Interplanetary Consensus (IPC)

Consensus Lab

Abstract

1 Introduction

A blockchain system is a platform for hosting replicated applications (a.k.a. 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 (a.k.a. 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). It is true that many other applications are often involved in transferring assets and asset transfer 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 typically tend to cluster in a way that users 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 it 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), where most of its state is ephemeral (until the end of the game), but 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.

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 SMR/blockchain platforms, while still allowing them to interact in a secure way. The basic idea behind IPC is dynamically deploying separate, loosely coupled SMR/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 subnet-specific definition of correctness) by its child.

To facilitate the interaction between different subnets, IPC provides mechanisms for communication between them. In particular:

1. Assuming a common notion of money (assets / tokens / ...) between a parent and a child and accounts that can hold them, IPC also defines

how money is moved between accounts in different subnets.

2. We define the notion of a *checkpoint* as a snapshot of the state of a subnet after having processed a certain sequence of transactions. IPC enables child subnets to place references to their checkpoints inside the state of their parents.

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, validator set, and other subnet-specific data).
- "Inheriting" by the subnet of some of its parent's security and trustworthiness, by periodically anchoring its state (through checkpoints) in the state of the parent.

In the rest of this document, we describe IPC in detail. In section 2 we define the system model and introduce the necessary terminology. Section 3 describes the main components of IPC and their interfaces. [**mp:** TODO: Finish this when sections are written.]

2 Model

The vocabulary used throughout this document is summarized in Appendix A. The reader is encouraged to read Appendix A before continuing. [**TODO:** Go through and address remaining Marko's comments of this section]

2.1 Computation and failure model

We model IPC as a distributed ("message-passing") system consisting of *processes* that communicate by exchanging *messages*[1] over a network. In practice, a process is a program running on a computer, having some state, and reacting to external events and messages received over a communication network. We describe processes as exemplified in Algorithm 1.

[1] Network messages are not to be confused with Filecoin actor messages, that this document refers to as transactions.

Algorithm 1: Process definition.

```
variable = initial value
variable = initial value
...
▶ process:
    upon event(params...) do
        // Logic to execute atomically
    upon event(params...) do
        // Logic to execute atomically
    ...
```

A process that performs all the steps exactly as prescribed by the protocols it is participating in is *correct*. A process that stops performing any steps (i.e., *crashes*) or that deviates from the prescribed protocols in any way is *faulty*. If a process is correct or may only fail by crashing, it is *benign*. A non-benign process is *malicious*. [**mp:** We can remove terms we end up not using...]

In general, faulty processes can be malicious (Byzantine), i.e., we do not put any restrictions on their behavior, except being computationally bounded and thus not being able to subvert standard cryptographic primitives, such as forging signatures or inverting secure hash functions. If the implementation of some component in our design requires additional assumptions on the behavior of faulty processes, they will be stated explicitly.

We use the term *participant* to describe an entity participating in the system that controls one or more processes. All processes controlled by one participant are assumed to be in the same trust domain – they trust each other, i.e., assume each other's correctness. For example, a participant in the child subnet will probably run multiple processes: one for participating in the child subnet's protocol (child replica), one for participating in the parent subnet (parent replica), and one process that processes the information from the above two and submits transactions accordingly (IPC agent). We precisely define the replicas and the IPC agent (all of them being processes) in Sections 2.2 and 3. The IPC agent of a participant always assumes that the information it receives from "its own" child replica is correct. However, messages received from another participant's replica or IPC agent are seen as potentially malicious.

The synchrony assumptions may vary between different components of IPC. We thus state those assumptions whenever necessary, when describing concrete implementations of IPC components.

2.2 State machine replication (SMR) and smart contracts

SMR and replicated state. A *state machine replication (SMR) system*[2] is a system consisting of processes called *replicas*, each of which locally stores

[2] In this document, we use the terms "SMR system" and "blockchain" interchangeably.

a copy of (or at least has access to) *replicated state* that it updates over time by applying a sequence of *transactions* to it. Without specifying the details of it, we assume that any process can *submit* a transaction to an SMR system (we call such a process an *SMR client*) and that this transaction will eventually be ordered and applied to the replicated state. We call an SMR system that is part of IPC a *subnet*.

An SMR system guarantees to each correct replica that, after applying n transactions to its local copy of the replicated state, the latter will be identical to any other correct replica's copy of the replicated state after applying n transactions. The replicas achieve this by executing an *ordering protocol* to agree on a common sequence of transactions to apply to the replicated state.

Note that replicas do not necessarily all hold the same replicated state at any instant of real time, since each replica might be processing transactions at a different time. In this context, there is no such thing as "the current replicated state of the SMR system". There is only the current replicated state of a single replica. The replicated state of the system is only an abstract, logical construct useful for reasoning about transitions from one replicated state to another, happening at individual replicas by applying transactions (at different real times). When referring to a "current" replicated state, we mean the state resulting from the application of a certain number of transactions to the initial state.

Smart contracts. The replicated state of an SMR system can be logically subdivided into multiple *smart contracts* (a.k.a. actors in Filecoin). A smart contract is a portion of the replicated state with well-defined semantics. It defines the logic (e.g., expressed in a programming language, like Solidity in Ethereum) that a replica needs to execute when applying transactions and the new state that results from it.

We model a smart contract as a logical object in the replicated state that contains arbitrary variables representing its state. Its associated logic reacts to *events* triggered by (1) the application of transactions or (2) execution of other (or even own) smart contract logic. We describe smart contracts as exemplified in Algorithm 2.

Algorithm 2: smart contract definition

```
variable = initial value
variable = initial value
...
▶ smart contract name:
    Function(params...)
        // Logic to execute
    Function(params...)
        // Logic to execute
```

Note that a process usually represent OS-level processes running on some physical machine executing a program, smart contracts are an abstraction over the replicated state of an SMR system and their logic is being executed by all its replicas. While a process can submit a transaction to an SMR system, a smart contract cannot.

Naming. We assign each subnet a name that is unique among all the children of the same parent. Similarly to the notation used for absolute paths 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.

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

Proofs of finality. To enable interaction between subnets, we define a *proof of finality (PoF)* to be data that proves that an SMR system definitively reached a certain replicated state. Regardless of the SMR system's ordering protocol's approach to finality (e.g., immediate finality for classic BFT protocols, or probabilistic finality in PoW-based systems), a PoF convinces the the proof's verifier that the replicated state the PoF refers to will not be rolled back. For example, for a BFT-based SMR system, a quorum of signatures produced by its replicas can constitute a PoF. We denote by *PoF (tx)* the proof that an SMR system reached a state in which transaction *tx* already has been applied.

2.3 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.[3] Each end user of the SMR system is assumed to have a personal wallet and a corresponding account in some subnet.

We also assume that the submission, ordering, and applications of transactions is associated with a va;uable cost. Each SMR client submitting a

[3]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.

transaction to a subnet is assumed to have an account in that subnet, from which this cost is deducted. If the funds are insufficient, the SMR system ignores the transaction.

3 Components and their Interfaces

We now focus on the interaction between two subnets in a parent-child relation. To enable the interaction between them, which comprises running the subnets, observing each other's replicated state, constructing proofs of finality, submitting the necessary transactions, and modifying the replicated state accordingly, the following components need to work together. Figure 1 illustrates the components and their interfaces.

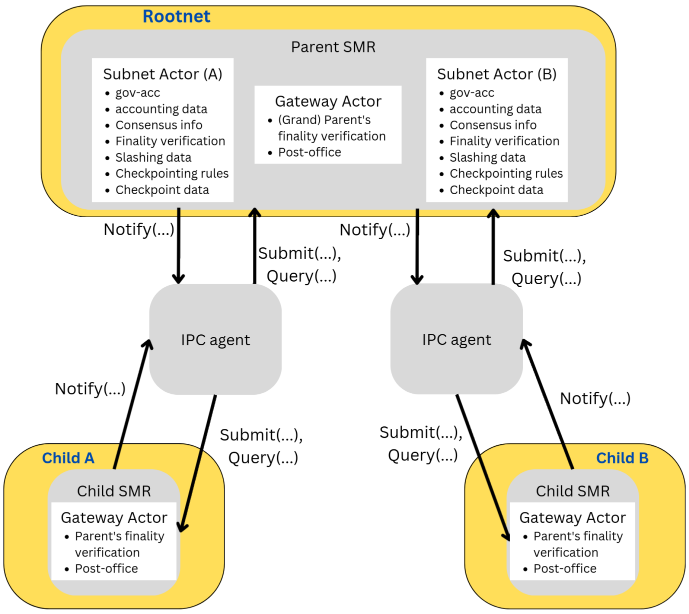

Figure 1: The basic components and their interfaces in an example with one parent (rootnet) and 2 child subnets (A and B).

3.1 Components

- Processes:

  1. **Parent replica:** The process that runs the parent subnet. It has a copy of the parent's replicated state, participates in receiving and ordering transactions and updates the replicated state (including the *SA* smart contract) accordingly.

2. **Child replica:** The process that runs the child subnet. It has a copy of the child's replicated state, participates in receiving and ordering transactions and updates the replicated state (including the GWA smart contract) accordingly.
3. **IPC agent:** The process that mediates the interactions between the two subnets. It has access to the replicated states of both the parent and the child (e.g., by sharing a trust domain with a child and a parent replica, or by securely downloading the replicated state from other replicas...) and acts as an SMR client (i.e. submits transactions) to both subnets. Depending on the implementation, it might also be responsible for constructing proofs of finality (which might involve communication with other processes).

- Smart contracts:
  1. **Subnet actor (*SA*):** The smart contract in the parent subnet's replicated state that stores all information about the child subnet that the parent subnet needs. The IPC agent's transactions submitted to the parent subnet mostly invoke the SA. The state of the SA includes:
     - *Accounting data.* This data describes the value that has been deposited to the child. It is considered locked inside the SA until it is withdrawn from the child. This data might consist of just a single value representing the sum of all such coins ("custodial" approach), but might also contain finer-grained information about balances for each account in the child subnet ("non-custodial" approach).
     - *Governance account.* This account facilitates the economic design of a subnet. It can be used for collecting fees or making payments to accounts of participants that perform operations on behalf of the child subnet. For example, when an IPC agent submits (and thus pays for) transaction linking a child's checkpoint to the parent's replicated state, the IPC actor logic might reimburse the associated account.
     - *Ordering protocol data.* This is the data (or a pointer to it) that is needed to run the ordering protocol of the child subnet. It is protocol-specific, but is generally expected to contain information such as
       * The ordering protocol used by the subnet.
       * Subnet configuration such as the validator set, voting rights, collateral deposits, etc.
       * Subnet governance mechanisms, e.g., transaction fees, block rewards, conditions for participation, ...

- *Child state finality verification.* Logic to verify (based on only the parent's replicated state) that a given child's replicated state/tx is final[4] or that a particular tx has been definitively included in the child's state. We expect that this logic will verify a PoF submitted (through transactions) by one or more IPC agents to the SA.
- *Slashing.*
  * List of slashable misbehaviors and a proving methodology. That is, for each slashable misbehavior there is a definition of what constitutes a valid proof of misbehavior (PoM).
  * Penalties for misbehavior and rewards for reporting, as well as the logic performing the necessary actions within the parent subnet.
- *Checkpointing.*
  * Child subnet's checkpoint data or a pointer to it.
  * Checkpoint validity rules (and logic enforcing them). E.g., "Checkpoints must be at least every Δ subnet blocks apart", or "A checkpoint must contain a reference to (hash of) a previous checkpoint.", etc.

2. **IPC coordinator/gateway actor (*GWA*):** a smart contract that exists in every subnet in the IPC hierarchy and contains all information and logic the subnet itself needs to hold in order to be part of IPC. The state of the GWA includes:
   - *Membership data* defining the set of replicas running the GWA's subnet. This may include the identities of the replicas, their network addresses, weights of their votes in the executed consensus protocol, and other subnet-specific membership information.
   - *Parent state finality verification.* Analogously to the SA's child state finality verification logic, this is the logic to verify that a state/tx is final in the parent subnet, using a PoF submitted as transaction(s) to the child subnet by the IPC agent(s).
   - *Inter-subnet transactions service* (denoted POST-OFFICE). The GWA contains a registry of subnets and a functionality that can be used to transfer data from one subnet to another.

[4]Finality is an elusive concept that we do not take upon ourselves to define here. For simplicity, we assume finality in a Boolean manner, either tx is final or it is not. This could easily be generalized to parameterized finality of the sort "the probability of tx persisting is at least x."

The POST-OFFICE specifies the methods and the state locations that are used for these services. This functionality is required for the communication of two smart contracts across subnets[5]

3.2 Interfaces

We now describe the interfaces of the Gateway Actor (*GWA*) and the Subnet Actor (*SA*) smart contracts (by listing the methods that can be invoked through transactions submitted to their respective subnets), and the IPC Agent process (by listing the events it reacts to). The other two processes, the parent and child replica, interact with the IPC Agent by the IPC Agent observing the relevant parts of their respective SMR systems' replicated state. The parent and child replicas do not perform any IPC-specific tasks themselves, except for providing the data necessary for a IPC Agents to construct proofs of finality.

Note the asymmetry in the of *Joined/Leave* within the *GWA* and *Join/Left* within the *SA* (and their analogues), but the symmetry across the two smart contracts. This is because the intended invocation pattern for several functionalities is as follows (to be detailed in Section 4).

1. The initiating subnet invokes a function on a smart contract (e.g., $SA.Join$).
2. Subnet Actor notices the invocation, constructs the required PoF, and submits a transaction to the other subnet (e.g., $GWA.Joined$)

Notation. To denote a Function a of Smart Contract in the replicated state of a Subnet, we write *Subnet.SmartContract.Function*. E.g., the *GWA*'s function *CreateChild* in subnet P is denoted *P.GWA.CreateChild*. We also use this notation for a transaction tx submitted to subnet P that invokes the function, e.g., $tx = P.GWA.CreateChild(P/C, params)$.

[5]When inter-subnet data transfer happens between users (Externally Owned Accounts — *EOA*— in Ethereum's jargon), they can actively participate in the propagation by submitting transactions to the parent and child subnets. Smart contracts, on the other hand, do not have that power and, therefore, cannot communicate inter-subnets as efficiently as users (*EOA*).

3.2.1 Gateway Actor (*GWA*)

Algorithm 3: *GWA* interface

```
▶ GWA:
    CreateChild(name, params)
        Creates a new SA with the given name and subnet-specific parameters
          (such as initial membership, etc.). The subnet governed by the created
          SA will be considered the child of the subnet of this GWA.
    KillChild(name)
        [mp: Is it meaningful to have this functionality at all?]
    Joined(identity, metadata, PoF)
        For subnets with explicit membership defined by the SA in their parent,
          if PoF is valid, updates the membership data of the GWA to include
          the replica with identity and the associated metadata. A valid PoF
          means that SA.Join(identity, ..., ...) has been successfully invoked in
          the parent's replicated state.
    Leave(identity)
        For subnets with explicit membership defined by the SA in their parent,
          removes the replica with identity from this subnet's membership.
    Deposited(amt, dest, PoF)
        If PoF is valid, adds amt newly minted coins to account dest. A valid
          PoF means that a corresponding SA.Deposit(..., amt, dest) has been
          successfully invoked in the parent's replicated state.
    Withdraw(src, amt, dest)
        Burns amt coins from account src to be returned to the dest account in
          the parent subnet.
    Propagate(tx)
        Adds the cross-net transaction tx to the list of transactions to be
          submitted to the the another subnet. (The IPC agents observing the state
          of this subnet will pick it up from here and do the actual submission.)
```

3.2.2 Subnet Actor (*SA*)

Algorithm 4: *SA* interface (governing subnet C)

▶ *SA*:

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

3.2.3 IPC Agent

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

Algorithm 5: IPC Agent interface

▶ *IPC Agent*:

upon *parent.SA.Join(identity, src, collateral)* **do**

Constructs a PoF proving that the invocation of *parent.SA.join(identity, src, collateral)* has been finalized in the parent's replicated state, as well as subnet-specific metadata based on the *identity* of the joining replica and the associated *collateral* and notifies the child subnet by submitting a transaction that invokes *child.GW.Joied(identity, metadata, PoF)*.

upon *child.GW.Leave(identity)* **do**

Constructs a PoF proving that the invocation of *child.GW.Leave(identity)* has been finalized in the child's replicated state and notifies the parent subnet by submitting a transaction that invokes *parent.SA.Left(identity, PoF)*

upon *parent.SA.Deposit(..., amt, dest)* **do**

Constructs a PoF proving that the invocation of *parent.SA.Deposit(src, amt, dest)* has been finalized in the parent's replicated state and notifies the child subnet by submitting a transaction that invokes *child.GW.Deposited(amt, dest, PoF)*

upon *child.GW.Withdraw(..., amt, dest)* **do**

Constructs a PoF proving that the invocation of *child.GW.Withdraw(..., amt, dest)* has been finalized in the child's replicated state and notifies the parent subnet by submitting a transaction that invokes *parent.SA.Withdrawn(amt, dest, PoF)*

upon *child.GW.CrossNetTX(tx, destName)* **do**

If *destName* points up the hierarchy, submits *tx* to the parent subnet.

upon *parent.GW.CrossNetTX(tx, destName)* **do**

If *destName* points down the hierarchy, submits *tx* to the child subnet.

4 IPC functionality

IPC exposes the following functionalities:

- Creating child subnets.
- Removing child subnets.
- Depositing coins from an account in a subnet to an account in its child.
- Withdrawing coins from an account in a subnet to an account in its parent.
- Checkpointing - including a checkpoint of a subnet's replicated state in the replicated state of its parent.
- Propagating cross-net-transactions - invoking smart contracts in a subnet through changes in the replicated state of another subnet.

In the following, we describe each functionality in detail.

4.1 Creating a child subnet

Any user of a subnet P can create a new subnet P/C by submitting a transaction $P.GWA.CreateChild(P/C, params)$. This results in the creation of a new subnet actor SA_C in P governing the subnet P/C. The *params* value describes all the subnet-specific parameters required to initialize the state of $P.SA_C$, such as the initial membership data, the consensus protocol to use, etc.

4.2 Killing a child subnet

A child subnet P/C can be removed from its parent P through a transaction invoking $P.GWA.KillChild(P/C)$. [**mp:** We will later define a mechanism to determine who has the right to do this and when.]

4.3 Deposits

A deposit is a transfer of funds (of some amount *amt*) from an account *src* in the parent subnet P to an account *dest* in the child subnet P/C. We assume that the owner of *src* is either running their own IPC Agent to perform the necessary operations described below, or uses another trusted IPC agent to act on their behalf. The deposit is performed as follows:

1. The owner of *src* submits a transaction $tx = P.SA.Deposit(src, amt, dest)$.

2. The parent SMR system orders and executes the *Deposit* transaction (provided *src* has enough funds) by transferring *amt* from *src* to the *SA* (concretely, to *dest* account representation within the *SA*). This effectively locks the funds within the *SA* smart contract, until the *SA* smart contract transfers it back to *src* during a withdrawal (see Section 4.4).

3. When the parent's replicated state that includes *tx* becomes final (for some SMR-system-specific definition of finality), The IPC agent constructs a $PoF(tx)$[6]

4. The IPC Agent submits a transaction $tx' = P/C.Deposited(amt, dest, PoF)$ to the child SMR system.

5. Upon ordering tx', the replicated logic of the child SMR system mints *amt* new coins and adds them to *dest*.

[6] The exact content of *PoF* for the transaction *tx* depends on the implementations of the SMR systems. It might contain, for example, a quorum of replica signatures, a Merkle proof of inclusion, or even be empty.

The events being produced and consumed by the deposit functionality and in Algorithm 6 the pseudocode per component to implement the functionality.

Algorithm 6: Deposit operation

```
▶ Owner of src:
    submit tx = P.SA.Deposit(src, amt, dest) to parent subnet
▶ P.SA.Deposit(src, amt, dest):
    move amt from src to P.SA.accounts.dest          // "lock" at parent
▶ IPC agent:
    upon tx = P.SA.Deposit final at parent do
        create PoF that tx is final at parent SMR    // see section 6 for
          details
        submit P/C.GWA.Deposited(amt, dest, PoF)
▶ P/C.GWA.Deposited(amt, dest, PoF):
    verify(PoF)
    increase dest account by amt
```

4.4 Withdrawals

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

1. The owner of *src* submits a transaction $tx = P/C.GWA.Withdraw(src, amt, dest)$.
2. The child subnet orders and executes the *Withdraw* transaction, burning *amt* funds in *src* (provided *src* has enough funds).
3. When the child's replicated state that includes the transaction becomes final (for some SMR-system-specific definition of finality that has been defined in the SA), the IPC agent constructs a corresponding PoF and submits a transaction *tx' = P.SA.Withdrawn(amt, dest, PoF)* to the parent subnet. .
4. Upon ordering tx', *P.SA.Withdrawn(amt, dest, PoF)* verifies the PoF and transfers *amt* from *SA* (concretely, to *src* account representation within the *SA*) to *dest* within the parent subnet.

Algorithm 7: Withdraw operation

```
▶ owner of src:
    submit tx = P/C.GWA.Withdraw(src, amt, dest)
▶ P/C.GWA.Withdraw(src, amt, dest):
    upon tx = Withdraw(src, amt, dest) do
        deduct amt from src // "burns" amt in child
▶ IPC agent:
    upon tx = P/C.GWA.Withdraw(src, amt, dest) final at child do
        create PoF(tx)                      // see section 6 for details
        submit P.SA.Withdrawn(amt, dest, PoF)
▶ P.SA.Withdrawn(amt, dest, PoF):
    verify(PoF(tx'))
    move amt coins from P.SA to dest // "unlocks" amt for dest
```

4.5 Checkpointing

A checkpoint contains a representation of the state of the child subnet to be included in the parent subnet's replicated state. A checkpoint can be triggered by predefined events (i.e. periodically after a number of state updates, triggered by a specific user or set of users, etc.). A checkpoint is created as follows:

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

2. [**TODO:** Here the IPC Agent should decide (based on some possible reward) whether to submit the Checkpoint transaction.]

3. The IPC agent submits a transaction tx = *P.SA.Checkpoint(chkp, PoF)*.

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

[**TODO:** Pseudocode]

The above pseudo code is intentionally abstract, with a number of implementation decisions not specified, such as the main function for creating and verifying a PoF, events that trigger the creation of a new checkpoint, the compression procedure with respect to the previous checkpoint, and the *ShouldSubmitCheckpoint* function to decide whether the participant submits or not a checkpoint.

4.6 Slashing

[**gg:** This section is immature for review (even a preliminary one)]
We show here the events produced and consumed by the slashing functionality. Given specific misbehavior from participants that is identified as Proofs of Fraud (PoFs), e.g. gathering signed equivocating messages, the child SMR reports the PoFs to the IPC agent, which immediately forwards a slash a request to the parent SMR. [**arp:** Extend with need to verify if child SMR can continue, needs to remedy its depleted collateral or should be killed with latest checkpoint/state update].

Algorithm 8: Slash Functionality

```
input: -
▶ Child SMR:
    upon Proofs of fraud pofs generated do
        Notify ⟨report, pofs⟩ to IPC agent
▶ IPC agent:
    upon ⟨report, pofs⟩ notified by child SMR do
        Submit ⟨slash, pofs⟩ to parent SMR
    upon [arp: State updated after slashing] do
        [arp: Check child SMR rules are still satisfied, remedy/close otherwise?]
▶ Parent SMR:
    upon ⟨slash, pofs⟩ submitted by IPC agent do
        Update SA state slashing/excluding participants Notify SA update to
          IPC agent
```

4.7 Propagating cross-net transactions

[**mp:** This section is in a preliminary stage, take with a grain of salt.]
The implementation of the Gateway Actor's *Propagate* function is sketched in Algorithm 9

Algorithm 9: Cross-net transaction propagation functionality

► *GWA.Propagate(tx* $= (data, src, dest, PoF)$*)*:
  verify(tx.PoF) **case** *dest in current subnet* **do**
    propagateHERE(tx)
  case *dest requires going up the tree* **do**
    propagateUP(tx)
  case *dest requires going down the tree* **do**
    propagateDOWN(tx)
► *PropagateUP(tx)*:
  Add transaction *tx' = parent.GWA.Propagate((data, src.(this_subnets_name), dest))* to the list of cross-net transactions (to be noticed and submitted by the IPC agent)
// *propagateDOWN*(*tx*) is analogous to *propagateUP*(*tx*)
// *propagateHERE*(*tx*) is trivial
► *IPC agent*:
  upon *new transaction tx' in the list of cross-net transactions* **do**
    Create PoF proving that *tx*' has indeed been added to the list fo cross-net transactions in the subnet
    submit *tx*', augmented by *PoF*

5 An Instance of IPC

Here we describe the particular choices implemented by the Consensus Lab team for the reference implementation of IPC. The current implementation considers Filecoin as the root subnet, and Trantor as child subnets. For our interest 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.

The two main conceptual choices in our implementation are: (i) batching upward transactions – both from the same type as well as batching different kinds of transactions together (which includes withdrawals, checkpointing and POST-OFFICE transactions); (ii) using multisigs for verifying finality of a child state, and local finality check for the finality of a parent state.

Batching through the *GWA*. All upward transactions are made via the *GWA* of the child. The transactions are batched there until it is time to checkpoint at the parent (specifically, every Δ child blocks). This way the *GWA* also serves as the single data structure (queue) storing the upward transactions, therefore, the agent only checks this single place to get all necessary info from the child subnet. When the checkpoint batch is executed at the parent, it is done atomically. Since POST-OFFICE transactions are included in the batch, the atomic execution of the batch also depends on the POST-OFFICE functionality at the parent handling those messages which lays in the *GWA*. Therefore, *SA* first commits the cid of the checkpoint batch without executing the included transactions, and then triggers the *GWA* to examine the batch as well. The *GWA* checks whether there are sufficient funds to handle the POST-OFFICE transactions in the batch. If the funds are insufficient, the entire checkpoint batch fails. If the *GWA* approves the batch, then it is executed — both at *SA* and at *GWA*.[7]

Checking for Finality. Recall that we use Trantor as the consensus engine for child subnets. Therefore, a transaction (or state) *tx* is accepted as final by providing a *PoF* containing 2/3 of the child's validators signatures[8] of *tx*. In other words, given that *tx* is a cid, the parent's subnet call to $SA.verifyGlobalFinality(tx,PoF)$ returns True iff *PoF* contains signatures (on *tx*) of validators with more than 2/3 of the voting rights in the child. The voting rights are measured according to what is written in *SA* for the epoch containing *tx*. [**arp:** I think for M2 we do not actually even have signatures from validators but instead $2n/3$ validators independently submitting the batched checkpoints. The plan is to improve this for M3 I think. REs to verify.][**gg:** You're right, but it's too embarrassing to document :-) (and should change quickly)]

For considering a transaction *tx* at the parent as final, we use the fact that a participant has view into a version of the parent subnet (through its local parent replica process). In this case, the *PoF* contains the block height *h* (and pointer to that block) at the parent subnet. A child replica then asserts with its parent that the state is final by checking with its local version of the parent blockchain at height *h*. If the the local version at the parent replica did not reach height *h* yet, the child replica considers the state to currently be non-final/non-valid. The child checks again when the parent reaches height *h*. [**TODO:** Verify with REs]

Compressed accounting. In section 3 we mentioned that *SA.accounts* can include fine grained accounting data. However, in the current implementation *SA.accounts* contain a single variable representing the sum of all the individually locked funds at the parent which are dedicated to the child subnet (we call this value *circulating supply*). This has the obvious benefit of reducing the space complexity of *SA*.

[7] Batching the POST-OFFICE transactions together with the checkpoint and withdrawals, as well as entangling the *GWA* functionality (POST-OFFICE transactions) with the *SA* functionality (withdrawals and checkpoint transactions) in a single atomic execution are design choices. This choice benefited the development velocity by having a single mechanism to handle everything and by transferring the responsibility of re-transmitting failed POST-OFFICE transactions to the validators rather than it being the responsibility of *SA* (which requires logic implementation).[**gg:** Alfonso, please explain...]

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

Incentives? Were not the most urgent topic so currently neglected. TBD.

The pseudocode of algorithm 10 demonstrates the main design choices made by the reference implementation.

5.1 IPC actors

- GWA:
  - [**TODO:** Do leaf nodes have a *GWA* in the IPC reference implementation?]
  - GW stores checkpoints of all immediate children (and not each SA).[**gg:** I'm not sure this is completely true.] ×
  - GW batches checkpoints messages with other crossnet messages (withdrawals, Postbox, etc.) ✓
  - Parent Finality verification implicit immediate as soon as transaction locally seen as ordered at parent SMR ✓[**TODO:** Check in code how this is done]
- SA:
  - Finality verification also implicit (hardcoded, 2n/3 submissions from child's validators, after 2n/3-th ordered at parent, finalized). This will be improved with threshold signatures (off-chain signing by child's validators and one submission, immediate finalization)
  - SA's Accounting data is the circulating supply ✓
  - No governance account, but **participation incentives** by:
    1. Additional IPC fee for validators to relay
    2. Incentivizing checkpoint submission as a result of batching with other crossnet messages (which contain the IPC fees), further justifying checkpoints being stored at GW.
    3. Keeping Stake at SA and slashing through fraud proofs
- batching: checkpoints + withdrawals
- *GWA* as the place for notifying the agent of upward messages

5.2 Functionality

5.2.1 Checkpoints

The IPC reference implementation makes certain implementation decisions that affect the design of checkpoints, particularly:

1. The *GWA* stores the checkpoints from all subnets, instead of having each subnet store their respective checkpoints at their specific *SA*.

2. Checkpoints are triggered periodically after Δ blocks are decided at the child

3. Crossnet messages are batched together with checkpoints and forwarded to the parent following checkpoint triggers.

We show in Algorithm 10 the pseudocode of checkpoints as implemented by the IPC reference implementation.

Algorithm 10: Checkpoints IPC reference implementation

```
▶ child SMR replica:
    upon new block b decided do
        notify newBlock(block.height) to IPC agent
▶ IPC agent:
    upon notification of newBlock(blockheight) from child SMR do
        if blockheight mod Δ = 0 then
            SA_state ← query parent for SA's state
            if Self in SA_state.validators then
                state ← query child for state
                gcChkps ← query child's GWA for grandchildren's checkpoints
                [TODO: What is gcChkps used for? gcChkps.crossnetmsgs?
                  ask REs]
                postboxmsgs ← query child's GWA for postbox messages
                pChkps ← query parent's GWA for previous checkpoints
                cState ← compressState(state,pChkps.latestChkp, gcChkps,
                  postboxmsgs)
                create PoF that cState is final at child        // generate
                  certificate containing 2/3 of the child validators'
                  signatures
                [TODO: For M2 this is each node submitting to parent?, check
                  with REs]
                tx = SA.Checkpoint (cState, PoF)
                submit tx to parent SMR replica
                [TODO: Is it like this? is there a refund?]
▶ parent SMR replica:
    upon tx = SA.Checkpoint (cState, PoF) do
        assert SA.verifyGlobalFinality(cState,PoF)     // Verify certificate
        SA.latestCheckpoint.update(cState)
```

[**TODO:** Discuss/verify] These implementation decisions come with a number of advantages. First, parent and child subnets periodically synchronize following predictable events. Second, having the *GWA* store all checkpoints easily enables batching of checkpoint messages with crossnet messages. Third, this batching inherently provides participation incentives for validators to submit checkpoints, as they will be rewarded with the IPC fee from crossnet messages.[**gg:** I would say that the third point is a problem not an advantage. It entangles checkpointing with crossnet messages and might make checkpoints depend on the cross-net messages for incentivation.] [**arp:** Agreed, but iiuc we will not have other incentives by M2/M3 for checkpoints, so some incentives are better than none at all for checkpoints. But yes, down the road we should work on independent checkpoint incentives (governance account).]

6 Verifying the Finality of *tx*

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

References

[1] 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]