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 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 [**af:** Maybe worth hinting at why we can assume such a thing (e.g. running a full node on the parent); at least I think some readers might start thinking about adversarial scenarios to break this assumption.] 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. Section 2 introduces an sample use case of IPC that we use throughout as a running example. In Section 3 we define preliminary concepts and assumptions used by IPC. Sections 4 and 5 respectively describe IPC's functionality in terms of the interaction between a parent and a child subnet and summarize the high-level implementation of the two actors IPC uses to enable this interaction. We discuss participants' incentives to spend the resources required for inter-subnet interaction in Section 6 and related work in Section 7. Finally, our implementation of IPC and its deviations from the general design are laid out in Section 8.

2 Example Use Case: On-Line Game Platform

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

Rootnet with all users' funds (`L1`). The rootnet is used as a financial settlement layer. Most users' coins are on accounts residing in the rootnet's replicated state. A robust established blockchain system like Filecoin 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 interact directly with it.

Gaming platform as a subnet (`L2`). The functionality of the gaming platform (such as maintaining score boards, recommending opponents to players, or organizing tournaments) is implemented as a distributed application on a dedicated subnet. This subnet uses a significantly faster BFT-style consensus protocol (such as Trantor) since the application needs to be responsive for the sake of user experience, and deals, in general, with fewer funds than the rootnet (only as much as users dedicate to playing). The replicas constituting this subnet are run by gaming clubs or even some (not necessarily all) individual players (who do not necessarily trust each other, e.g. to not manipulate the score boards). To have a replica in the `L2` subnet, the club (or the player) needs to lock a certain amount of funds as collateral that can be slashed by the system if the replica misbehaves. The collateral / slashing mechanism is described in more detail in **??** and Section 4.7.3.

Individual games (`L3`). For each individual game, a new child of the `L2` subnet is created (Section 4.1) and acts as a (distributed) game server. Since not much is usually at stake in a single game and only few players are involved, the whole `L3` subnet may even be implemented by a single server the players trust. However, this decision is completely up to the players and they may choose a different implementation of the `L3` subnet when starting the game (by submitting the corresponding transactions to the `L2` subnet). When the game finishes, its result is automatically reported to the `L2` subnet (Section 4.5), which updates the players' ratings accordingly, and the `L3` subnet is disposed of (Section 4.6).

Player accounts. Each player has an account on the `L2` subnet where they deposit funds (Section 4.2) from the rootnet by submitting a corresponding `L1` transaction[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) by submitting an `L2` transaction.

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

This simple use case utilizes most of IPC's features. Throughout the rest of the document, we will use the on-line 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.)][**af:** I would find it interesting to discuss mechanisms to ensure trust in the results coming from the subnets, so no player can artificially boost their standing.][**js:** the weakness of the example is really how far from typical blockchain use cases it is; but I think it's a good proxy for explaining things, and I like where this is going.]

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 note, the vocabulary seems to be defined throughout the body. If we're defining in the body anyway, the appendix seems redundant – this isn't a book.] [**mv:** If the glossary is stable, I suggest bringing it to the main body of the paper by defining concepts inline, as we go. Glossary can additionally stay in the appendix, for quick reference.]

Basic abstractions. A *subnet* consists of multiple *replicas*, yet we abstract a subnet as a single entity which maintains an abstraction of *replicated state* (of which each replica maintains a copy) that can only be modified through transactions submitted either by *users* or by an *IPC agent*. It is the state (and only the state) that all replicas agree on. [**af:** What about cron, and similar end-of-epoch changes? Although I saw cron will be removed from Filecoin.] [**js:** There are proposals to fix some cron issues (https://github.com/filecoin-project/FIPs/discussions/638), but I don't think the current direction is towards removing it; in fact, I think there are very strong opinions against that solution.] [**mp:** Even that must be bound to blocks. As I understand it, it is some actions triggered every k blocks. That can be easily modeled by every k-th block containing an (implicit) transaction triggering the action (submitted by

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

some implicit system-level user).] We further abstract away the concrete mechanism of transaction submission and execution, as it is specific to the implementation of each particular subnet.

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

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

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

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

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

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

[**mv:** but what if it is - esp at longest chain parent net?] [**arp:** what happens if state is rolled back before a checkpoint for example?. If we are talking about payments, if checkpoint occurred and then state rolled back then whoever got its money back (due to the rollback) cannot withdraw that money to the parent anymore. If we are talking about state then behavior is undefined.] [**mp:** If the state is rolled back despite a *PoF*, then the assumptions underlying the verification of the *PoF* were wrong and the verifier treats the subnet as faulty (e.g. the same way as if the adversary started controlling the majority of its replicas / conputing poser / stake ...)] [**af:** There's always that word "assume"... perhaps what's missing from the defintion of PoF is that violations are detectable, e.g. by checkpointing.] [**mp:** Are they though in all the imaginable cases? I'm not quite sure. I assume that by "violation" you mean a state with a valid *PoF* being rolled back. Is that the case?] [**js:** I think I +1 Akosh on "assume". If we're saying a rollback is a violation, then this isn't a question of assumptions – it's a question of definition. Something like "... proof that a subnet reached *state* and cannot be rolled back without triggering a violation."?]. [**mp:** I just removed the second part of the sentence to avoid this confusion. It's quite clear from the definition of a *PoF* (it "convinces the verifier that the state will not be rolled back") and

removes the necessity of any kind of assumptions. I.e., only the verifier locally makes whatever assumptions they need to be reasonably "convinced" by the *PoF*.] We also denote by $PoF(tx)$ the proof that a subnet reached a state in which transaction tx already has been applied to the replicated state.

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

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`*`(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] We represent this value by associating some number of coins (also referred to as funds) with accounts and actors in a subnet's replicated state. Each user is assumed to have an account in each subnet the user interacts with. All the transactions spending coins from an account must be signed by the corresponding user's private key. For simplicity of presentation, we do not explicitly include these signatures in the further description of IPC.

We also assume that the submission, ordering, and application of transactions is associated with a cost (known as transaction fees, or *gas*). Each subnet client (user wallet or IPC agent) submitting a transaction to a subnet must have an account in that subnet, from which this cost is deducted. If the funds are insufficient, the subnet may fail to execute the transaction.

Note that the operation of IPC requires the submission and processing of transactions that are not easily attributed to a concrete user. This is the case with transactions that an IPC agent submits on behalf of a whole subnet. We discuss incentivization of participants to run IPC agents and pay for the associated transaction fees in **??**. [**TODO:** Move this part after IPC agent and actors, as it already uses thos terms.]

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

1. The `IGA` is an actor that contains all IPC-related information and logic associated with a subnet that needs to be replicated *in the subnet itself*. Each subnet contains exactly one `IGA`. We describe the `IGA` in detail in Section 5.1 [**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)?] [**mp:** Those questions should be answered in Section 5.1]

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

[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.

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

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

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

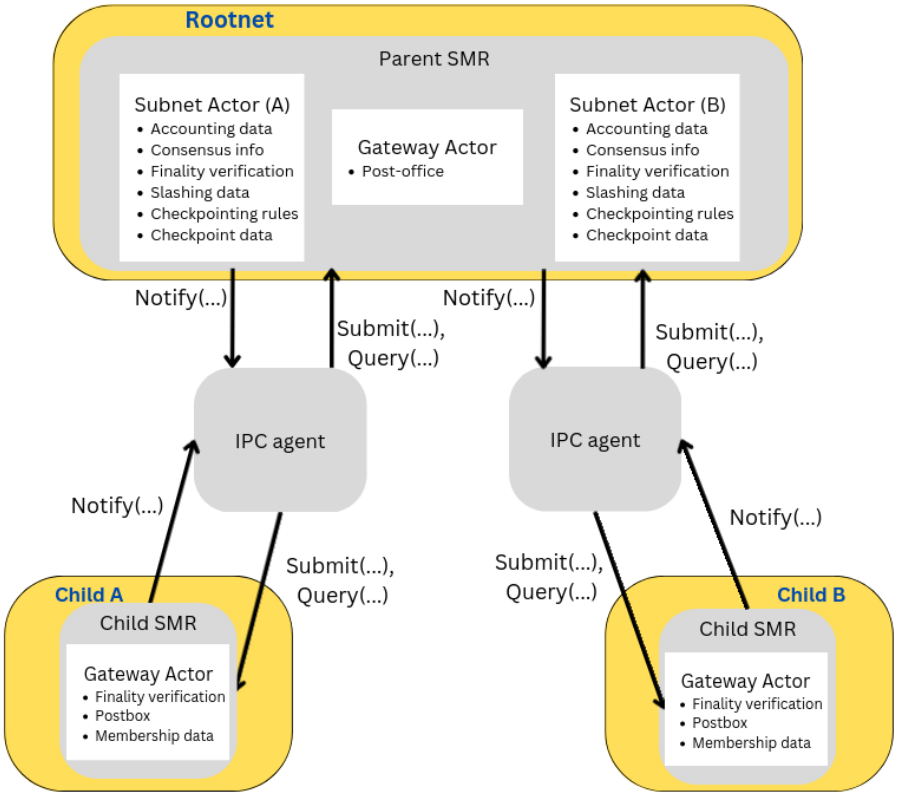

Figure 1: The basic IPC components and their interfaces in an example with one parent and 2 child subnets (A and B). [**TODO:** Update figure to be consistent with new notation.]

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

The key to providing incentives for IPC agents and replicas is that the `ISA` and the `IGA` can, as actors, also hold funds that their logic can distribute among other accounts or actors on their respective subnets. Thus, when an IPC agent submits a transaction to one of these actors, they can be configured to reimburse (potentially adding an extra reward) the account from which the transaction fee was paid (i.e., an account associated with the IPC agent)

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

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.

[**TODO:** Add example of gaming system incentives: The replicas and IPC agents running the `L2` subnet might get a commission from transaction fees, while the single-game subnets might not need explicit incentivization at all, as they are mostly run by players, whose incentive is being able to play the game.]

[**TODO:** Matej: Describe firewall property (and trust model in general).]

[**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. The IPC interface exposes the following functionalities:

1. Creating child subnets in the IPC hierarchy.
   E.g., when starting a new game on the gaming platform.

2. Depositing funds from an account in a subnet to an account in its child.
   E.g., when a player tops up the balance of their account on the gaming platform.

3. Withdrawing funds from an account in a subnet to an account in its parent.
   E.g., when a player withdraws money they won in a tournament.

4. Checkpointing a subnet's replicated state in the replicated state of its parent.
   E.g., after every 100 blocks of transactions applied to the gaming platform's subnet.

5. Invoking actor functions across subnets, i.e., the replicated logic of one subnet acting as a client of another subnet.
   E.g., when a game finishes and its the involved players' rankings are automatically updated.

6. Removing child subnets from the IPC hierarchy.
   E.g., when a game fihishes, rankings have been updated, and the state of the game can be disposed of.

7. Managing proof-of-stake subnets, exposing functions for adding and removing replicas, managing the associated collateral, and slashing of provably misbehaving replicas.
   E.g., when mutually distrusting players play together.

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

4.1 Creating a child subnet

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

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

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

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

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

Example 1. *Imagine that a player wants to create a game subnet to play against 3 opponents, such that each player will run their own replica in the game subnet (i.e., their own copy of the game server). As an incentive for honest behavior, the player decides that the child subnet will be PoS-based (see Section 4.7), where each replica must be backed by a minimal collateral of 10 coins that would be slashed (and, say, redistributed to the other players) if the replica is caught misbehaving (see Section 4.7.3).*

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

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

4.2 Depositing funds

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

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

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 `ISA.`*`Deposit`*`(`*amount, account*`)` has been applied to the parent subnet's replicated state and that state is final (i.e., cannot be rolled back).

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

1. The owner of `P.a` submits, using their wallet, a transaction
   tx = `P.ISA`$_{\texttt{C}}$`.Deposit`(*amount*, `b`).
2. `P` orders and executes tx, transferring *amount* coins from `a` to `ISA`$_{\texttt{C}}$.
3. When `P`'s replicated state that includes tx becomes final (for some subnet-specific definition of finality provable to `P/C.IGA`, which contains the PoF verification logic), the IPC agent constructs a $PoF(tx)$.
4. The IPC agent submits a transaction
   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 `P.ISA`$_{\texttt{C}}$. However, those coins can effectively only be used by the owner of `P/C.b`, since `P.ISA`$_{\texttt{C}}$ will not transfer its coins within `P` until they are burned in `P/C` during a withdrawal operation (see below).

Example 2. *Imagine that a player wants to start using the gaming platform (running on the subnet `P/C`), but only has funds in an account `P.a` in the parent subnet. To be able to join games that require a collateral (as in Example 1), the player decides to fund their account `P/C.b` with 20 coins. Thus, the player submits a `P.ISA`$_{\texttt{C}}$`.Deposit(20, b)` transaction, starts an IPC agent process that performs steps 3 and 4 above, and waits until the funds appear on `P/C.b`.*

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 `P.ISA`$_{\texttt{C}}$), The IPC agent constructs a $PoF(tx)$.

4. The IPC agent submits a transaction[3]
   tx' = P.ISA$_C$.*ReleaseWithdrawn*(*amount*, a, *PoF*(*tx*)).

5. P orders and executes *tx'*, which results in P.ISA$_C$ 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 P.ISA$_C$ 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.

Example 3. *A player might want to stop using the gaming platform and withdraw all the funds back to the parent subnet, in order to spend them on something else. They still have 20 coins on their account P/C.b that they want to transfer back to P.a. The player performs the withdrawal by submitting a transaction P/C.IGA.Withdraw(20, a), starts an IPC agent process to perform the necessary inter-subnet communication, and waits until the coins arrive at P.a. (The player might need to spend a part of the 20 coins on fees for both the Withdraw and the ReleaseWithdrawn transactions.)*

4.4 Checkpointing

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

ISA.*Checkpoint*(*snapshot*, *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 = P.ISA$_C$.*Checkpoint*(*state*, *PoF*(*state*)).

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

[3] In a practical implementation, instead of submitting a separate transaction for each withdrawal, the IPC agent may submit multiple withdrawals batched in a single transaction, with a single *PoF* proving the finality of the child state in which all the corresponding funds have been burned. This optimizes both performance and cost (transaction fees) at the parent. Our implementation applies this optimization, further combined with checkpoints.

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`.

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

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

1. An actor `P/A.ActorA` constructs a tarnsaction
   $tx = \texttt{P/B.ActorB.}\textit{SomeFunction}(someParams)$

2. `P/A.ActorA` invokes the funcion $\texttt{P/A.IGA.}\textit{Dispatch}(tx, \texttt{P/A}, \texttt{P/B})$ (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
   $tx_{\texttt{A}} = \texttt{P.IGA.}\textit{Propagate}(tx, \texttt{P/A}, \texttt{P/B}, PoF(state_{\texttt{A}}))$.

5. Subnet `P` orders and executes $tx_{\texttt{A}}$, verifying $PoF(state_{\texttt{A}})$ and (internally) invoking $\texttt{P.IGA.}\textit{Dispatch}(tx, \texttt{P/A}, \texttt{P/B})$. 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
   $tx_{\texttt{P}} = \texttt{P/B.IGA.}\textit{Propagate}(tx, \texttt{P/A}, \texttt{P/B}, PoF(state_{\texttt{P}}))$, 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.

Example 4. *In our gaming example, imagine that a game (running in its own subnet that is a child of the gaming platform's subnet) has finished and the ranking of the involved players needs to be updated. The game server is implemented as an actor on the game's own subnet, while the gaming platform (storing the player ranking tables) is an actor of its parent. To update the ranking, the game actor would use a cross-net transaction to inform the platform actor about the results of the game and the platform actor would update the rankings accordingly.*

4.6 Removing a child subnet

TODO

4.7 Proof-of-stake subnets

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

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

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

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

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

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

Example 1 provides a concrete case of where and how PoS-based subnets can be used.

4.7.1 Staking collateral

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

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

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

4.7.2 Releasing collateral

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

1. The owner of `P.a` submits the transaction
   tx = $\texttt{P.ISA}_\texttt{C}$.*RequestCollateral* (*replica*, *amount*, `a`),
   where *amount* is the amount of funds the owner of `P.a` wants to reclaim.

2. After ordering *tx* and checking that `P.a` has indeed previously staked at least *amount* for *replica*, $\texttt{P.ISA}_\texttt{C}$ decreases the collateral associated with *replica* by *amount*. Note that `P.ISA` does not yet transfer *amount* back to `P.a`

3. The IPC agent, upon detecting the updated *membership* in $\texttt{P.ISA}_\texttt{C}$ through the application of *tx*, constructs a $PoF(tx)$ and submits the transaction
   tx' = `P/C.IGA`.*UpdateMembership* (*membership*, $PoF(tx)$).

4. Upon ordering *tx'* and successfully verifying $PoF(tx)$, `P/C.IGA` updates its target membership.

5. Subnet `P/C` reconfigures (the reconfiguration procedure is specific to the implementation of the subnet and its ordering protocol) to use the new *membership* as its current membership.

6. The IPC agent detects that the current membership of `P/C` has changed. Let *state* be the the replicated state of `P/C` where the current membership has already been updated.

7. The IPC agent constructs a $PoF(state)$ and submits the transaction
   tx" = $\texttt{P.ISA}_\texttt{C}$.*ReleaseCollateral* (*membership*, $PoF(state)$)

8. Upon ordering *tx"* and successfully verifying $PoF(tx)$, $\texttt{P.ISA}_\texttt{C}$ verifies that the received *membership* reflects the requested change in the collateral of *replica* and, if this is the case, transfers *amount* to `P.a`.

4.7.3 Slashing a misbehaving replica

Slashing is a penalty imposed on provably malicious replicas in 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 $tx = \texttt{P.ISA}_\texttt{C}.\textit{Slash}(\textit{replica}, \textit{PoM})$

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

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

Example 5. *Imagine the setting from Example 1, where a player creates a PoS-based subnet for a game of 4 players with a minimal collateral of 10 coins. Suppose that one of the players makes a move in the game, but later realizes that it would be beneficial to revert that move. What is more, the majority of other players would also benefit from the game state being reverted to the state just before the move occurred. Those players might collude and agree off-band to revert their replicas of the game server to an older state, and propose (in the child's ordering protocol) a different transaction to be ordered instead of the one containing the original game move.*

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

5 IPC 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 the parent subnet. 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.

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
```

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.
- 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 (*childMembership*), how much collateral (and for which replica) is staked from which account (*collateral*), and which accounts requested the withdrawal of how much collateral (*collateralRequests*). Moreover, the `ISA`'s state contains a predicate for checking the validity of proofs of misbehavior (*validPom*) and a procedure to execute when a valid PoM is received through the `Slash` function. Both *validPoM* and *slashingPolicy* are part of *params* passed to `IGA.CreateChild` when the `ISA` is created.

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.] [**mp:** You are right, this section just had a lot of redundancy and was too bloated. I condensed and moved the essential parts of it where they more naturally belong (Preliminaries and PoS subnets). An example in the Preliminaries section will be the next thing I write.]

7 Related Work

[**TODO:** Structure is there, TODO add citations (and add them throughout document too)]

Algorithm 2: IPC Subnet Actor (`ISA`)

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

7.1 Consensus protocols

Recent developments in asynchronous consensus protocols have significantly increased throughput by decoupling data dissemination from metadata ordering[**arp:** cite and talk a bit about Dumbo, Bullshark, Aptos, Sui.]. These solutions offer vertical scaling and are orthogonal to the horizontal scaling proposed by IPC, in that each subnet can benefit from these advances at their consensus level. Also, these proposals in isolation are inherently restricted by the need to replicate all state updates across all members of the consensus protocol.

7.2 ZK and optimistic rollups.

Rollups scale blockchains by having third parties (sequencers) locally order and execute batches of transactions and posting only the result of the batch in the blockchain. Optimistic rollups rely on the result of a sequencer allowing for a predefined dispute period. If disputed, the sequencer is not punished only if it proves within a limited amount of time, via executing the batch in the blockchain, that the result of its batch matches its submitted result. ZK rollups cryptographically generate a succint proof of the correctness of the result for fast verification.

To the best of our knowledge, all ZK rollups proposed to date rely on centralized sequencers, a problem not trivial to solve as rollups suffer from the need to assign disjoint batches for concurrent execution across different sequencers. [**arp:** add citations.]

7.3 Sharding

Sharding solutions periodically partition blockchain state across different group of replicas. Unlike subnets, shards do not conceptually differ from each other, leading to a potential increase of cross-shards communication compared to subnets, where users and services partition the state on demand. [**arp:** add citations]

7.4 State and payment channels

State channels scale blockchains by having participants lock their state in the blockchain (on-chain) among them in order to update locally the state (off-chain) with which to release the funds back on-chain. Payment channels are the analogous solutions for channels where the state are coins in the UTXO model. Channels can form network graphs in which nodes can relay payments in exchange of a fee. Though many works, particularly in state channels, are addressing significant problems specific to channels, IPC subnets are a generalization of channels[**arp:** add citations and detail some of them, then update this sentence]. This is because channels require all parties to sign in order to update the channel's state off-chain, whereas subnets (like the ones we show in this document) allow for a variable quorum size. [**arp:** add citations.]

7.5 Subnets networks

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

Single-layer networks. Polygon scales the Ethereum blockchain by using a network of subnets (sidechains in the Polygon terminology) connected to the Ethereum mainnet, with a developing framework for services to create their own subnet. Avalanche offers a similar solution for its rootnet with Avalanche subnets, a network of subnets that are connected to their rootnet for further scaling and customized applications. Yet, Polygon PoS offers one subnet, and neither Polygon nor Avalanche offer native cross-net communication protocols. Polkadot's relay-chain connects Polkadots parachains with native cross-net transactions, in a topology that also resembles a single scaling layer of direct child subnets of a rootnet. They also offer especialized bridges to other blockchains, like that of Bitcoin or Ethereum. Similarly to IPC, Polkadot's parachains rely on anchoring their compressed state to the parent for total ordering and cross-net interactions, which limits the capabilities of horizontal scaling: parachains need to constantly lease an auctioned slot on the relay-chain for a specific period, a problem that is exarcebated in a single layer of child subnets checkpointing to the same parent compared to IPC's tree topology. Internet Computer Protocol's (ICP) subnets are independent blockchains with the ICP rootnet as the common parent, and whose committees are subcommittees of the rootnet's validator set. ICP subnets are tightly integrated with its rootnet and dedicated to execute smart contracts seamlessly. Unfortunately, none of these works consider a structure beyond a single layer of direct child with their respective rootnet as the only parent.

Topology-agnostic networks. Cosmos zones are independent blockchains that interact in a general graph not necessarily in the topology of a tree, with a reference implementation that spawns new zones running Tendermint as the consensus protocol. Cosmos relies on the single

finality of Tendermint for inter-blockchain communication (IBC), with the Cosmos hub as an intermediary to facilitate IBC interactions, similarly to the Polkadot relay-chain. The flexibility for any topology between subnets comes with the need to have further requirements for IBC, like having nodes of a blockchain to run light client of each other blockchain they want to interact with, or to require fast finality. [**arp:** TODO PoS sidechains mention]

UTXO-based subnets. Plasma chains horizontally scale payments via multiple subnets (childchains) organized in a tree-like structure. It is, along with PoS sidechains, one of the first works to consider subnet-like interactions between blockchains. Given the topology, many of the challenges that IPC subnets face are present in Plasma chains. However, Plasma chains are UTXO-based blockchains focused on payments. In this sense, IPC subnets generalize the problem that Plasma chains are trying to solve, allowing state to be transacted in subnets, and not just payments. Also, Plasma chains rely on synchrony for withdrawals with a dispute resolution period, similarly to state and payment channels, and optimistic rollups, which pose an unreconcilable trade-off between finality latency and security[**arp:** Cite platypus], a problem that is theoretically solve by the partially synchronous Platypus, that approaches the PoF shown for IPC subnets running Trantor. IPC subnets also generalize Plasma's and Platypus' withdrawal mechanisms by allowing subnets to define what constitutes a proof of finality, a slashing rule and a fraud proof.

8 IPC's reference implementation

[**js:** I like the idea of the section but it currently reads a little weird, particularly beyond 8.1. There is this list of components/topics and a number of entries under them, but no narrative and, for many of the entries, it's unclear whether the implementation details actually add useful details, vs. just repeating the design or listing implementation "facts" without explaining their relevance. I don't know if the solution is less content of more content, but I might start with making it less "entry"-based and more narrative, focusing on things that are either very relevant/important os potentially unexpected and hence deserving of an explanation. I think e.g. 8.4 (incentives) is written in a more useful style.]

In this section, we describe the particular choices implemented by the ConsensusLab team for the reference implementation of IPC. It is not the purpose of this section to comprehensively describe the reference implementation, but to list the relevant differences with the description of IPC made in previous sections.

8.1 Preliminaries

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

The reference implementation makes use of IPFS-style content addressing, in that data is stored where relevant and referred to with a Multiformats-compliant 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 running full nodes of the subnet. This means that if a subnet only has faulty participants, the content referred to by this CID may

not be available. However, this is not a problem, as this would just mean that operations that have this subnet as source will not be resolved, not affecting the rest of the IPC tree.

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

8.2 Components

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

8.2.1 IPC agent

In the previous sections, we considered that every parent-child pairing had an independent IPC agent process. In fact, the implementation manages to execute one single IPC agent for the entire tree of subnets that may be of relevance to a participant. [**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.

8.2.2 `IGA` as entry point

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

As a result, in the reference implementation, `IGA` contains the locked funds of each subnet, i.e. the subnet's *circulating supply* (unlike in Section 5, where the `ISA` held the locked funds). The circulating supply of each child subnet is stored in a map at the `IGA`, where the key is the subnet ID. As withdrawals contain a *PoF*, the circulating supply suffices for the firewall property[**TODO:** Alex: add firewall property (and other properties). @matej is this on your plate or should I get on it?] [**mp:** Yes, the firewall property is part of the trust model, which is on my plate.]

8.2.3 `ISA` for subnet customization

In the reference implementation, the subnet actor of subnet `C` holds the state specific to subnet `C`. The aforementioned entry point for all funcionality is the `IGA` (explained in Section 8.2.2), and not the `ISA`. However, a subnet can augment the default functionality of the `IGA` in the `ISA`. In particular, the `ISA` can include conditions for the validation of proofs of finality, releases of funds and stake, and slashing rules[4].

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

8.3 Functionality

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

8.4 Deposits

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

1. The owner of `P.a` submits a transaction $tx =$ `P.IGA.`*`Deposit`*$($`P/C`, *b, amount, fee*$)$.
2. tx is ordered and executed at `P`. The ordering and execution of tx is as follows:
   - `IGA` checks that *fee* is above a hard-coded minimum IPC base fee. The parameter *fee* is an amount of coins to be paid to the child replicas to incentivize them to participate in the validation of top-down transactions for the child (see Section 8.7)[5].
   - *amount* is deposited in the `IGA` of the parent subnet.
   - `IGA` creates a top-down transaction $tx =$ `P/C.IGA.`*`MintDeposited`*$($*b, amount, fee*$)$ and stores it in the top-down registry.
   - The top-down transaction tx' is ordered and executed at the child subnet, resulting in the minting of *amount* sent to account `P/C.b`. We detail further the ordering and execution of top-down transactions in Section 8.7.

8.5 Withdrawals

Analogously to deposits, withdrawals must carry an IPC fee to be paid to the child replicas. In the case of bottom-up transactions, though, the procedure is analogous to the one describe in Section 4, with the exception that the `ISA` of the child subnet does not hold the funds, and thus it relays the release of the funds to the `IGA` at the parent. More concretely, withdrawing *amount* coins from an account `P/C.b` in the child subnet `P/C` to an account `P.a` in the parent subnet `P` involves the following steps:

- The owner of `P/C.b` submits a transaction $tx =$ `P/C.IGA.`*`Withdraw`*$($*amount, a, fee*$)$.
- tx is ordered and executed at `P/C`. The ordering and execution of tx is as follows:
  - `IGA` checks that *fee* is above a hard-coded minimum IPC base fee. The parameter *fee* is an amount of coins to be paid to the child replicas to incentivize them to participate in the validation of bottom-up transactions for the child (see Section 8.7).
  - *amount* is burned from `b`.
  - `IGA` creates a bottom-up transaction $tx =$ `P.`$\mathtt{ISA}_{\mathtt{C}}$`.`*`ReleaseWithdrawn`*$($*amount, b, fee*$)$ and stores it in the bottom-up registry.

[5]The IPC fee is different from and in addition to transaction fees for replicas to order and execute transactions in a subnet, and which we omit throughout the document for ease of exposition.

– The bottom-up transaction tx' is ordered and executed at the parent subnet. This results in $\texttt{ISA}_\texttt{C}$ calling `IGA` to release *amount* and send it to account `P.a`. We detail further the ordering and execution of bottom-up transactions in Section 8.7.

8.6 Checkpointing

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

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

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

3. The IPC agent submits $tx =$`P.`$\texttt{ISA}_\texttt{C}$.Checkpoint(`P/C`, *chkpCID, gcChkps, BUpTxs*). Recall that the *PoF* of a checkpoint is already contained in the block's CID.

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

8.7 Propagating cross-net transactions

Similarly to the description in Section 4, the current reference implementation uses the postbox in each `IGA` to split cross-net transactions between subnets not immediately adjacent into multiple cross-net transactions in the parent-child hierarchy. As explained in previous sections, a cross-net transaction tx is propagated to each immediately adjacent subnet along the path to its destination until it reaches the destination by traversing through the postbox of all intermediate subnets, via cross-net transactions $tx'(tx)$ containing tx as payload.

However, once tx' is ordered and executed at an intermediate subnet `S`, an IPC agent must pay for the cost to pay the fees for ordering and executing another transaction $tx''(tx)$ in `S` so as to move the state to the next subnet along the path. Instead, tx is left in the postbox of that intermediate subnet until the account that originally triggered the cross-net transaction creates tx'' and pays for the fees required to execute it[6]. This process is repeated until tx reaches its destination subnet.

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

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

Top-down transactions. In order to prevent inconsistencies across replicas, the IPC agent does not immediately submit a top-down transaction to the child subnet. Instead, the IPC agent batches top-down transactions in a *top-down checkpoint* that is consistently broadcast to other replicas every Δ_T blocks. 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. 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.

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

Bottom-up transactions. The child subnet aggregates bottom-up transactions attaching their corresponding CIDs to the next checkpoint, along with an increasing nonce value per CID that is unique for each parent-child pair[7]. The CIDs of these bottom-up transactions are placed in the `IGA` of the child. Bottom-up transactions are stored in the `IGA` of the child until it is time to checkpoint to the parent (see Section 8.6). This way, the `IGA` serves as the single location for the CIDs of bottom-up transactions and the IPC agent only needs to monitor the `IGA` to get all necessary information from the child subnet. As show in Section 8.6, since the CIDs of bottom-up transactions are attached to the checkpoint, the execution of the checkpoint transaction also depends on the validity of those transactions.

8.8 Creating and removing a child subnet

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

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

8.9 Staking and releasing collateral

[**arp:** TODO verify with Matej and Alfonso, something is missing here] Contrary to Section 4, staking collateral does not require a explicit call to update any state in the `IGA` of the child subnet. A replica simply increases its stake with account `P.a` by submitting a transaction $tx =$ $\texttt{P.ISA}_\texttt{C}$.*`StakeCollateral`*(`P.a`, *replica*, *amount*). This transaction stakes the collateral in the `IGA` of the parent `P`. Notwithstanding, it does not incur a reconfiguration in the weights of the replica set running the Trantor protocol at the child subnet `P/C`, but it is instead up to the current replica set at `P/C` to decide when and how to update their membership to reflect this change of the weighted voting, and notify the new reconfiguration to `P.IGA` with a bottom-up transaction.

A reconfiguration at the child subnet being notified at the parent can also trigger a release of collateral to reflect this update in the weighted voting of the child subnet. It is possible

[7] the nonce value is in place to prevent replay attacks

that a reconfiguration triggers a release of collateral that drops the staked collateral below the minimum threshold for subnet creation. IPC subnets of the reference implementation must always hold this minimum amount of collateral per subnet (defined in the `IGA`). If, after it has been created and registered, the subnet's collateral drops below the required minimum, the subnet enters an *inactive* state. This means that the subnet can no longer interact with the rest of the active subnets registered in the `IGA`, or even withdraw funds back to the parent subnet. In this case, though, users and remaining replicas can stake enough collateral to reactivate the subnet.

8.10 Incentives

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

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]