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... [**TODO:** Finish this when all sections are stable.]

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 a 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) needs to lock a certain amount of funds as collateral that can be slashed by the system in case of misbehavior. The collateral / slashing mechanism is described in more detail in Sections 4.7.3 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 4.6).

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[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.

Tournaments. Chess tournaments can be organized using the platform, where each player registers by submitting a corresponding `L2` transaction. When the tournament finishes, the

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

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.] [**arp:** Following discussion on integrating cross-net txs in this example, how about having a common ranking of a user across different games (e.g.: heckers, 3d chess, Xiangqi, etc.) that are in different subnets (rootnet children) and cross-net txs inform each other of match results to locally update user's ranking (and to authenticate in the first place)] [**mp:** In fact, I meant basically that above, in the paragraph about individual games (last sentence is already pointing to the section on cross-net txs.)]

3 Preliminaries

The vocabulary used throughout this document is described in the Glossary [1] (e.g., *subnets*, *actors*, *accounts*, *users*, and *IPC agents*). 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.] [**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 (which is replicated across all replicas) that can only be modified through transactions submitted either by *users* or by an *IPC agent*. We further 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 (or, simply, state) of one subnet often needs to react to changes in the state of another subnet. As the state of every subnet evolves independently of the state of other subnets, *IPC establishes a protocol for interactions between the states of different subnets.*

To enable such interaction and as the basis of the protocol, IPC relies on *Proofs of Finality (PoF)*. In a nutshell, *PoF* consists of 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 [**?**], or probabilistic finality in PoW-based systems [**?**]), a *PoF* serves to convince the *PoF* 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*. [**mv:** what about longest chain style protocols, how do we construct *PoF* there?] If a *PoF* is associated with subnet `A`'s replicated state at *block height* $h_{\texttt{A}}$, and *PoF* is included in subnet `B`'s replicated state at block height $h_{\texttt{B}}$, then subnet `B`'s replicated logic will consider all `A`'s state changes up to $h_{\texttt{A}}$ to have occurred at `B`'s height $h_{\texttt{B}}$.

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* and we assume *state* never be rolled back. [**mv:** but what if it is - esp at longest chain parent net?] We also denote by $PoF(tx)$ the proof that a subnet reached a state in which transaction tx already has been applied to 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`.

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`*$(\texttt{P/C}, params)$.

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 an own 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, or *gas*). Each subnet client (user or IPC agent) 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 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. We discuss incentivization of participants to run IPC agents and pay for the associated transaction fees in Section 6.2.

IPC agent and actors. For inter-subnet communication, IPC relies on two special types of actors: the IPC Gateway Actor (`IGA`) and the IPC Subnet Actor (`ISA`); 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*. Each subnet contains exactly one `IGA`. [**mv:** Why is IGA an actor an not a blob of information? When does the state of IGA need to change (to demonstrate the need of IGA to be an actor) and how does it change (governance)?]

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 $\texttt{ISA}_\texttt{C}$.

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 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`.

[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.

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

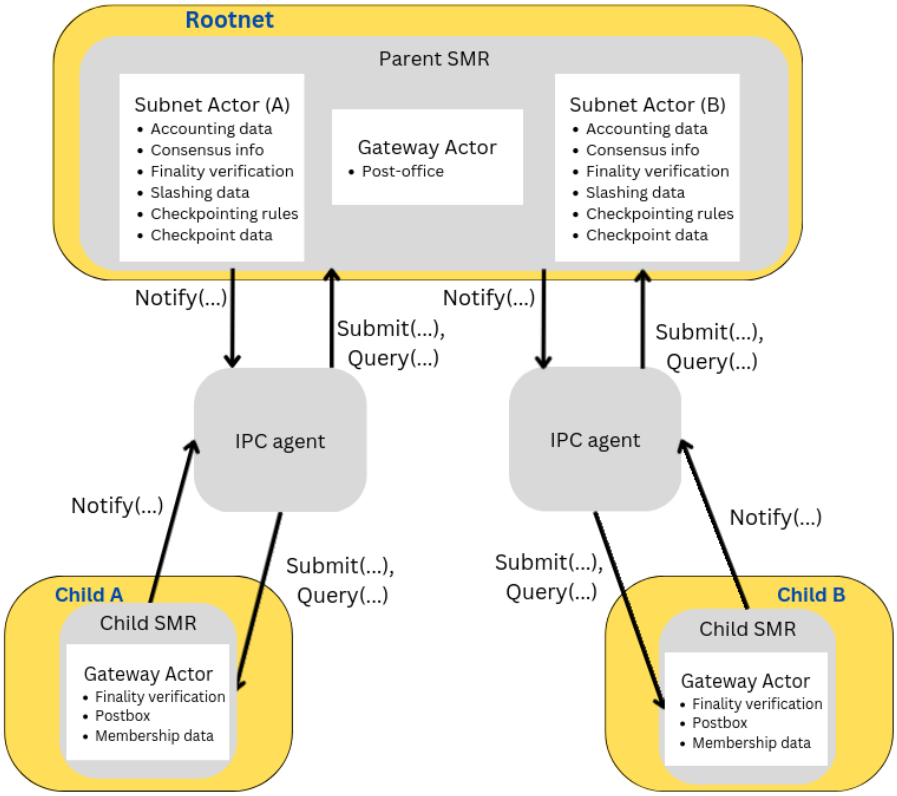

Figure 1: The basic IPC components and their interfaces in an example with one parent and 2 child subnets (A and B). [**TODO:** Update figure to be consistent with new notation.]

[**TODO:** Matej: Describe firewall property.]

[**mv:** What is missing here is a trust model - sth like an expanded slide 12 in] `https://docs.google.com/presentation/d/1l-wEprVrUQFvOjn_eV_QeAZqy6xy1Z7nnKS42zRMH-w/edit#slide=id.g10e8b8656ea_0_65`. It needs to be clear why do we need BFT in the first place and who trusts whom.

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. IPC interface exposes the following functionalities:

1. Creating child subnets in the IPC hierarchy.
2. Depositing funds from an account in a subnet to an account in its child.
3. Withdrawing funds from an account in a subnet to an account in its parent.
4. Checkpointing a subnet's replicated state in the replicated state of its parent.
5. Propagating cross-net transactions.
6. Slashing misbehaving child replicas.
7. Removing child subnets from the IPC hierarchy.

8. Managing proof-of-stake subnets, exposing functions for adding and removing replicas, managing the associated collateral, and slashing of provably misbehaving replicas.

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. 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.

`IGA.`*`CreateChild`*(*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 *PoM*s and *PoF*s, and slashing policies. This results in the creation of a new IPC Subnet Actor $\texttt{ISA}_\texttt{C}$ in `P` governing the subnet `P/C`, initialized with *params*.

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.

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 $tx = \texttt{P.ISA}_\texttt{C}\texttt{.}\textit{Deposit}(\textit{amount}, \texttt{b})$.
2. `P` orders and executes *tx*, transferring *amount* coins from `a` to $\texttt{ISA}_\texttt{C}$.[3]
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]

[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.

4. The IPC agent submits a transaction
   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`.

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).

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.

`IGA.`***`Withdraw`***(*amount, account*)
`ISA.`***`ReleaseWithdrawn`***(*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 `IGA.`***`Withdraw`***(*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 = `P/c.IGA.`***`Withdraw`***(*amount*, `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
   tx' = `P.IGA.`***`ReleaseWithdrawn`***(*amount*, `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`.

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. 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, along with the proof of its finality $PoF(state)$.

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

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 $\texttt{S}_1$ produces a transaction for a different subnet $\texttt{S}_2$, it calls $\texttt{S}_1$.`IGA.`*Dispatch*, which saves the transaction in $\texttt{S}_1$'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 $\texttt{S}_1$ to $\texttt{S}_2$ using `IGA.`*Propagate*.

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, an IPC agent only propagates the transaction to the parent or child, depending on which is next along the shortest path from $\texttt{S}_1$ to $\texttt{S}_2$ 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 cross-net transaction using an example where an actor 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 tarnsaction
   $tx = \texttt{P/B.ActorB.}\textit{SomeFunction}(\textit{someParams})$

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 local collection-type data structure that we call postbox and denote P/A.IGA.postbox.

4. Let $state_{\texttt{A}}$ 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 $state_{\texttt{A}}$ is final, it constructs a $PoF(state_{\texttt{A}})$ and submits a transaction $tx_{\texttt{A}} = \texttt{P.IGA.}\mathit{Propagate}(tx, \texttt{P/A}, \texttt{P/B}, PoF(state_{\texttt{A}}))$.

5. Subnet P orders and executes $tx_{\texttt{A}}$, verifying $PoF(state_{\texttt{A}})$ and (internally) invoking P.IGA.*Dispatch*(*tx*, P/A, P/B). This, in turn, adds *tx* along with its routing metadata to P's postbox P.IGA.postbox.

6. Analogously to step 4, the IPC agent submits a transaction $tx_{\texttt{P}} = \texttt{P/B.IGA.}\mathit{Propagate}(tx, \texttt{P/A}, \texttt{P/B}, PoF(state_{\texttt{P}}))$, where $state_{\texttt{P}}$ is the state of P with *tx* already included in P.IGA.postbox.

7. Upon ordering and executing $tx_{\texttt{P}}$, P/B.IGA.*Propagate* verifies $PoF(state_{\texttt{P}})$. Detecting that the destination is the own subnet, the implementation of P/B.IGA.*Propagate* executes *tx* instead of propagating it.

4.6 Removing a child subnet

TODO

4.7 Proof-of-stake subnets

IPC provides specialized functionality for subnets based on proof-of-stake (PoS). A child subnet can leverage the parent subnet's coins to serve as collateral disincentivizing the child's replicas from misbehaving. In such a case, the ISA governing the PoS-based child subnet maintains the child's membership – the list of replicas the child subnet consists of, along with the amount of collateral associated with each of them. 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 reconfigures accordingly.

IPC provides the following 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)

[**TODO:** Matej: Finish the rest of this subsection in the same style as the previous subsections.]

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 (at least *amount*) 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
   $tx = \texttt{P.ISA}_\texttt{C}.\textit{StakeCollateral}(\texttt{P.a}, \textit{replica}, \textit{amount})$.
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
   $tx' = \texttt{P/C.IGA}.\textit{UpdateMembership}(\textit{membership}, PoF(tx))$.
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 and that no replica has more voting power than corresponds to its collateral after the release. 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 PoC-based subnet `P/C`. The procedure for releasing collateral is as follows.

1. The owner of `P.a` submits the transaction
   tx = `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*, `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 `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" = `P.ISA`$_{\texttt{C}}$`.ReleaseCollateral`(*membership*, $PoF(state)$)

8. Upon ordering *tx"* and successfully verifying $PoF(tx)$, `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 proof-of-stake 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 = `P.ISA`$_{\texttt{C}}$`.Slash`(*replica*, PoM)

3. Upon ordering *tx*, `P.ISA`$_{\texttt{C}}$ applies 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.

5 IPC Actor Implementation

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

This section describes, at a high level, the implementation of IPC's main components: the `IGA`, `ISA`, and the IPC agent.

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 *PoF*s 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 messages).
- 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.

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 created by the `IGA` by invoking the parent's `IGA.`*`CreateChild`*(*subnetName, params*) function (see Section 4.1). The *params* value plays a crucial part in the creation of the `ISA`, as it defines several parts of of the `ISA`'s state. 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.

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, params)
9      newSubnetActor(params)
10     childSubnets = childSubnets ∪ {name}
11 RemoveChild(name)
12     childSubnets = childSubnets \ {name}
13 MintDeposited(amount, account, PoF)
14     if valid(PoF) then
15         mint amount new coins
16         transfer minted coins to account
17 Withdraw(amount, account)
18     if account.balance ≥ amount then
19         Burn amount coins from account
20 Dispatch(tx, src, dest)
21     postbox = postbox ∪ {(tx, src, dest)}
22 Propagate(tx, src, dest, PoF)
23     if valid(PoF) then
24         if dest = ownSubnetName then
25             execute tx
26         else if ∃s ∈ childSubnets ∪ {parentSubnetName} : s is part of dest then
27             Propagate(tx, src, dest)
28 UpdateMembership(membership, PoF)
29     if valid(PoF) then
30         targetMembership = membership
```

- 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 is has how much collateral (*child-*

Membership), 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. Both *validPoM* and *slashingPolicy* are part of *params* passed to `IGA.CreateChild` when the `ISA` is created.

Algorithm 2: IPC Subnet Actor (`ISA`)

valid: predicate over a *PoF* defining its validity criteria
lockedFunds: total amount of funds circulating in the child subnet
checkpoints: set of checkpoints of the child's replicated state
childMembership: map of replica identities to their respective staked collaterals
collateral: map of accounts to replica identities, to staked collaterals
collateralRequests: set of received but unsatisfied requests for releasing collateral
validPoM: predicate over a PoM defining its validity criteria
slashingPolicy: procedure to execute on reception of a valid PoM

Deposit(amount, account)
    lockedFunds += amount
ReleaseWithdrawn(amount, account, PoF)
    if *valid*(*PoF*) $\wedge$ *lockedFunds* $\geq$ *amount* **then**
        lockedFunds $-=$ *amount*
        transfer *amount* to *account*
Checkpoint(snapshot, PoF)
    if *valid*(*PoF*) **then**
        checkpoints = *checkpoints* $\cup$ {*snapshot*}
StakeCollateral(account, replica, amount)
    childMembership[*replica*] += *amount*
    collateral[*account*][*replica*] += *amount*
RequestCollateral(replica, amount, account)
    if *collateral*[*account*][*replica*] $\geq$ *amount* **then**
        childMembership[*replica*] $-=$ *amount*
        collateralRequests = *collateralRequests* $\cup$ {(*amount, account*)}
ReleaseCollateral(membership, PoF)
    if *valid*(*PoF*) $\wedge$ *membership* = *subnetMembership* **then**
        for *(amount, account)* $\in$ *collateralRequests* **do**
            transfer *amount* to *account*
Slash(replica, PoM)
    if *valid*(*PoM*) **then**
        slashingPolicy(*PoM*)

6 Incentives

[**mv:** I did not read much new or concrete in this section... After reading it I have no concrete idea how this works. We ned running examples badly]

In the previous sections, we defined the components of an IPC system and their roles in implementing the IPC functionality. The functionality often involves 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 4.7.3), 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 `ISA` 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 `IGA`.

Subnet Actor. The `ISA` accummulates funds from the subnet it governs. There are several ways how one can imagine the `ISA` 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 `ISA` also holds at the parent 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 `ISA`. 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 `ISA` 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 `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.

7 Related Work

- ZK and optimistic rollups
- sharding
- state channels
- payment channels
- plasma channels
- PoS sidechains
- Avalanche/ICP subnets
- Polygon supernets
-
-
-

8 IPC's reference implementation

In this section, we describe the particular choices implemented by the ConsensusLab 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.

8.1 Components

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

8.1.1 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. [**mv:** remember to define trust model.] 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].

8.1.2 `ISA`

In the reference implementation, the subnet actor of subnet `C` holds the state specific to subnet `C`.

Circulating supply. In Section 5 we mentioned that the state of `ISA` contains accounting data. In the reference implementation, the `ISA` contains no accounting data as it does not hold the deposited funds. Instead, `IGA` contains a single variable representing the sum of all the individually locked funds at the parent which are accessible from 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, 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.

`ISA`'s state. In particular, the state of $\texttt{ISA}_{\texttt{C}}$ at `P` contains:

- Consensus mechanism used at `C` (always Trantor at the moment)
- Checkpointing data and rules
- `C`-specific slashing rules
- `C`'s finality verification
- `C`'s replica set
- Address of `C`'s `IGA`
- ID of the parent subnet `P`
- `C`'s genesis block
- `C`'s status, whether inactive or active
- `C`'s circulating supply
- `C`-specific checks for the approval of deposits/release of funds and stake

Inactive subnet. If the total amount of collateral drops below the required minimum (e.g. by a set of replicas 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 replicas. In this case, though, users and remaining replicas can retrieve their funds by either (i) depositing enough funds as collateral to reactivate the subnet; or (ii) retrieving their funds thanks to the latest snapshot checkpointed at the parent.

We illustrate the interactions between `IGA` and `ISA`'s states after introducing in Section 8.1.3 the `IGA` as per the reference implementation.

8.1.3 `IGA`

The entry point of all functionality is the `IGA`, unlike in the high-level functionality described in Section 4. In the reference implementation, 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 child replicas. The `IGA` then performs checks for the state change to take place, and then calls the `ISA` to perform any additional subnet-specific check that the particular subnet may implement.

For example, let u be a user with an account a at a subnet `P` that implements a chess platform. Suppose u wants to join a tournament at subnet $\texttt{P/C}_\texttt{T}$ that has a registration cost of c. To do so, u joins the tournament via submitting at `P` a transaction $tx = \texttt{P.IGA.}\textit{Deposit}\,(P/C_T,\ c)$ signed with account a. However, the chess tournament restricts registration to accounts with a chess rating of over 1700. This is a subnet-specific restriction defined at $\texttt{ISA}_{\texttt{C}_\texttt{T}}$. In the reference implementation, the execution to $\texttt{P.IGA.}\textit{Deposit}\,(P/C_T,\ c)$ invokes a check on $\texttt{P.ISA}_{\texttt{C}_\texttt{T}}\texttt{.}\textit{Deposit}\,(a,\ c)$ that will return *true* only if $a.rating > 1700$, and $false$ otherwise. If the call returns $false$, then the top-down transaction is not propagated to $\texttt{C}_\texttt{T}$ and u will not be able to play the tournament. Otherwise, u will be able to join by depositing the cost c in `P`'s `IGA`.

`IGA`'s state. The state of `P`'s `IGA` consists of:

- `P`'s ID
- Minimum required total stake per child subnet
- List of child subnets' IDs
- Checkpoint period Δ
- List of children checkpoints
- State required for the execution of cross-net transactions, including the postbox.

We detail in Section 8.2 the execution of cross-net transactions for the reference implementation.

[**mv:** frankly, I am not following this... Please re-read and try to explain being more concrete. Examples help.][**arp:** Rewrote, lmk if still not clear]

8.2 Cross-net transactions

Postbox. In IPC, a transaction in a subnet may trigger a state change in another subnet not immediately adjacent. 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, as someone will have to pay the fees for that new transaction, but instead leaves the transaction in the postbox of that intermediate subnet. Only once an account pays 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.

Cross-net queues. Hence, in the current implementation, a transaction at the postbox has not been paid for, and it will not be propagated until an account pays for it. 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 the *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 ready to be propagated.

Example. For example, suppose an actor in a child subnet $P_G/S_{P_1}/S_{C_1}$ triggers a state change in its uncle subnet S_G/S_{P_2}. Then, the cross-net transaction tx that represents that state change must reach the grandparent through the intermediate parent subnet S_P that connects them. In the reference implementation, this means that first a bottom-up transaction $tx_{b_1}(tx)$ with tx as payload, S_{C_1} as source and S_{P_1} as recipient is sent to S_{P_1}'s postbox from S_{C_1}'s bottom-up registry. Once tx_{b_1} is ordered and executed at S_{P_1}, an account needs to pay for the cost of moving tx into S_{P_1}'s bottom-up registry, creating a new transaction $tx_{b_2}(tx)$ with tx as payload, S_{P_1} as source and S_G as recipient. Following, tx_{b_2} is ordered and executed at S_G, reaching the postbox at S_G. Finally, once an account pays to create a transaction $tx_{t_1}(tx)$ with tx as payload, S_G as source and S_{P_2} as recipient, can tx_{t_1} be ordered an executed, meaning that tx reaches its destination.

Nonce. Once a subnet orders and executes cross-net transactions, the `IGA` updates its state by increasing a nonce specific for the recipient subnet (i.e. the parent if bottom-up or the specific child if top-down). The nonce is necessary to avoid replay attacks. We detail now the particularities of top-down and bottom-up transactions.

8.2.1 Top-down transactions

***PoF* in top-down transactions.** In order to prevent inconsistencies across replicas, the IPC agent does not immediately propose the transaction to the child replica. Instead, the IPC agent batches top-down transactions in a *top-down checkpoint* that are consistently broadcast to other replicas every Δ_T period. As such, the PoF of a top-down transaction tx is obtained once enough signatures containing at least a supermajority of the voting power from child replicas are received for a batch containing tx (i.e. a certificate). In this consistent broadcast, replicas broadcast batches of top-down 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), and they in turn sign a batch if they consider all transactions of the batch as valid.

Consistency across child replicas. 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 provides the batch to the child replica for ordering and execution.

8.2.2 Bottom-up transactions

Batching at checkpoints. The child subnet aggregates bottom-up transactions from within its own subnet and those propagated from its children, and includes their CIDs in the next checkpoint. 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 at the parent (specifically, every Δ child blocks). 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. Since the CIDs of bottom-up transactions are included in the checkpoint, the execution of the checkpoint also depends on the validity of those transactions (recall that the parent's postbox is located at the parent's `IGA`). If the bottom-up transactions sent in the batch along with the checkpoint fail to execute, the execution of the entire checkpoint will fail.

Reusing checkpoints' *PoF*. Therefore, a child's transaction (or state) *is* 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[5]. This *PoF* is the same one used for checkpoints, as bottom-up transactions are batched there (see Section 8.3). In other words, given transaction tx, the *PoF* (tx) needed for bottom-up transactions is a CID to the latest block decided by the child's Trantor consensus protocol, which already contains a certificate to verify finality. The parent subnet considers the *PoF* valid if it contains signatures from replicas with at least 2/3 of the voting power in the child. The voting power is measured according to what is stored in `ISA` for the epoch containing tx.

8.3 Functionality

Checkpointing. We show in Algorithm 3 the main design choices of the reference implementation for the checkpointing functionality. A checkpoint 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:

- The IPC agent obtains a state snapshot from the child's subnet. At the moment, the state snapshot is a CID to the latest decided block of the child's subnet that contains a checkpoint, as not all Trantor's blocks contain a checkpoint. The *PoF* of the checkpoint is the certificate of the block.
- The IPC agent obtains the CIDs of all new grandchildren's checkpoints, and of the bottom-up transactions in the bottom-up registry.
- The IPC agent computes the checkpoint `chkp`. Recall that the *PoF* of the checkpoint is already the certificate of the latest decided block.
- The IPC agent submits $tx = \texttt{P.ISA}_{\texttt{C}}\texttt{.}\mathit{Checkpoint}\,(\texttt{chkp})$.

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

- The parent subnet, upon ordering and executing tx, calls the `ISA` which verifies the PoF and if all checks pass then calls `P.IGA.Checkpoint(P/C, chkp)` for the `IGA` to save the checkpoint in the state and execute all cross-net transactions batched with the checkpoint.

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

Algorithm 3: Checkpoints IPC reference implementation

```
1  ▶ IPC agent:
2      upon newBlock from subnet C do
3          if newBlock.blockheight mod Δ = 0 then
4              if P.ISA_C.isReplica(Self) then
5                  chkpData ← newBlock.GetCID()
6                  gcChkps ← C.IGA.NewChkps()                    // grandchildren's chkps
7                  BUpTxs ← C.IGA.BottomUpRegistry
8                  chkp ← createChkp(chkpData, gcChkps, BUpTxs)
9                  submit P.IGA.Checkpoint(P/C, chkp)
10 ▶ P.IGA.Checkpoint(P/C, chkp):
11     if P.ISA_C.Checkpoint(chkp) then
12         Save chkp in the state
13         Execute tx, ∀ tx ∈ chkp.BUpTxs Save chkp in the state [arp: Why in both?]
14 ▶ P.ISA_C.Checkpoint(chkp):
       // [subnet-specific treatment of checkpoints]
15     return P.ISA_C.verify(chkp)                               // PoF is newBlock's certificate
```

Although omitted in Algorithm 3, executing each `tx` in *chkp*.`BUpTxs` may require additional checks. An example is a withdrawal of funds or a release of collateral for a replica leaving the subnet, that will require a call to check validity of the withdrawal transaction by calling the subnets $\texttt{ISA}_\texttt{C}$, in addition to the checks `IGA` already performs on withdrawals and release of collateral. Recall that the checkpoint operation is only atomically executed if all transactions of the batch are valid and are atomically executed as well.

Create. 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`. This registration in the `IGA` is what allows this subnet to interact with the rest of subnets registered in IPC through `IGA`, and thus we refer to it as the creation of the subnet.

Join and leave. Replicas join the replica set by depositing the required collateral in the `IGA` (the minimum amount of collateral per replica is defined in the `ISA`), collateral that they recover by leaving the subnet (provided they have not been slashed before).

Slashing. The IPC team is currently working on designing and implementating a mechanism for slashing. The current MVP does not natively support slashing at the parent.

Removing a subnet and retrieving funds. An active subnet can be removed by a majority of current replicas, releasing all the collateral and the circulating supply back at the parent. This provides suers and replicas with an orderly way of retrieving their funds at the aprent, other than having their funds retrievable from the latest snapshot at the parent once enough replicas recover their collateral and leave the subnet inactive.

8.4 Incentives

In the current reference implementation, replicas get rewarded for executing the checkpoint algorithm by charging an IPC fee on all transactions forwarded from the cross-net registry. This incentivizes replicas in participating on the checkpointing functionality, even if that costs them a fee to be paid for the transaction at the parent. All transactions batched with checkpoints 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 is ordered and executed, which requires collaboration from at least a supermajority of replicas for the reward to be paid. However, no specific incentives are given at the moment to the submitter(s) of the checkpoint transaction, or to the signers of the checkpoint certificate, in that the sum of IPC fees of the batch is evenly distributed proportionally to the stake of each replica set. 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.

9 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, `ISA`.$verifyGlobalFinality(tx, PoF)$ 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]