Interplanetary Consensus (IPC)

Consensus Lab

Abstract

TODO

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 global notion of money (assets / tokens / ...) and accounts that can hold them, IPC also defines how money is moved

between accounts in different subnets.[**gg:** Globality is unnecessary. We rely on a parent–child common currency. One may create a new token inside a child and use this new token in the grandchild while making all child–grandchild monetary interactions based on that token.]

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). [**mp:** Go in more detail already here and mention PoS, collateral, slashing, firewall property, etc...?]
- "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.
- [**mp:** Explain each item better. Any more items?]

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 the IPC Glossary, currently available at https://docs.google.com/document/d/15pA7ahjeA-HYOl8Pxj0n6PxEswYlRVrZ112MJuRR0fY/edit?usp=sharing

[**mp:** When the Glossary becomes stable, we can maybe add it as an appendix to this document.]

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.

Algorithm 1: Process definition.

```
variable = initial value
variable = initial value
...
▶ process:
    upon event(params...) do
        // Logic to execute atomically
    upon message(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 trust each other, i.e., can 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

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

IPC. We thus state those assumptions whenever necessary, when describing concrete implementations of IPC components.

2.2 State machine replication and smart contracts

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

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.

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.

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

Algorithm 2: smart contract definition

```
variable = initial value
variable = initial value
...
▶ smart contract name:
    upon event(params...) do
        // Logic to execute
        trigger event(params...)
    upon tx(params...) do
        // Logic to execute
        trigger event(params...)
    ...
```

Note that, despite using similar syntax to describe processes and smart contracts, those are fundamentally different. The former usually represent OS-level processes running on some physical machine, the latter 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.

In this document, we mostly focus on the interaction between a single parent SMR system (consisting of parent processes) and a single child SMR system (consisting of child processes), as this is the most important building block of IPC. In the next section, we describe the components involved in this interaction and their interfaces.

2.3 Money

We assume that all smart contracts have a notion of *money*, that they can transfer among each other within one SMR system. Each end user of the SMR system is assumed to have a personal wallet and a corresponding account in some subnet

3 Components and their Interfaces

We separate the software needed to run IPC into three processes and two smart contracts:

[**mp:** TODO: Express those components and their interfaces also in pseudocode.]

1. **IPC agent:** The software that is in charge of the interactions between the two blockchains. This includes, for example, observers for the parent and child subnets. (Note that it is a process and not a smart contract). The IPC agent is a piece of software that mediates the interactions between the child and parent SMR software modules.

2. **Parent SMR replica:** The software that runs the parent blockchain. Note that this module also entails the interaction with the IPC smart

contract *SA*, which is maintained at the parent subnet. Any update that the parent process performs on *SA* is notified to the IPC agent.

3. **Child SMR replica:** The software that runs the child blockchain. Note that some of the rules the child blockchain must satisfy are listed in *SA*. Any output operation (withdraw, checkpoint) is notified to the parent process through the IPC agent.

1. **IPC subnet actor (*SA*):** The smart contract implementation that is running on the parent blockchain. It is invoked only through transactions that are included in the parent blockchain.

2. **IPC coordinator/gateway actor (*GWA*):** a smart contract that exists in every subnet in the IPC hierarchy and contains methods facilitating inter-subnet operations.

We now define minimal interfaces between the different modules that enable the correct operation of an Interplanetary Consensus system. A guiding principle in the interface design is to minimize changes to the SMR codebase; therefore, most extra logic of the IPC will be added into the IPC agent and the smart contracts *SA* and *GWA*. Doing so should facilitate the deployment of IPC with new SMR protocols by not requiring a developer familiar with IPC to be an expert on SMR: some understanding is still needed to optimize the agent's implementation, but the SMR code would remain portable.

We require four interfaces: (i) IPC agent — parent SMR, (ii) IPC agent — child SMR, (iii) parent SMR— *SA*, and (iv) any SMR— *GWA*. Both (i) and (ii) can comprise of only:

1. Agent submits a transaction *tx* to the SMR process.[3]

2. Agent queries the state of the SMR process. The SMR process returns its current state (possibly limited to only a requested part of the state).

3. SMR process notifies the agent on events of interest (e.g., changes to the state of *SA*).

The interface between an SMR and *SA* or *GWA* is based on the execution engine of that SMR and the functionality desired by *SA*. The specifics of the execution engine's system calls depend on implementation. Whenever such a call is not clear from context we provide a description of what it entails.

The state of *SA* includes representations of:

[3]As part of the notification defined below, it could be that after submitting *tx*, until the SMR process returns *complete* (perhaps with a finality parameter) or *declined*, *tx* is considered *pending*.

- Accounting data. This can vary from a single account representing all the parent's coins in the child to an account balance for each user in the child subnet (custodial vs non-custodial accounting). We continue with the non-custodial approach as the other can be viewed as a specific limitation of it.

- Governance account (denoted *gov-acc*). This account facilitates the economic design of a subnet. It can be used for governing operations of the subnet. For example, collecting fees and making payments (to validators, for checkpoint reimbursement etc.)

- Consensus information. The data (or a pointer to it) that is needed to run the ordering of the subnet.
  - Consensus protocol.
  - Subnet configuration such as the validator set, voting rights, collateral deposits, etc.
  - Payments methods for participation. E.g., transaction fee mechanism, block rewards.

- Finality verification. A method to Verify that a state/tx is final[4] in the child subnet. For this, we will use the function $SA.verifyGlobalFinality(tx,PoF)$ which excepts as arguments a transaction (or state) and a PoF, and outputs True if tx is considered globally final in the child subnet and False otherwise. This function must only depend on its inputs and the internal state of SA. For example, PoF is a threshold signature that can be verified against the set of validators in SA.

The above suffice for an Interplanetary Consensus system with a minimal inter-subnet functionality of users' asset-transfer, and a general SMR per subnet. We continue with the additional state required for enhanced functionalities.

- Slashing functionality.
  - 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).
  - Incentives design, i.e., specified penalties for misbehavior and rewards for reporting.

- Checkpointing rules.

[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."

  - When checkpoints are valid. E.g., every Δ subnet-blocks from the previous checkpoint, or the checkpoint's L_2 distance from the previous is larger than L.
  - Fee payments for checkpoints.

Recall that SA lives at the parent SMR. However, some of the objects that are represented in SA are modified in the child subnet (e.g., accounting data). Therefore, such objects are likely to have a representation in the child SMR as well. Moreover, the representation in the child SMR may differ from those in SA. This is due to SA being less frequently updated (it is part of the parent SMR state). The representations are periodically synchronized, e.g., at a checkpoint event. Figure 1 illustrates the components and their interfaces.

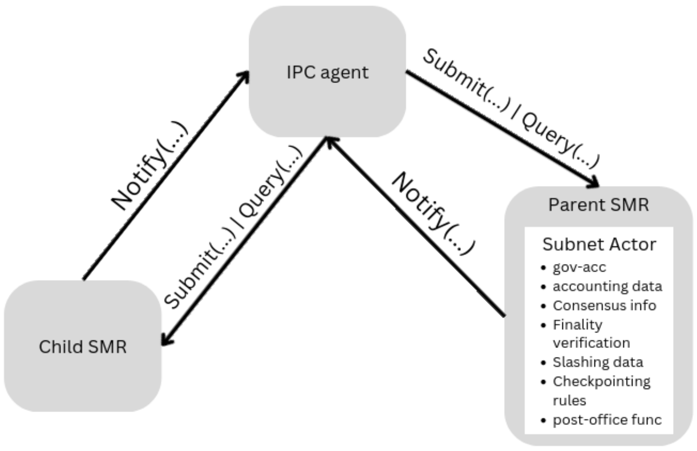

Figure 1: The basic components and their interfaces.[**gg:** ALEX, please add a fig for GWA and move the post-office from the SA to GWA.]

The state of GWA includes:

- Parent's finality verification. A method to verify that a state/tx is final in the parent subnet. For this, we will use the function $GWA.verifyParentFinality(tx, PoF)$ which excepts as arguments a transaction (or state) and a PoF, and outputs True if tx is considered globally final in the parent subnet and False otherwise. This function must only depend on its inputs (and perhaps some internal state of GWA).

- Inter-subnet transactions service (denoted POST-OFFICE). GWA contains a registry of subnets and a functionality that can be used to transfer data from one subnet to another. The POST-OFFICE specifies the methods and the state locations that are used for these services.

In particular, consider the following case involving a smart contract.[5] Smart contract *SmCt* emits an event *e* that contains *data* which is desired to reach the destination *dest* in a different subnet.

4 IPC functionality

We list in this section the functionality that should be provided by the IPC components. We first list the minimal functionality required for every subnet (deposits and withdrawals), to then extend it with enhanced functionalities. We model components as processes that produce and consume events. Events consumed by the IPC agent are the result of either a notification from one of the SMRs or the response of a query made by the IPC agent. Events produced by the IPC agent result in the IPC agent submitting a transaction that will change the state of the SMR that consumes the event.

We note that our focus is on the core functionalities, disregarding optimizations for the moment. Batching is a prime example of this. It is expected that batching will be a key optimization whenever *verifyGlobalFinality*(*tx*,*PoF*) is used, as calling *verifyGlobalFinality*(*tx*,*PoF*) can be costly. Batching allows us to perform multiple operations for one *verifyGlobalFinality*(*tx*,*PoF*) call, reducing its overall cost.

4.1 Minimal Functionality

We show in this section the functionalities for deposits and withdrawals.

4.1.1 Deposits

[**arp:** Consider need to pause/remedy subnet after deposit (e.g. collateral not enough with new supply). IPC agent should check in that case]

A deposit is a transfer of funds (of some amount *amt*) from user u_P's wallet in the parent subnet to user u_C's wallet in the child subnet. We assume that u_P is a participant running a parent replica, a child replica, and an IPC agent.[6] The deposit is performed by the user controlling the IPC agent as follows:

[5]When inter-subnet data transfer happens between users (Externally Owned Accounts — *EOA*— in Ethereum's jargon), they can actively participate in the propagation via the IPC agent that communicates with both 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*).

[6]If u_P does not run these processes, u_P contacts a trusted participant that does and that performs the deposit on u_P's behalf.

1. The local IPC agent submits to the parent SMR replica the corresponding (properly signed) transaction $tx = Deposit(src, amt, SA.accounts.dest)$ with $src = u_P$ and $dest = u_C$.

2. The parent SMR system orders and executes the Deposit transaction (provided u_P has enough funds) by transferring amt from u_P's parent account to the SA (concretely, to u_P's account representation within the SA). This effectively locks the funds within the SA smart contract, until the SA smart contract transfers it back to u_P's account during withdrawal (see Section 4.1.2).

3. When the parent's replicated state that includes the transaction becomes final (for some SMR-system-specific definition of finality), the local parent replica notifies the local IPC agent, potentially attaching a proof of finality of $PoF(tx)$ to the notification.[7]

4. The IPC agent constructs a transaction $tx' = Deposited(\langle src, amt, SA.accounts.dest\rangle, PoF)$ and submits it 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 u_C's account.

We show in Figure 2 the events being produced and consumed by the deposit functionality and in Algorithm 3 the pseudocode per component to implement the functionality.

Algorithm 3: Deposit operation

```
input: $src$ account in parent, $dest$ account in child, amount $amt$
▶ IPC agent:
    submit $tx = Deposit(src, amt, SA.accounts.dest)$ to parent SMR replica
▶ Parent SMR replica:
    upon $tx$ do
        move $amt$ from $src$ to $SA.accounts.dest$          // "lock" at parent
        notify agent ParentDeposited($tx$)
▶ IPC agent:
    upon notification of ParentDeposited(tx) from parent SMR do
        create $PoF$ that $tx$ is final at parent SMR         // see Sec. ?  for
          details
        submit Deposited= $\langle tx, PoF\rangle$ to child SMR
▶ Child SMR replica:
    upon Deposited do
        assert $PoF$ for $tx$
        increase $dest$ account by $amt$
```

[7]The exact content of $PoF(tx)$ 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.

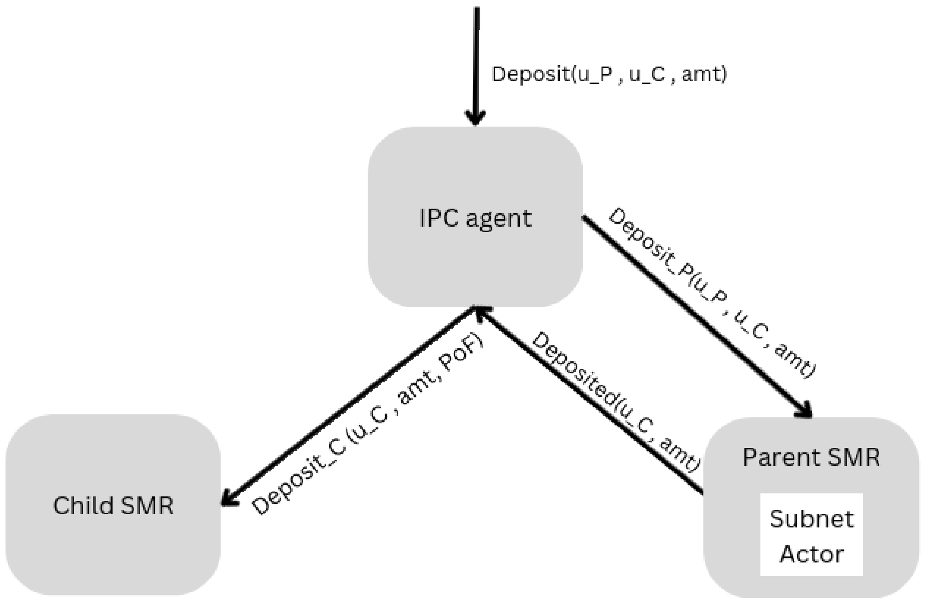

Figure 2: Events produced and consumed during a deposit.

One thing that differs a downward transaction (e.g., deposit) from an upward transaction (e.g., checkpoint) is that any participant that operates the child SMR replica also has visibility into the state of the parent SMR (albeit stale) through its local parent SMR replica. This enables the **local validity check** method to assert the finality at the parent (which may or may not be preferred over others).[8]

4.1.2 Withdrawals

A withdrawal is a transfer of funds from user u_C's wallet in the child subnet to some user u_P's wallet in the parent subnet. We assume that u_C is a participant running a parent replica, a child replica and an IPC agent. The withdraw is performed as follows:

1. u_C triggers the $Withdraw(u_C, u_P, amt)$ event at the local IPC agent.

[8]**local validity check** (simpler, efficient, *weaker guarantees*): PoF contains a pointer to the block containing tx at the parent, together with the height h of that block. To assert that tx is final, the child queries the parent about TX, if it exists – return valid, else – return invalid. If invalid but the parent is still below height h, then query again when parent reaches height h. This is a test inside the child SMR process. Therefore, if we want this method (and I believe we do), we should widen the interface so that a child SMR can ask the agent to get data from the parent. However, this optimization comes at the expense of the encapsulation of components, that is, it entails tinkering with the child SMR code.

2. The local IPC agent submits the corresponding (properly signed) transaction $tx = Withdraw_C(u_C, u_P, amt)$ to the child SMR system.
3. The child SMR system orders and executes the Withdraw transaction, burning amt funds in u_C's account (provided u_C has enough funds).
4. 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 local child replica notifies the local IPC agent, potentially attaching a proof PoF that this state is final.
5. The IPC agent constructs a transaction $tx' = Burned(u_P, amt, PoF)$ and submits it to the parent SMR system.
6. Upon ordering tx', the replicated logic of the parent SMR system updates the state of the SA transferring the funds from SA (concretely, to u_P's account representation within the SA) to u_P's account.

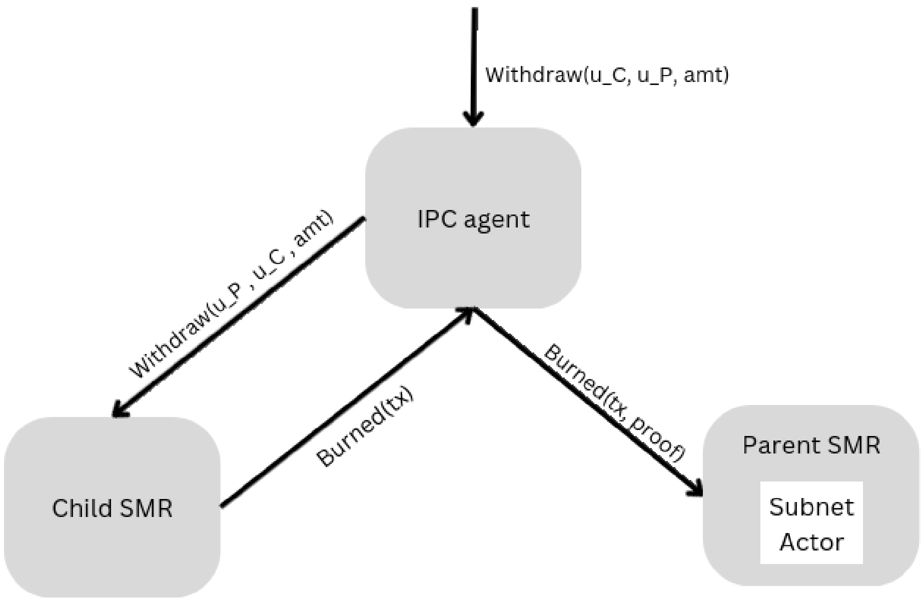

Figure 3: Events produced and consumed during a withdrawal.

Algorithm 4: Withdraw operation

```
input: $src$ account in child, $dest$ account in parent, $amt$ amount of coins
▶ IPC agent:
    submit $tx = Withdraw(src, amt, dest)$ to child SMR
▶ Child SMR replica:
    upon $tx = Withdraw(src, amt, dest)$ do
        deduct $amt$ from $src$ account at child // ‘‘burns" amt in child
        notify agent Burned($tx$)
▶ IPC agent:
    upon notification of Burned(tx) from child SMR replica do
        create $PoF$ that $tx$ is final at child SMR // see Sec. ? for details
        submit $tx' =$ Burned ($tx, PoF$) to parent SMR replica
▶ parent SMR replica:
    upon $tx' =$ Burned ($tx, PoF$) do
        assert $SA.verifyGlobalFinality(PoF, tx)$
        move $amt$ coins from $SA$.accounts[$src$ ] to $dest$
```

4.2 Enhanced Functionality

We show here a list of desirable functionalities that build upon the basic withdrawals and deposits.

4.2.1 Checkpointing

A checkpoint contains a representation of the updated state of the child SMR system to be included in the parent SMR system. 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.). As such, the checkpoint functionality may or may not be triggered by a user request on the child SMR. A checkpoint is performed as follows:

1. If the predefined checkpoint trigger is met, then the IPC agent queries the child SMR replica for the updated state to be represented in this checkpoint.

2. The IPC agent creates a proof *PoF* that this updated state of the child SMR system is final, possibly compressing its representation of the state.

3. The IPC agent submits a transaction $tx' =$ `Checkpoint` $(state, PoF)$ to the parent SMR replica

4. Upon ordering tx', the replicated logic of the parent SMR system updates the state of the SA according to the checkpoint state, if necessary.

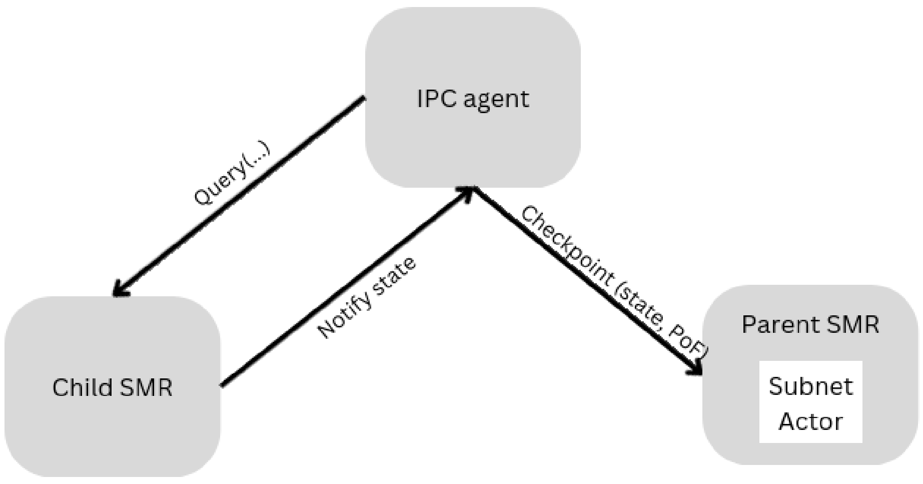

Figure 4: Events produced and consumed by the checkpointing functionality.

Algorithm 5: Checkpoint operation

```
▶ IPC agent:
    if trigger for checkpoint then
        state ← query the child SMR replica for the state
        create PoF that state is final at child    // Possibly compress state
    submit $tx' = \texttt{Checkpoint}(state, PoF)$ to parent SMR replica
▶ parent SMR replica:
    upon $tx' = \texttt{Checkpoint}(state, PoF)$ do
        assert $SA.verifyGlobalFinality(PoF, state)$
        $SA.latestCheckpoint \leftarrow state$
```

4.2.2 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].

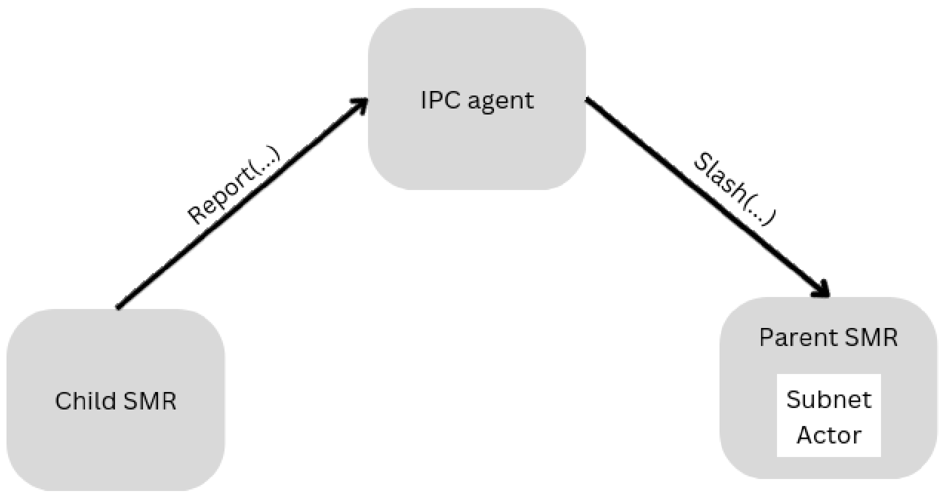

Figure 5: Events produced and consumed by the slashing functionality.

Algorithm 6: Slash Functionality

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

4.2.3 post-office

The POST-OFFICE functionality is an inter-subnet transaction service. The main motivation for this functionality comes from a "potential clients" request: enable a smart contract in one subnet to interact with a smart contract in a different subnet.

[**gg:** Edge case: a leaf subnet does not have a *SA* and, therefore, no POST-OFFICE. We can consider removing the POST-OFFICE functionality from the *SA* and to deploy it as an independent smart contract that will appear only once per subnet. In this case, it needs permissions to call $SA.verifyGlobalFinality(tx,PoF)$ function.]

Algorithm 7: POST-OFFICE Functionality

```
input: $tx = \langle data, src, dest, PoF \rangle$
▶ $SA$.POST-OFFICE:
    upon POST-OFFICE.propagate(tx) do
        case dest in current subnet do
            POST-OFFICE.$propagateHERE(tx)$
        case dest requires going up the tree do
            POST-OFFICE.$propagateUP(tx)$
        case dest requires going down the tree do
            POST-OFFICE.$propagateDOWN(tx)$
    upon POST-OFFICE.propagateUP(tx) do
        if src not from this subnet then
            assert($SA.verifyGlobalFinality(tx,PoF)$)
        src.append(SA's subnet id)
        emit event POST-OFFICE.UP$\langle data, src, dest \rangle$
    // propagateDOWN(tx) is analogous to propagateUP(tx)
    // propagateHERE(tx) is trivial
▶ parent SMR process:
    upon event POST-OFFICE.UP$\langle data, src, dest \rangle$ do
        $tx \leftarrow \langle data, src, dest \rangle$
        notify agent on POST-OFFICE.UP($tx$)
▶ IPC agent:
    upon notification of propagateUP(tx) from child SMR do
        create $PoF$ that $UP(tx)$ is final at child SMR
        $tx_{new} \leftarrow \langle UP(tx), PoF \rangle$
        submit $SA$.POST-OFFICE.$propagate(tx_{new})$ to parent SMR
```

4.2.4 Atomic Execution

TODO Discuss in Lanzarote?

4.3 Future

5 An Instance of IPC

Here we describe the particular choices implemented by the Consensus Lab team.

6 Verifying the Finality of *tx*

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