Interplanetary Consensus (IPC)

Consensus Lab

Abstract

1 Introduction

A blockchain system is a platform for hosting replicated applications (represented by smart contracts in Ethereum [??] or actors in Filecoin [??]). A single system can, at the same time, host many such applications, each of which containing logic for processing inputs (also known as transactions, requests, or messages) and updating its internal state accordingly. The blockchain system stores multiple copies of those applications' state and executes the associated logic. In practice, applications are largely (or even completely) independent. This means that the execution of one application's transactions rarely (or even never) requires accessing the state of another application.

Nevertheless, most of today's blockchain systems process all transactions for all hosted applications (at least logically) sequentially. The whole system maintains a single totally ordered transaction log containing an interleaving of the transactions associated with all hosted applications. The total transaction throughput the blockchain system can handle thus must be shared by all applications, even completely independent ones. This may greatly impair the performance of such a system at scale (in terms of the number of applications). Moreover, if processing a transaction incurs a cost (transaction fee) for the user submitting it, using the system tends to become more expensive when the system is saturated.

The typical application hosted by blockchain systems is asset transfer between users (wallets). Asset transfers often involve other applications and may create system-wide dependencies between different parts of the system state. In general, if users interacted in an arbitrary manner (or even uniformly at random), this would indeed be the case. However, in practical systems, users tend to cluster in a way that those inside a cluster interact more frequently than users from different clusters. While this "locality" makes it unnecessary to totally order transactions confined to different clusters (in practice, the vast majority of them), many current blockchain systems spend valuable resources on doing so anyway.

An additional issue of such systems is the lack of flexibility in catering for the different hosted applications. Different applications may prefer vastly different trade-offs (in terms of latency, throughput, security, durability, etc...). For example, a high-level money settlement application may require the highest levels of security and durability, but may more easily compromise on performance in terms of transaction latency and throughput. On the other hand, one can imagine a distributed online chess platform (especially one supporting fast chess variants) whose state is mostly ephemeral (lasting until the end of the game) but which requires high throughput (for many concurrent games) and low latency (few people like waiting 10 minutes for the opponent's move). While the former is an ideal use case for the Bitcoin network, the latter would probably benefit more from being deployed in a single data center. [**js:** It's an okay example but just noting that concurrent games are independent and can be seen as different applications; I don't think it negates the specific point being made here.]

In the above example, one can also easily imagine those two applications being mostly, but not completely independent. E.g., a chess player may be able to win some money in a chess tournament and later use it to buy some goods outside of the scope of the chess platform. In such a case, few transactions involve both applications (e.g., paying the tournament registration fee and withdrawing the prize money). The rest (e.g., the individual chess moves) are confined to the chess application and can thus be performed much faster and much cheaper (imagine playing chess by posting each move on Bitcoin for comparison).

Interplanetary Consensus (IPC) is a system that enables the deployment of heterogeneous applications on heterogeneous underlying blockchain platforms, while still allowing them to interact in a secure way. The basic idea behind IPC is dynamically deploying separate, loosely coupled blockchain systems that we call *subnets*, to host different (sets of) applications. Each subnet runs its own consensus protocol and maintains its own ordered transaction log.

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 tree of subnets expresses a hierarchy of trust. All components of a subnet and all users using it are assumed to fully trust their parent and regard it as the ultimate source of truth. 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.

To facilitate the interaction between different subnets, IPC provides mechanisms for inter-subnet communication. Since subnets are distributed transaction-processing systems without an obvious single entity to submit transactions to one subnet on behalf of another subnet, we introduce processes called *IPC agents* that 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. Out of the box, IPC provides several primitives for 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.

The operating model described above is simple but powerful. In particular, it enables

- Scaling, by using multiple blockchain/SMR platforms to host a large number of applications.
- Optimization of blockchain platforms for applications running on top of them.
- Governance of a child subnet by its parent, by way of the parent serving as the source of truth for the child and, for example, maintaining the child's configuration, replica set, and other subnet-specific data.
- "Inheriting" by the subnet of some of its parent's security and trustworthiness, by periodically anchoring its state in the state of the parent using checkpoints.

In the rest of this document, we describe IPC in detail. In Section 3 we define the system model and introduce the necessary terminology. Section 5 describes the main components of IPC and their interfaces. [**TODO:** Finish this when sections are written.]

2 Example Use Case: Chess Platform

To better understand how IPC works and how it is useful, let us expand on the example application of a distributed chess platform sketched in the introduction of this document. Imagine platform where registered chess players meet and play against each other, while the platform maintains player rankings (e.g. Elo ratings). 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 of users' coins are on accounts residing in the rootnet's replicated state. A robust established blockchain system like Filecoin would be a good 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 directly interact with it.

Chess platform as a subnet (`L2`). The functionality of the chess 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 significantly fewer funds than the rootnet (only as much as users dedicate to playing chess). The replicas constituting this subnet are run by chess clubs or even some (not necessarily all) individual chess 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) need to lock a certain amount of funds as collateral that can be slashed by the system in case of the replica misbehaving. The collateral / slashing mechanism is described in more detail in Sections 4.6 and 6.3.

Individual games (`L3`). For each individual game of chess, a new child of the `L2` subnet is created (Section 4.1). Since not much is usually at stake in a single game and only two players are involved, the whole `L3` subnet may even be implemented by a single server that both players trust. However, this decision is completely up to the players and they may choose a different implementation of the `L3` subnet when starting the game (by submitting the corresponding transactions to the `L2` subnet). A chess game is also very easily modeled as a simple application, its state consisting of the positions of the individual pieces on the board, while players' moves are represented as transactions. 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 4.7).

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[1]. 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) – again, by submitting an `L2` transaction.

[1]We call a transaction submitted to the `L1` blockchain an "`L1` transaction".

Tournaments. Chess 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 that hosts the platform (i.e., the `L2` subnet).

This simple use case utilizes most of IPC's features. Throughout the rest of the document, we will use the on-line chess platform as a running example when describing IPC's functionality in more detail. [**mp:** This "running example" part is still to be added to the rest of the document. Coming soon, but I'd prefer to have some feedback on the general suitability of this example. Then I start integrating it in the text throughout the document.]

3 Preliminaries

The vocabulary used throughout this document is described in the Glossary [1]. The reader is assumed to be familiar with the terminology defined there. [**js:** Despite this not, 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.]

Abstractions. In this document, we reason in terms of subnets, actors, accounts, users, and IPC agents (see Glossary [1]]). While a subnet consists of multiple replicas, we treat it as one entity maintaining a common abstraction of a replicated state that can only be modified through transactions submitted either by a users or by an IPC agent. We abstract away the concrete mechanism of transaction submission and execution, as it is specific to the implementation of each particular subnet.

Interaction between subnets. In IPC, the replicated state of one subnet must react to (changes in) the replicated state of another subnet. As the replicated state of every subnet is distributed among its replicas and evolves independently of other subnets, we must establish a mechanism for interactions between the states of subnets. In particular, we must explicitly link the two replicated states of two subnets. More precisely, for any interaction between two subnets (`A` and `B`), define block heights $h_{\mathtt{A}}$ and $h_{\mathtt{B}}$, such that `A`'s replicated state at height $h_{\mathtt{A}}$ considers `B`'s replicated state to have evolved exactly until $h_{\mathtt{B}}$.

Proofs of Finality. To enable interaction between subnets, we define a *Proof of Finality (PoF)* to be data that proves that a subnet definitively reached a certain replicated state. Regardless of the subnet's ordering protocol's approach to finality (e.g., immediate finality for classic BFT protocols, or probabilistic finality in PoW-based systems), a PoF convinces the proof's verifier that the replicated state the PoF refers to will not be rolled back. For example, for a subnet using a BFT-style ordering protocol, a quorum of signatures produced by its replicas can constitute a PoF. This helps us establish the above-mentioned link between the replicated states of two subnets. If a PoF is associated with subnet `A`'s replicated state at $h_{\mathtt{A}}$, and PoF is included in subnet `B`'s replicated state at 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 $h_{\mathtt{B}}$. We denote by $PoF(tx)$ the proof that a subnet reached a state in which transaction tx already has been applied.

IPC agent and actors. For inter-subnet communication, IPC relies on two special types of actors (the IPC Gateway Actor (`IGA`) and the Subnet Actor (`SA`)) and one special type of process (the IPC Agent). 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*.

2. The `SA` 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 `SA` "governs" that child subnet).

3. Finally, 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 an SMR client of both. 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`.

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

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`.

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 `IGA`'s function *`CreateChild`* in subnet `P` is denoted `P.IGA.`*`CreateChild`*. We also use this notation for a

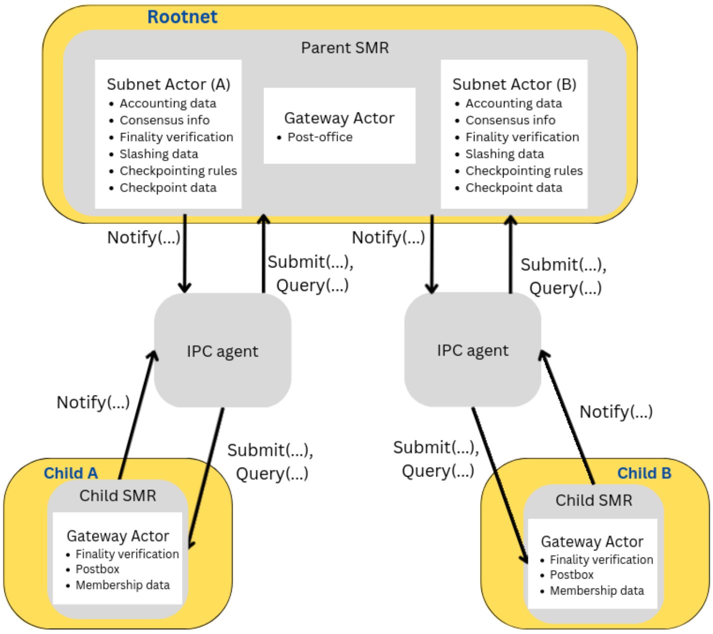

Figure 1: The basic IPC components and their interfaces in an example with one parent and 2 child subnets (A and B). [**TODO:** Update figure to be consistent with new notation.]

transaction tx submitted to subnet P that invokes the function, e.g., tx = P.IGA.$\mathit{CreateChild}$(P/C, params).

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.[2] Each user is assumed to have a personal wallet and a corresponding account in some subnet.

We also assume that the submission, ordering, and application of transactions is associated with a cost (known as transaction fees). Each SMR client submitting a transaction to a subnet is assumed to have an account in that subnet, from which this cost is deducted. If the funds are insufficient, the SMR system ignores the transaction.

Note that the operation of IPC requires the submission and processing of transactions that are not easily attributed to a concrete user. It is the transactions the IPC agent submits on behalf of a whole subnet. We discuss incentivizing (through refunds) participants to run IPC agents and pay for the associated transaction fees in Section 6.2.

[2]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.

4 IPC functionality

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

- Creating child subnets.
- Depositing funds from an account in a subnet to an account in its child.
- Withdrawing funds from an account in a subnet to an account in its parent.
- Checkpointing a subnet's replicated state in the replicated state of its parent.
- Propagating cross-net transactions.
- Slashing misbehaving child replicas.
- Removing child subnets.

In the following, we describe each functionality in detail, introducing the functions of the `IGA` and `SA` through which this functionality is exposed and the patterns in which the users and the IPC agent invoke them through submitting transactions. We summarize those components and their implementation in Section 5. For clarity, this section does not deal with the incentives for participants to run IPC agents or subnet replicas to communicate with it. We defer the discussion of incentives to Section 6.

4.1 Creating a child subnet

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

$$\texttt{IGA}.\textit{CreateChild}(\textit{subnetName, params})$$

Any user or actor of a subnet `P` can create a new subnet `P/C` by submitting a transaction `P.IGA.`*CreateChild*(`C`, *params*), where *params* is a data structure containing all subnet-specific parameters, such as the used consensus protocol, rules for joining the subnet, definitions and evaluation logic for PoMs and PoFs, and slashing policies. This results in the creation of a new Subnet Actor `SA` in `P` governing the subnet `P/C`, initialized with *params*.

The subnet is considered created as soon as `SA` 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.

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.

$$\texttt{SA}.\textit{Deposit}(\textit{amount, account})$$
$$\texttt{IGA}.\textit{MintDeposited}(\textit{amount, account, PoF})$$

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 SA.*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 a transaction
   tx = P.SA.*Deposit* (*amount*, b).
2. P orders and executes *tx*, transferring *amount* coins from a to SA.[3]
3. When P's replicated state that includes *tx* becomes final (for some subnet-specific definition of finality provable to P/C), The IPC agent constructs a *PoF*(*tx*).[4]
4. The IPC agent submits a transaction
   tx' = P/C.IGA.*MintDeposited* (*amount*, b, *PoF*).
5. P/C orders and executes *tx'*, which results in minting *amount* new coins and adding them to the balance of b.

After all the above steps are performed, the involved *amount* of coins "exists" twice: once in P, where it is owned by p.SA, and once in P/C, where it is owned by P/C.b. However, those coins can effectively only be used by the owner of P/C.b, since P.SA will not transfer its coins within P until they are burned in P/C during a withdrawal operation (see below).

We show in Algorithm 1 the pseudocode to implement the functionality.

[3]For simplicity of presentation, we always assume that transactions are properly signed by the owner of the account from which coins are being spent, that that account has sufficient balance, and in general, that the transactions passes all validity checks that we assume the subnet to perform.

[4]The exact content of the *PoF* and the way it is constructed depends on the implementation of the subnet. It might contain, for example, a quorum of replica signatures or a Merkle proof of inclusion. The IPC agent may construct it by simply observing the subnet's replicated state, or by exchanging messages with the subnet's replicas in a dedicated *PoF*-construction protocol.

Algorithm 1: Deposit operation

▶ *Owner of src*:
  submit $tx = P.\texttt{SA}.Deposit(src, amt, dest)$ to parent subnet
▶ *P.SA.Deposit(src, amt, dest)*:
  move *amt* from *src* to P.SA.*accounts.dest* `// "lock" at parent`
▶ *IPC agent*:
  upon *tx = P.SA.Deposit final at parent* **do**
    create *PoF* that *tx* is final at parent subnet `// see Section 8`
    submit *P/C.IGA.Deposited(amt, dest, PoF)*
  [**js:** this notation is also a bit weird, in that "upon tx ... final at parent" is running text and it's not immediately obvious that that's the condition]
▶ *P/C.IGA.Deposited(amt, dest, PoF)*:
  verify(*PoF*)
  increase *dest* account by *amt*

4.3 Withdrawals

A withdrawal is a transfer of funds from an account *src* in the child subnet P/C to an account *dest* in the parent subnet P. The *Withdraw* is performed analogously to the *Deposit*, but starting at the child subnet P/C:

1. The owner of *src* submits a transaction $tx = P/C.\texttt{IGA}.Withdraw(src, amt, dest)$.

2. The child subnet orders and executes the *Withdraw* transaction, burning *amt* funds in *src* (provided *src* has enough funds).

3. When the child's replicated state that includes the transaction becomes final (for some SMR-system-specific definition of finality that has been defined in the SA), the IPC agent constructs a corresponding *PoF* and submits a transaction *tx' = P.SA.Withdrawn(amt, dest, PoF)* to the parent subnet.

4. Upon ordering *tx'*, *P.SA.Withdrawn(amt, dest, PoF)* verifies the *PoF* and transfers *amt* from SA (concretely, to *src* account representation within the SA) to *dest* within the parent subnet.

We show in Algorithm 2 the pseudocode to implement the functionality.

Algorithm 2: Withdraw operation

```
▶ owner of src:
    submit tx = P/C.IGA.Withdraw(src, amt, dest)
▶ P/C.IGA.Withdraw(src, amt, dest):
    deduct amt from src // "burns" amt in child
▶ IPC agent:
    upon tx = P/C.IGA.Withdraw(src, amt, dest) final at child do
        create PoF(tx)                          // see Section 8 for details
        submit P.SA.Withdrawn(amt, dest, PoF)
▶ P.SA.Withdrawn(amt, dest, PoF):
    verify(PoF(tx'))
    move amt coins from P.SA to dest // "unlocks" amt for dest
```

4.4 Checkpointing

A checkpoint contains a representation of the state of the child subnet to be included in the parent subnet's replicated state. 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.). A checkpoint is created as follows:

1. When the predefined checkpoint trigger is met (the IPC Agent, monitoring the child subnet's state, is configured with the checkpoint trigger), the IPC agent retrieves the corresponding checkpoint data (*chkp*) from the child subnet, along with the proof of its finality (*PoF*).

2. The IPC agent submits a transaction $tx = P.\texttt{SA}.Checkpoint(chkp,\ PoF)$.

3. The *P.SA.Checkpoint(chkp, PoF)* invocation, after verifying the *PoF*, includes *chkp* in its state.

Note that we do not show here incentives for participants to submit checkpoints, the same way we do not discuss in Algorithm 3 whether the particular participant running the IPC agent has even rights to submit the checkpoint. We show instead in Section 6 different mechanisms that can be used to incentivize participants running IPC agents, while in Section 7 we describe how the reference implementation of IPC incentivizes and gives participants the right to submitting checkpoints. Analogously, we discuss in Section 7 optimizations and other design decisions made to the checkpoint operation.

Algorithm 3: Checkpoint operation

▶ *IPC agent*:
  upon *Checkpoint condition in child* **do**
    chkp = obtain state snapshot from child
    create $PoF(chkp)$
    submit $P.\texttt{SA}.Checkpoint(chkp, PoF(chkp))$

▶ $P.\texttt{SA}.Checkpoint(chkp,\ PoF\ (chkp))$:
  verify$(PoF(tx'))$
  save *chkp* in the state

4.5 Propagating cross-net transactions

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

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 postbox and an IPC agent. In a nutshell, if an actor's logic in subnet S_1 produces a transaction for a different subnet S_2, this transaction is saved at S_1's `IGA` in the postbox buffer. The IPC agent, monitoring the postbox, then iteratively submits the transaction to the appropriate subnets along the path from S_1 to S_2. In a nutshell, if a actor's logic produces a transaction for a different subnet, this transaction is saved the local Gateway Actor in the postbox buffer. The IPC agent, monitoring the postbox, then submits the transaction to the appropriate subnet.

Since, in general, we only rely on IPC Agents to be able to submit transactions to parents or children of a subnet whose state they observe, the IPC agent only propagates the transaction to the parent or child, depending on which is closer in the IPC hierarchy to the ultimate destination subnet. 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.

The implementation of the `IGA`'s *Propagate* function is sketched in Algorithm 4.

[**TODO:** Outline algorithm in text, like in the rest of functionalities.]

Algorithm 4: Cross-net transaction propagation functionality

```
▶ S.IGA.Propagate(tx, src, dest, PoF):
    verify(tx.PoF)
    case dest = src do
        apply tx
    case dest requires going up the tree do
        postbox ← postbox ∪ (tx, S/srcsrc.Parent, dest)
    case dest requires going down the tree to child S_c do
        postbox ← postbox ∪ (tx, src/SS_c, dest)
▶ IPC agent:
    upon new entry (tx, src, dest) in parentS.IGA.postbox do
        Create PoF proving that tx' has indeed been added to the list of
          cross-net transactions in the subnet
        submit tx', augmented by PoF
```

4.6 Slashing

Slashing is a penalty imposed on provably malicious validators. When validators of a child subnet misbehave, other participants can report the misbehavior for these malicious validators to get punished (e.g. by losing a previously collateralized amount). Contrary to misbehaviors at a rootnet (a subnet with no parent), where misbehavers sucessfully perform their attack without escrow available, misbehaviors at the child can be resolved at the parent subnet, provided the misbehavers have not left the subnet. For this reason, a slashing is notified first at the parent, and reconciled at the child later on. In particular, a slash on provably malicious validators of a child subnet is performed as follows:

- When the IPC agent of a correct participant identifies slashable misbehavior at subnet P/C from a set $\mathcal{M}$ of malicious validators of subnet P/C, the IPC agent constructs a *Proof of Misbehavior* (*PoM*).
- The IPC agent then submits transaction $tx = P.\texttt{SA}.Slash(\mathcal{M}, PoM)$ at the parent.
- The parent subnet, upon ordering and executing tx, penalizes the misbehavers and updates the state of SA.
- Once the parent's replicated state that includes tx becomes final, the IPC agent constructs a $PoF(tx)$ and submits a transaction $tx' = P/C.\texttt{IGA}.Slashed(\mathcal{M}, PoM, PoF(tx))$
- The child subnet, upon ordering and executing tx', updates its state to reflect the penalization at the parent.

Algorithm 5: Slash Functionality

▶ *IPC agent*:
  upon *misbehavior from set* $\mathcal{M}$ *found* **do**
    Construct PoM ($\mathcal{M}$)
    submit $tx = P.\texttt{SA}.Slash(\mathcal{M}, PoM)$ to parent subnet

▶ $P.\texttt{SA}.Slash(\mathcal{M}, PoM)$:
  verify(PoM)
  `// Update state to reflect punishment to` $\mathcal{M}$

▶ *IPC agent*:
  upon $tx = P.\texttt{SA}.Slash(\mathcal{M}, PoM)$ *final at parent* **do**
    create PoF that tx is final at parent subnet `// see Section 8`
    submit $tx' = P/C.\texttt{IGA}.Slashed(\mathcal{M}, PoM, PoF)$

▶ $P/C.\texttt{IGA}.Slashed(\mathcal{M}, PoM, PoF)$:
  verify(PoM, PoF)
  `// Update state to reflect punishment to` $\mathcal{M}$

4.7 Removing a child subnet

A child subnet P/C can be removed from its parent P through a transaction invoking $P.\texttt{IGA}.RemoveChild(P/C)$. This transaction effectively deregisters the subnet from the IPC hierarchy. We detail further how to remove a child subnet for the reference implementation in Section 7.

5 Components and their Interfaces

[**mp:** This section has been moved around and needs updating. Read with care or don't read yet. Update coming soon.]

This section summarizes IPC's main components (the `IGA`, `SA`, and the IPC agent) and their interfaces used in Section 4 to implement IPC's functionality.

5.1 Actor state

The following describes the information that needs to be part of the IPC actors (the `IGA` and `SA`).

5.1.1 Subnet Actor (`SA`)

`SA` is the actor in the parent subnet's replicated state that stores all information about the child subnet that the parent subnet needs. The `SA` is created by the `IGA` (see below) and its functions are invoked by transactions issued by the IPC agent. The state of the `SA` includes:

- *Accounting data.* This data describes the value that has been deposited to the child. It is considered locked inside the SA until it is withdrawn from the child. This data might consist of just a single value representing the sum of all such coins ("aggregated accounts" approach), [**js:** maybe just use omnibus account as in traditional banking?] but might also contain finer-grained information about balances for each account in the child subnet ("segregated accounts" approach).

- *Subnet consensus data.* This is the data (or a pointer thereto) that is needed to run the ordering protocol of the child subnet. It is specific to the ordering protocol but generally expected to contain information such as the ordering protocol used by the subnet, subnet configuration data such as the validator set, voting rights, and collateral deposits, and subnet governance mechanisms such as transaction fees, block rewards, and conditions for participation. [**TODO:** Mention our reference implementation as a concrete example here, saying what information it stores.]

- *Child state finality verification.* Logic to verify that a given child's replicated state is final, or that a particular $h$as been definitively included in the child's state. We expect that this logic will verify a *PoF* submitted (through transactions) by one or more IPC agents to the `SA`.

- *Slashing data.* List of slashable misbehaviors and corresponding definition of what constitutes a valid proof of misbehavior (*PoM*), as well as penalties for misbehavior and rewards for reporting *PoM* and logic performing the actions necessary for slashing in the parent subnet.

- *Checkpointing.* Child subnet's checkpoint data, or a pointer thereto, and checkpoint validity rules (and logic enforcing them).

5.1.2 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 state of the `IGA` includes:

- *Membership data* defining the set of replicas running the `IGA`'s subnet. This may include the identities of the replicas, their network addresses, weights of their votes in the consensus protocol (e.g., storage power table, or stake power table), and other subnet-specific membership information.

- *Parent state finality verification.* Analogously to the `SA`'s child state finality verification logic, this is the logic to verify that a state is final in the parent subnet, using a *PoF* submitted as transaction(s) to the child subnet by the IPC agent(s).

- *Inter-subnet transactions service* (denoted postbox). The IGA contains a registry of subnets and a functionality that can be used to transfer data from one subnet to another. The postbox specifies the methods and the state locations that are used for these services. This functionality is required for the communication of two actors across subnets.[5]

5.2 Interfaces

We now describe the interfaces of the Gateway Actor (IGA) and the Subnet Actor (SA), by listing the methods that can be invoked through transactions submitted to their respective subnets, as well as that of the IPC Agent process, by listing the events it reacts to. The other two processes, the parent and child replica, interact with the IPC Agent by having the IPC Agent observe the relevant parts of parent's and child's replicated state. The parent and child replicas do not perform any IPC-specific tasks themselves, except for providing the data necessary for IPC Agents to construct Proofs of Finality.

[5]When inter-subnet data transfer happens between users, they can actively participate in the propagation by submitting transactions to the parent and child subnets. actors, on the other hand, do not have that power and, therefore, cannot communicate across subnets as efficiently as users.

5.2.1 Gateway Actor (`IGA`)

Algorithm 6: `IGA` interface

▶ *IGA*:

CreateChild(name, params)

Creates a new `SA` with the given name and subnet-specific parameters (such as initial membership, etc.). The subnet governed by the created `SA` will be considered the child of the subnet of this `IGA`.

RemoveChild(name)

Deregisters the subnet from IPC.

Joined(identity, metadata, PoF)

For subnets with explicit membership defined by the `SA` in their parent, if *PoF* is valid, updates the membership data of the `IGA` to include the replica with *identity* and the associated *metadata*. A valid *PoF* means that *SA.Join(identity, ..., ...)* has been successfully invoked in the parent's replicated state. [**js:** I understand we call them joined/deposited because we're including a proof of invoking join/deposit in the parent, but this still sounds less than idiomatic. It sounds like a state accessor or a variable rather than an action.]

Leave(identity)

For subnets with explicit membership defined by the `SA` in their parent, removes the replica with *identity* from this subnet's membership.

Deposited(amt, dest, PoF)

If *PoF* is valid, adds *amt* newly minted coins to account *dest*. A valid *PoF* means that a corresponding *SA.Deposit(..., amt, dest)* has been successfully invoked in the parent's replicated state.

Withdraw(src, amt, dest)

Burns *amt* coins from account *src* to be returned to the *dest* account in the parent subnet. [**js:** amt? I feel like this is C or matlab and we have some arbitrary character limit for function prototypes :P]

Propagate(tx)

Adds the cross-net transaction *tx* to the list of transactions to be submitted to another subnet. The IPC agents observing the state of this subnet will pick it up from here and perform the actual submission.

Slashed($\mathcal{M}$, PoM, PoF)

If *PoM* is a valid proof of misbehavior of a set $\mathcal{M}$ of validators, and if *PoF* is valid, then update state to reflect predefined punishment to misbehaviors in $\mathcal{M}$.

5.2.2 Subnet Actor (SA)

Algorithm 7: SA interface (governing subnet C)

► *SA*:

- *Join(identity, src, collateral)*
  - For subnets with explicit membership defined by the SA, adds the replica with the given *identity* to the replica set of the subnet governed by this SA. Join also locks *collateral* coins in the *src* account, which will only be released when the replica leaves.
- *Left(identity, PoF)*
  - For subnets with explicit membership defined by the SA, if *PoF* is valid, adds the replica with the given *identity* to the replica set of the subnet governed by this SA. The *PoF* proves that the corresponding *C.IGA.Leave(identity)* has been successfully invoked and the replica with the given *identity* is not running subnet C any more.
- *Deposit(src, amt, dest)*
  - Locks *amt* coins on account *src* to be deposited to the *dest* account on the child subnet.
- *Withdrawn(amt, dest, PoF)*
  - If *PoF* is valid (meaning that a corresponding *C.IGA.Withdraw(..., amt, dest)* has been successfully invoked), unlocks *amt* coins on the *dest* account.
- *Checkpoint(chkp, PoF)*
  - If *PoF* is valid (meaning that a corresponding child subnet indeed reached finality on the state represented by *chkp*), saves *chkp* in the replicated state as part of the SA. This will be the most recent state that the child subnet is considered to have reached.
- *Slash($\mathcal{M}$, PoM)*
  - If *PoM* is a valid proof of misbehavior of a set $\mathcal{M}$ of validators, then update state to reflect predefined punishment to misbehaviors in $\mathcal{M}$.

5.2.3 IPC Agent

We assume that an IPC Agent is only responsible for a single pair of parent-child subnets, the state of which it can access. It reacts to changes in those states triggered by the corresponding invocations of actors, as listed below.

Algorithm 8: IPC Agent interface

▶ *IPC Agent*:
- **upon** *parent.SA.Join(identity, src, collateral)* **do**
  - Constructs a *PoF* proving that the invocation of *parent.SA.join(identity, src, collateral)* has been finalized in the parent's replicated state, as well as subnet-specific metadata based on the *identity* of the joining replica and the associated *collateral* and notifies the child subnet by submitting a transaction that invokes *child.`IGA`.Joined(identity, metadata, PoF)*.
- **upon** *child.`IGA`.Leave(identity)* **do**
  - Constructs a *PoF* proving that the invocation of *child.`IGA`.Leave(identity)* has been finalized in the child's replicated state and notifies the parent subnet by submitting a transaction that invokes *parent.SA.Left(identity, PoF)*
- **upon** *parent.SA.Deposit(..., amt, dest)* **do**
  - Constructs a *PoF* proving that the invocation of *parent.SA.Deposit(src, amt, dest)* has been finalized in the parent's replicated state and notifies the child subnet by submitting a transaction that invokes *child.`IGA`.Deposited(amt, dest, PoF)*
- **upon** *child.`IGA`.Withdraw(..., amt, dest)* **do**
  - Constructs a *PoF* proving that the invocation of *child.`IGA`.Withdraw(..., amt, dest)* has been finalized in the child's replicated state and notifies the parent subnet by submitting a transaction that invokes *parent.SA.Withdrawn(amt, dest, PoF)*
- **upon** *child.`IGA`.cross-netTX(tx, destName)* **do**
  - If *destName* points up the hierarchy, submits *tx* to the parent subnet.
- **upon** *parent.`IGA`.cross-netTX(tx, destName)* **do**
  - If *destName* points down the hierarchy, submits *tx* to the child subnet.
- **upon** *checkpoint condition in child* **do**
  - Create checkpoint, a *PoF* of the checkpoint at the child and submit from child to parent
- **upon** *misbehavior from set $\mathcal{M}$ found* **do**
  - Create a proof of misbehavior *PoM* and submit to parent
- **upon** *parent.`SA`.Slash($\mathcal{M}$, PoM)* **do**
  - Constructs a *PoF* proving that the invocation of *parent.`SA`.Slash($\mathcal{M}$, PoM)* has been finalized in the parent's replicated state and notifies the child subnet by submitting a transaction that invokes *child.`IGA`.Slashed($\mathcal{M}$, PoM, PoF)*

Note the function pairs *Joined/Leave* of the `IGA` actor (Algorithm 6) and *Join/Left* of the `SA` (Algorithm 7). This is because the intended invocation pattern for several functionalities is as follows (to be detailed in Section 4).

1. The initiating subnet invokes a function on a actor (e.g., *`SA`.Join*).
2. IPC agent notices the invocation, constructs the required *PoF*, and submits a transaction to the other subnet (e.g., *`IGA`.Joined*)

6 Incentives

In the previous sections, we defined the components of an IPC system and their roles in implementing the IPC functionality. Most of the functionality involved submitting transactions to subnets by an IPC agent. However, 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 4.6), participants running subnet replicas might also need incentives to participate in the creation of a PoF.

This section describes mechanisms that can be used to incentivize participants running IPC agents to submit the required transactions and pay the corresponding transaction fees, as well as replicas to participate in PoF creation. It is *not* the goal of this section to provide a game-theoretic model of viable incentive mechanisms and their analyses. We merely present tools for implementing such mechanisms, to be used by those who design and implement concrete instances of IPC subnets.

6.1 Accounts and Actors

We assume that a participant running an IPC agent has associated accounts in both the parent and child subnets and that the fees for the transaction the IPC agent submits to the respective subnets are deducted from the respective accounts. If the balance of the account is insufficient to pay the transaction fee, the transaction is considered invalid and is ignored by the subnet. The `SA` and the `IGA`, can, as actors, also hold funds that their logic can distribute among other accounts or actors on their respective subnets.

Gateway Actor. The `IGA` accumulates funds from its own subnet. For example, the subnet's implementation can require a certain part of each transaction fee to be sent to the Gateway Actor.

Subnet Actor. The `SA` accummulates funds from the subnet it governs. There are several ways how one can imagine the `SA` to be funded, for example, by charging fees for checkpoint transactions, or by periodicaly charging the child's replicas for being included in the replica set. The subnet actor also holds at the parents all the funds deposited at the child, which can be thus used as part of the incentive mechanism (for example, through slashing).

6.2 Refunds and Rewards

As shown in Section 4, an IPC Agent may need to submit transactions that invoke functions of the `IGA` or `SA`. The participant running this IPC agent can receive a refund of the transaction fee (possibly augmented by an additional reward) directly from the invoked actor, according to an incentive mechanism defined in the actor's logic. For example, the actor's logic may credit an account with the amount of the transaction fee for reporting frauds through slashing, submitting cross-net transactions, or submitting checkpoint transactions.

Similarly, other behavior can be punishable by slashing an amount. The `SA` has access to the accounting data and can thus penalize participants that misbehave by slashing a portion or their funds, in accordance with the accounting data.

To incentivize the replicas of a subnet to collaborate with the 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.

6.3 Collateral and Slashing

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*. To this end, the `SA` can associate each replica of the child subnet with a collateral. Replicas must transfer this collateral to the `SA`, and the `SA` 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 `SA` (invoking its *Slash* function). The `SA` 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 `SA` until the associated replica stops participating in the subnet, either by leaving or by being slashed.

7 IPC's reference implementation

In this section, we describe the particular choices implemented by the Consensus Lab team for the reference implementation of IPC. The current implementation considers Filecoin as the rootnet, and Trantor running in child subnets. For our purposes, it is important to note that Trantor is a BFT consensus protocol with immediate finality, and Filecoin is a longest chain style protocol with probabilistic finality.

7.1 Components

The IPC reference implementation preserves all the components described in Section 5 without any additions. We however list here implementation decisions concerning these components.

`IGA` and `SA`. The `IGA` is the only built-in actor of the reference implementation. It is thus the entry point for all IPC-related operations. For this reason, it is the `IGA` that receives the funds to be deposited for all subnets, and that releases the funds on withdrawals. It also holds the collateral of members of the validator's list. The `SA` is user defined by the creators of that subnet, and thus it holds the subnet-specific information, such as the consensus mechanism used, the checkpointing data and rules, slashing rules, etc. This means that unlike the high-level funcionality described in Section 4, in the implementation the IPC agent calls the actor's method on the `IGA` (and not the `SA`), since that is the actor that must lock or release funds, and perform other state changes. The `IGA` then requires the minimal required checks for the state change to take place, and then calls the `SA` to perform any additional subnet-specific check that the particular subnet may perform. If the checks pass, then the `IGA` performs the operation.[**TODO:** Expand on this when talking about each of the functionalities]

Compressed accounting. In Section 5 we mentioned that the state of `SA` contains accounting data. In the reference implementation, `SA` contains as accounting data of the subnet a single variable representing the sum of all the individually locked funds at the parent which are dedicated to the child subnet, i.e. the subnet's *circulating supply*.

Content addressing. The reference implementation makes use of IPFS-style content addressing, in that data is stored where relevant and referred to with a Multiformats-compliant content identifier (CID) elsewhere. In particular, CIDs that refer to information of a specific child's subnet can be retrieved through BitSwap from any of the participants with replicas 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, as we will show, this is not a problem, as this would just mean that operations that have this subnet as source will not be resolved, not affecting the rest of the IPC tree.

IPC agent and metadata. In the previous sections, we considered that every parent-child pairing will have 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 the 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[**arp:** basically a user using IPC wallet once we separate both].

7.2 Topdown transactions

Once the parent subnet orders and executes topdown transactions, the `IGA` updates its state by increasing a nonce specific for the child. The IPC agent then adds the respective child transactions to the cross-net transactions pool (referencing the same nonce). In order to prevent inconsistencies across replicas, the IPC agent generates a PoF by running an instance of an all-to-all broadcast that generates signed *certificates* containing a supermajority of signatures from child validators[6]. In this all-to-all broadcast, validators broadcast the topdown transactions that they locally consider as valid (as participants run a local parent replica and the IPC agent is notified of changes to the local replica). This way, when a supermajority of correct child validators locally see and consider the corresponding transaction at the parent as final in their parent subnet, the rest of the validators can instead verify the certificate to update the state of the child subnet, preventing inconsistencies with participants running straggling parent replicas. After the IPC agent verifies this PoF, the IPC agent provides the transactions to the child replica for ordering and execution.

7.3 Bottomup transactions

Batching through the `IGA`. The child subnet aggregates cross-net transactions from within its own subnet and those propagated from its children, and includes their CIDs in the next checkpoint. All related bottomup transactions are made via the `IGA` of the child. Bottomup transactions are batched there until it is time to checkpoint at the parent (specifically, every Δ child blocks). This way, the `IGA` serves as the single data structure storing bottomup transactions and the IPC agent only needs to monitor a single location to get all necessary information from the child subnet. Since postbox transactions are included in the checkpoint, the execution of the checkpoint also depends on the postbox functionality at the parent handling those transactions (recall that the parent's postbox is located at the parent's `IGA`). If

[6] an example of such a protocol can be found in Trantor's availability module.[**arp:** In reality, atm the way this actually works is with local validity check after execution at the child replica, which can generate inconsistencies between updated and straggling replicas, but there's works to change this to what is described aboved: `https://github.com/consensus-shipyard/ipc-agent/issues/72`]

the bottomup transactions sent in the batch along with the checkpoint fails, the atomic execution of the checkpoint will fail too.

Cross-net transactions traversing subnets. In IPC, a transaction in a subnet may trigger a state change in another subnet for which there is a path via the parent. The current reference implementation uses the postbox in each `IGA` to split cross-net transactions of this type into multiple atomic operations in the parent-child hierarchy. The cross-net transaction tx is propagated to each immediately adjacent subnet along the path until it reaches its destination by traversing through the postbox of all intermediate subnets, via cross-net transactions $tx'(tx)$ containing tx as payload. Once tx' is ordered and executed at an intermediate subnet, the IPC agent does not create another cross-net transaction $tx''(tx)$ at the next subnet along the path, but instead leaves the transaction in the postbox of that intermediate subnet. Only once an externally owned account (EOA) of that subnet creates tx'', paying for the fees required to execute this step of the path, can tx reach the postbox of the next subnet of the path to its destination.

In the current implementation, thus, a transaction at the postbox has not been paid for, and thus it will not be propagated until an EOA pays for it. A cross-net transaction that has been paid for leaves the postbox to join the *cross-net registry*. Both contain cross-net transactions and are part of the state of `IGA`, but only those transactions in the cross-net registry are propagated by the `IGA`.

For example, when an actor in a child subnet S_C triggers a state change in its grandparent subnet S_G, the cross-net transaction tx must reach the grandparent through the intermediate parent subnet S_P that connects them. In the reference implementation, this means that first a bottomup transaction $tx_{b_1}(tx)$ with tx as payload, S_C as source and S_P as recipient is propagated to S_P's postbox from S_C's cross-net registry. Once tx_{b_1} is ordered and executed at S_P, an EOA needs to pay for the cost of moving tx into S_P's cross-net registry, creating a new transaction $tx_{b_2}(tx)$ with tx as payload, S_P as source and S_G as recipient. Finally, tx_{b_2} is ordered and executed at S_G, meaning that tx reaches its destination.

Checking for Finality. Therefore, a child's transaction (or state) *i*s accepted at the parent as final by providing a PoF containing enough signatures amounting for at least 2/3 of the voting power in the child running an instance of Trantor[7].

In other words, given transaction tx, the PoF (tx) needed for bottomup transactions is a CID to the latest block decided by the child's Trantor

[7] The current implementation relies on collecting multiple signatures. A next step in the implementation road-map is to offer a threshold signature mechanism instead of using a multisig. For now, multisigs serve the purpose of an MVP implementation.

consensus protocol, which already contains a certificate to verify finality. The parent subnet considers the PoF valid if it contains signatures from validators with at least 2/3 of the voting power in the child. The voting power is measured according to what is stored in `SA` for the epoch containing tx.

Checkpointing. [**TODO:** This must be updated] We show in Algorithm 9 the main design choices made by the reference implementation. A checkpoint is first triggered every Δ blocks decided at the child subnet. If the latest block decided meets this condition, and if the participant's child subnet is a validator according to the state stored at the parent, then the IPC agent starts computing the checkpoint as follows:

- The IPC agent obtains a state snapshot from the child's subnet.
- The IPC agent obtains the CIDs of all new grandchildren's checkpoints, and of the bottomup transactions in the cross-net registry
- The IPC agent computes the checkpoint, and creates a $PoF(chkp)$
- The IPC agent submits $tx = P.\texttt{SA}.Checkpoint(chkp, PoF(chkp))$
- The parent subnet, upon ordering and executing tx, saves $chkp$ in the state. [**TODO:** Saves a CID after submitting it to IPFS? or the actual checkpoint?] [**TODO:** and updates also `IGA` with the new postbox transactions?]

[**TODO:** Talk about control messages as part of the checkpoint (in text) as future work for garbage collection and gas cost]

Algorithm 9: Checkpoints IPC reference implementation [**TODO:** Fix formatting]

▶ *IPC agent*:
- **upon** *newBlock from child subnet* **do**
  - **if** *newBlock.blockheight* mod $\Delta = 0$ **then**
    - **if** *P.SA.isValidator(Self)* **then**
      - $chkpData \leftarrow$ `newBlock`.$GetCID()$;
      - $gcChkps \leftarrow$ query child's `IGA` for grandchildren's checkpoints
      - $CrossnetTxs \leftarrow$ `IGA`.CrossnetRegistry
      - $chkp \leftarrow createChkp$(chkpData, gcChkps, CrossnetTxs) submit $P.\texttt{SA}.Checkpoint(chkp)$

▶ *P.SA.Checkpoint(chkp)*:
- $P.\texttt{SA}.verify(chkp)$ `// PoF is newBlock's certificate`
- $P.\texttt{IGA}.Checkpoint(P/C, chkp)$

▶ *P.IGA.Checkpoint(P/C, chkp)*:
- Save $chkp$ in the state
- Execute tx, $\forall tx \in chkp.CrossnetTxs$

[**TODO:** Talk about execute might require additional checks on P.`SA`, such as withdrawals of collateral requires calling verification on any additional check that the `SA` might define]

7.4 Other operations

[**TODO:** Extend this part, but we are getting there. Also reformat section and include answers to Q4 on from Alfonso's sync]

Create. Subnets are created by instantiating a new `SA` and registering the `SA` in `IGA`. When the `IGA` contains a minimum amount of collateral for `SA` (where enough is user defined in the `SA` [**arp:** Question: is it?]), the subnet can be registered in `IGA`. This registration in the `IGA` is what allows this subnet to interact to the rest of subnets registered in `IGA` through IPC, and thus we refer to it as the creation of the subnet.

Join and leave. Validators join the validator set by depositing the required collateral in the `IGA`, collateral that they recover by leaving the subnet (provided they have not been slashed before). If the amount of collateral drops below the required minimum (e.g. by a validator, or validators, leaving the subnet), 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`, until the minimum collateral is restored via deposits from validators.

Removing a subnet and retrieving funds. An active subnet can be removed by a majority of current validators, releasing all the collateral and the circulating supply back at the parent. This prevents validators and users from having their funds stuck at the `IGA` once enough validators recover their collateral and leave the subnet inactive. In this case, though, users and remaining validators can retrieve their funds by either (i) depositing enough funds as collateral as needed to reactivate the subnet; or (ii) retrieving their funds thanks to the latest snapshot checkpointed at the parent.

Slashing. [**arp:** Nothing atm, should we even mention it then in previous sections? Option2: describe here some of the sketches in issues of ipc-agent (ipc-actors?) repos ("next" label)]

7.5 Incentives

At the moment, validators get rewarded for executing the checkpoint algorithm by charging an IPC fee on all transactions traversing the postbox. This incentivizes validators in participating on the checkpointing functionality, even if that costs them a fee to be paid for the transaction at the parent. [**js:** in a given checkpointing period, which validators charge the fees, which validators pay for the checkpoint, and are these the same? in other words, are incentives tightly aligned?] [**TODO:** TBC. No governance account, but **participation incentives** by:

1. Additional IPC fee for validators to relay
2. Incentivizing checkpoint submission as a result of batching with other cross-net transactions (which contain the IPC fees), further justifying checkpoints being stored at `IGA`.
3. Keeping Stake at SA and slashing through fraud proofs

]

[**TODO:** Deposits, withdrawals, check collateral conditions? something else worth mentioning? How are subnets created in the reference impl, and killed? how does one join/leave (something worth mentioning)?] [**TODO:** When is collateral checked, and what happens if collateral not met?] [**TODO:** Nodes are just rewarded with IPC hardcoded base fees. Are they ever punished? how? (lost collateral by equivocation, checkpoint omission?, removed from replica set?]

8 Verifying the Finality of tx

[**arp:** I think this section should contain much more than this (but that perhaps this section does not follow our timelines for document completion (more of a complement of the document). Particular content here imo: Analysis for improvements wrt reference implementations (i.e. threshold signatures instead of everyone submitting checkpoints, governance account instead of no incentives, etc.); and Comprehensive list of different approaches for functionality/functions (like we had in the legacy document).][**gg:** Agreed] A main ingredient in any Interplanetary Consensus implementation is the creation and verification of a finality proof for a given tx in some subnet. In the previous sections we left these functions opaque. For example, `SA`.$verifyGlobalFinality(tx,PoF)$ was used by the parent replica to verify the finality at the child subnet of tx. The creation of PoF and the verification method at the child replica (for transactions of that occur at the parent subnet), are only hinted by plain text. There are multiple ways to implement these functionalities, each with its own trade-offs. Below we propose several such implementations.

References

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

A Glossary

[**TODO:** (Marko)Add IPC Glossary [1] here and move most of model down here]