christian Cachin, editors, *PODC '20: ACM Symposium on Principles of Distributed Computing, Virtual Event, Italy, August 3-7, 2020*, pages 129–138. ACM, 2020.

[47] Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. A secure sharding protocol for open blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 17–30, New York, NY, USA, 2016. Association for Computing Machinery.

[48] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, Dec 2008.

[49] Kamilla Nazirkhanova, Joachim Neu, and David Tse. Information dispersal with provable retrievability for rollups. *arXiv preprint arXiv:2111.12323*, 2021.

[50] Joseph Poon and Vitalik Buterin. Plasma: Scalable autonomous smart contracts, 2017.

[51] Joseph Poon and Thaddeus Dryja. The bitcoin lightning network: Scalable off-chain instant payments, 2016.

[52] Alejandro Ranchal-Pedrosa and Vincent Gramoli. Platypus: Offchain protocol without synchrony. In *IEEE NCA*, 2019.

[53] Andrew Samokhvalov, Joseph Poon, and Olaoluwa Osuntokun. Basis of lightning technology (BOLTs), 2018.

[54] Andrew Samokhvalov, Joseph Poon, and Olaoluwa Osuntokun. The lightning network daemon, 2018.

[55] Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. Bullshark: DAG BFT protocols made practical. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, pages 2705–2718. ACM, 2022.

[56] StarkWare. Starkware. `https://starkware.co/`, 2022. Accessed: March 24, 2023.

[57] Chrysoula Stathakopoulou, Matej Pavlovic, and Marko Vukolic. State machine replication scalability made simple. In Yérom-David Bromberg, Anne-Marie Kermarrec, and Christos Kozyrakis, editors, *EuroSys '22: Seventeenth European Conference on Computer Systems, Rennes, France, April 5 - 8, 2022*, pages 17–33. ACM, 2022.

[58] ConsensusLab Research Team. IPC Glossary. `https://docs.google.com/document/d/15pA7ahjeA-HYOl8Pxj0n6PxEswYlRVrZ112MJuRR0fY/edit?usp=sharing`.

[59] Polygon Technology. Polygon launches zkevm, a new ethereum-compatible scaling solution. `https://polygon.technology/polygon-zkevm`, 2022. Accessed: March 24, 2023.

[60] Louis Tremblay Thibault, Tom Sarry, and Abdelhakim Senhaji Hafid. Blockchain scaling using rollups: A comprehensive survey. *IEEE Access*, 10:93039–93054, 2022.

[61] Marko Vukolić. The quest for scalable blockchain fabric: Proof-of-work vs. BFT replication. In Jan Camenisch and Dogan Kesdogan, editors, *Open Problems in Network Security - IFIP WG 11.4 International Workshop, iNetSec 2015, Zurich, Switzerland, October 29, 2015, Revised Selected Papers*, volume 9591 of *Lecture Notes in Computer Science*, pages 112–125. Springer, 2015.

[62] Marko Vukolić. On the future of decentralized computing. *Bull. EATCS*, 135, 2021.

[63] Jiaping Wang and Hao Wang. Monoxide: Scale out blockchains with asynchronous consensus zones. In *NSDI*, volume 2019, pages 95–112, 2019.

[64] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. In *Ethereum Project Yellow Paper*, 2014.

[65] Guangsheng Yu, Xu Wang, Kan Yu, Wei Ni, J. Andrew Zhang, and Ren Ping Liu. Survey: Sharding in blockchains. *IEEE Access*, 8:14155–14181, 2020.

[66] Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova. Rapidchain: Scaling blockchain via full sharding. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 931–948, New York, NY, USA, 2018. Association for Computing Machinery.