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 10 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

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.

```
1 variable = initial value
2 variable = initial value
3 ...
4 ▶ process:
5     upon event(params...) do
          // Logic to execute
6     upon message(params...) do
          // Logic to execute
7     ...
```

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 of the system 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 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

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

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 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.

Algorithm 2: smart contract definition

```
1 variable = initial value
2 variable = initial value
3 ...
4 ▶ smart contract name:
5     upon event(params...) do
          // Logic to execute
6         trigger event(params...)
7     upon tx(params...) do
          // Logic to execute
8         trigger event(params...)
9     ...
```

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 user of the SMR system is assumed to be associated with their personal smart contract called *wallet*.

3 Components and their Interfaces

We separate the software needed to run IPC into three processes and one smart contract:

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

- **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 the *SA* is notified to the IPC agent.
- **Child SMR replica:** The software that runs the child blockchain. Note that some of the rules the child blockchain must satisfy are listed in the *SA*. Any output operation (withdraw, checkpoint) is notified to the parent process through the IPC agent.
- **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 this is not a smart contract (it is not the *SA*). It is a piece of software that runs at a node and mediates the interactions between the child and parent SMR software modules.
- **IPC smart contract / 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.

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 minimise changes to the SMR codebase; therefore, most extra logic of the IPC will be added into the IPC agent. 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 three interfaces: (i) IPC agent — parent SMR, (ii) IPC agent — child SMR, and (iii) parent SMR— *SA*. 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 the parent SMR and *SA* is based on the execution engine of the parent 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:

- 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.

[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*.

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

- Inter-subnet transactions service (denoted POST-OFFICE). SA contains functionality that can be used to transfer data from one subnet to another. 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. The POST-OFFICE specifies the methods and the state locations that are used by this service.

[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."

[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).

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 object 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.

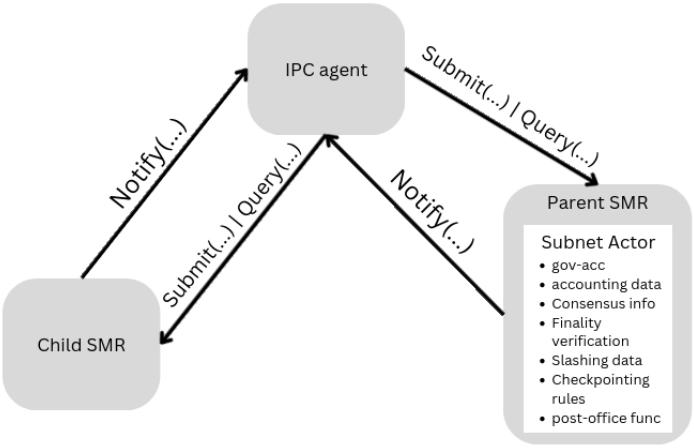

Figure 1: The basic components and their interfaces.

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:

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.

[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.

[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.

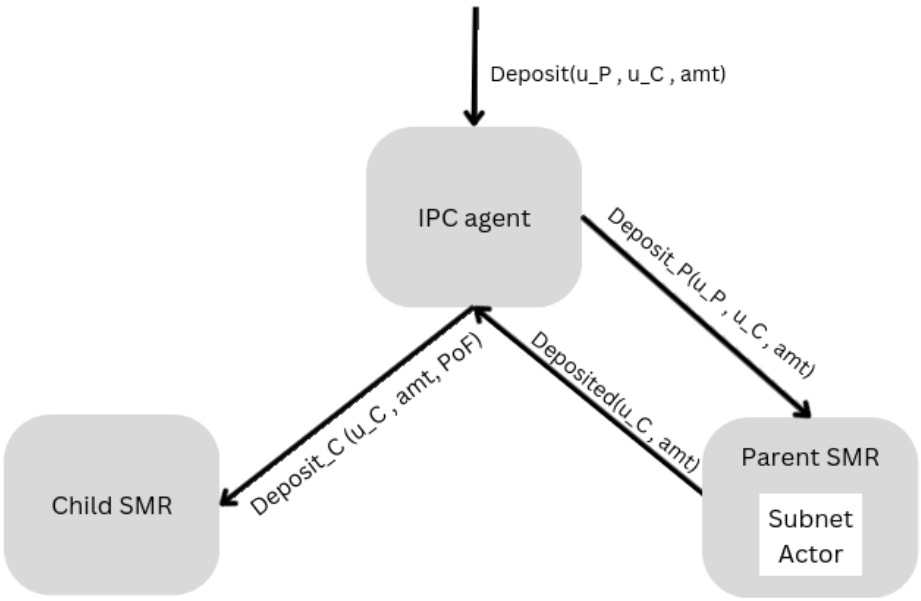

Figure 2: Events produced and consumed during a deposit.

Algorithm 3: Deposit operation

input: *src* account in parent, *dest* account in child, amount *amt*

1 ▶ *IPC agent*:
2 submit $tx = Deposit(src, amt, SA.accounts.dest)$ to parent SMR replica
3 ▶ *Parent SMR replica*:
4 **upon** *tx* **do**
5 move *amt* from *src* to *SA.accounts.dest* `// "lock" at parent`
6 notify agent `ParentDeposited`(*tx*)
7 ▶ *IPC agent*:
8 **upon** *notification of* `ParentDeposited`*(tx) from parent SMR* **do**
9 create *PoF* that *tx* is final at parent SMR `// see Sec. ? for details`
10 submit `Deposited`= $\langle tx, PoF \rangle$ to child SMR
11 ▶ *Child SMR replica*:
12 **upon** `Deposited` **do**
13 assert *PoF* for *tx*
14 increase *dest* account by *amt*

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.
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.

[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.

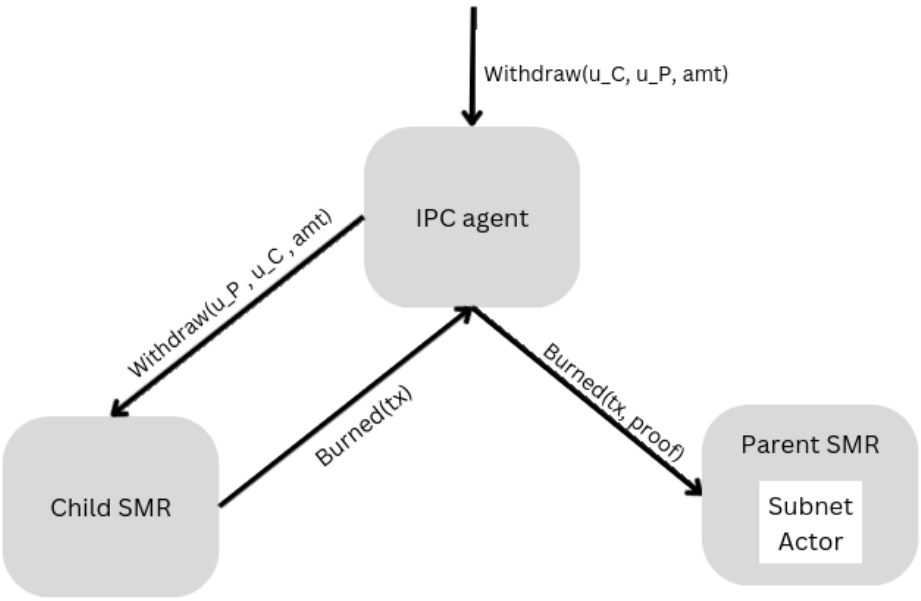

Figure 3: Events produced and consumed during a withdrawal.

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 $tx' = \texttt{Checkpoint}\,(state, PoF)$ 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.

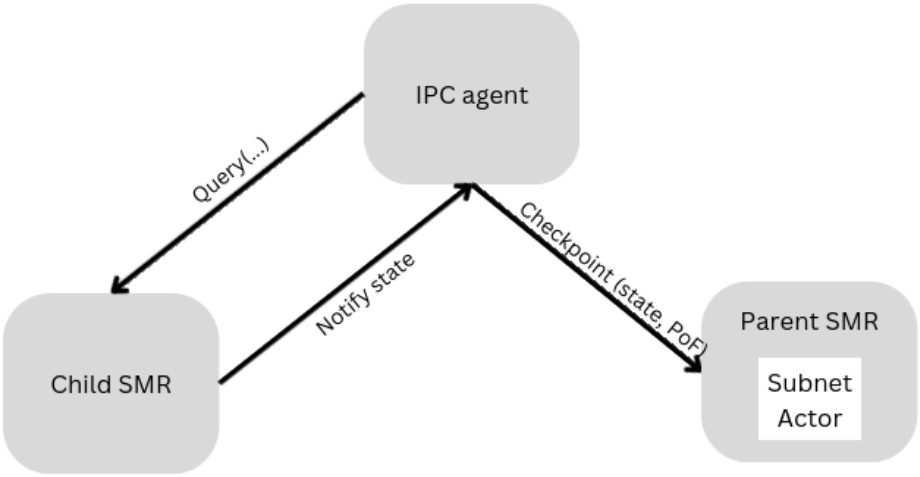

Figure 4: Events produced and consumed by the checkpointing functionality.

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' = Checkpoint(state, PoF) to parent SMR replica
▶ parent SMR replica:
    upon tx' = Checkpoint(state, PoF) do
        assert SA.verifyGlobalFinality(PoF,state)
        SA.latestCheckpoint ← 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].

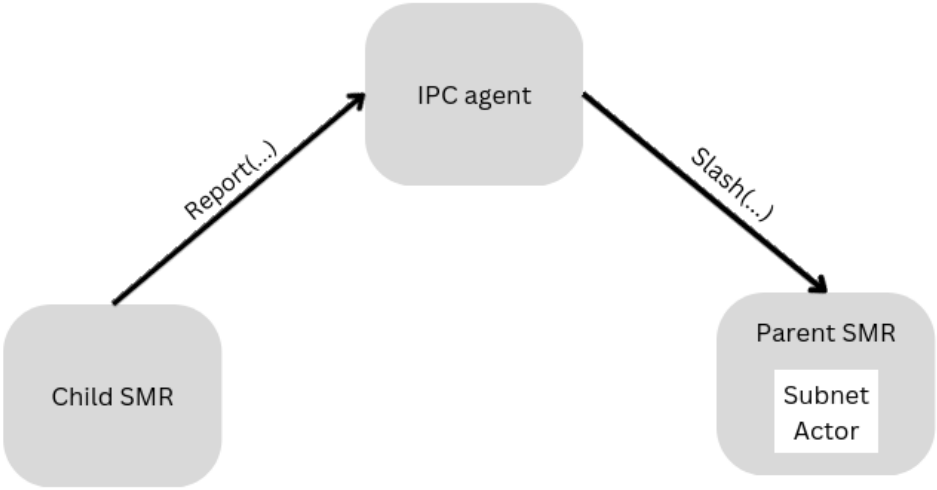

Figure 5: Events produced and consumed by the slashing functionality.

Algorithm 6: Slash Functionality

input: -

▶ *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

4.2.3 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 = ⟨data, src, dest, PoF⟩
 1 ▶ SA.POST-OFFICE:
 2     upon POST-OFFICE.propagate(tx) do
 3         case dest in current subnet do
 4             POST-OFFICE.propagateHERE(tx)
 5         case dest requires going up the tree do
 6             POST-OFFICE.propagateUP(tx)
 7         case dest requires going down the tree do
 8             POST-OFFICE.propagateDOWN(tx)
 9     upon POST-OFFICE.propagateUP(tx) do
10         if src not from this subnet then
11             assert(SA.verifyGlobalFinality(tx,PoF))
12         src.append(SA's subnet id)
13         emit event POST-OFFICE.UP⟨data, src, dest⟩
       // propagateDOWN(tx) is analogous to propagateUP(tx)
       // propagateHERE(tx) is trivial
14 ▶ parent SMR process:
15     upon event POST-OFFICE.UP⟨data, src, dest⟩ do
16         tx ← ⟨data, src, dest⟩
17         notify agent on POST-OFFICE.UP(tx)
18 ▶ IPC agent:
19     upon notification of propagateUP(tx) from child SMR do
20         create PoF that UP(tx) is final at child SMR
21         tx_new ← ⟨UP(tx), PoF⟩
22         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.

7 RAW LEGACY TEXT

Anything below should be ignored by readers

8 TODO

- Collateral
- Schema

9 Introduction

Hierarchical Consensus is still an ongoing project, yet it already has an open source implementation available [**?**]. The best architecture description available is [1].

This document will propose an architecture for the core functionality of HC, on which additional layers/applications can be implemented.

The goal of HC is to improve the scalability of blockchain systems.[9] The chosen approach is by exploiting the "locality" phenomenon of users interactions. In general, if users interact with each other in an arbitrary manner (or even uniformly at random), then there is no "locality" to exploit. However, typically in practical systems, users tend to cluster in a way that users inside a cluster interact with each other more frequently than with outside users. Roughly speaking, we wish to exploit this phenomenon by allowing a cluster to interact internally with less involvement of the main blockchain network. Consequently, the internal interactions become cheaper and faster, thus, reducing the costs for the user. Moreover, the main blockchain also benefits from reduced load, since the demand to record every intra-cluster action is (partially) removed from the main blockchain.[10]

10 Model

- Blockchain = Net; sub-net; parent-net; root-net.
- nodes[**arp:** or validators?] — the main participants in a blockchain system. All security guarantees of a blockchain is based on them. The verifiers of the consensus in the blockchain.
- Users — users are the clients of the system. They represent the "real world" entities that own coins and wish to use the system for interactions. Note: a node in the system is owned by a user (which receives the revenue), but a user does not have to own a node in order to use the system.

[9] This includes throughput increase as well as flexibility in Consensus mechanism choice.

[10] Intra-cluster actions are summarized and recorded periodically to the main blockchain. So these interactions are not completely removed from the main blockchain, but the amount of resources that are demanded from the main blockchain significantly reduces.

- accounts — an account exists in a specific sub-net. A user that wishes to operate in sub-net $\mathcal{SN}$ needs to own an account in $\mathcal{SN}$. The account is controlled by the user. While users are abstract entities that are used to analyse the protocol, accounts are entities that exist in the system, hold balances, interact at the protocol level and create operations for miners to include in the sub-net. For simplicity we assume a user controls at most one account in each sub-net. (This generalizes to multiple accounts trivially.)

11 Skeleton

- Every node in sub-net $\mathcal{SN}$ is also a node in all subnets that are ancestors of $\mathcal{SN}$. This limits the practical depth of the hierarchy, since a "deep" node must divide its finite resources (bandwidth, computation, etc.) among all ancestor subnets. [**gg:** Should still offer great benefits. Should be analyzed theoretically under some assumption, for example: share of "local" interactions.]
  - only consider full nodes; light nodes should be considered independently.
- For each user there exists a hierarchy of accounts — one for each subnet it operates in. The account at $\mathcal{SN}$ contains the amount of tokens the user has at $\mathcal{SN}$ as well as the amount of tokens that are locked for use in the immediate children of $\mathcal{SN}$ (if the user has accounts there).
  - For simplicity, we consider a user always has its root account in the rootnet. In general, it is possible to have the root account at any subnet. However, once the root account is established the user can only open accounts in descendants of that subnet.[11]
- An account interacts directly only with accounts at the same subnet *or* with the account's parent/children. [**gg:** My rational here is to provide the minimal necessary interface. See section 11.3 for details on inter subnet interactions.]
  1. **Intra sub-net interactions.** These interactions could include general operations (e.g., CAS) and are not restricted by HCs design, they should follow the specific subnet definitions.
  2. **Parent–Child interactions.** These interactions are limited to token transfers between the user's parent/child accounts. [**gg:** Since both accounts are controlled by the same user, we do not need to worry about the flow of information between them — it is the same entity and

[11] To add accounts in other subnets, the user will need to create a new root account.

it knows what it's doing! We do need to reflect the state change to the entire system. That is, the new balances in the accounts (after tokens were transferred between the parent and child) must be visible to the respective sub-nets.]

- Interactions between parents and children are done periodically[**arp:** not necessarily] (e.g., together with checkpointing). [**gg:** clarify with Alfonso section 11.2]
  - This may be slower (in inter sub-net communication) but has the advantage of better isolation (and security) for sub-nets.
  - A different option would be to allow parent-child interactions continuously in an atomic execution manner. However, this increases the complexity of the system and enables edge cases. For example, when executed alone, sub-net validators can easily ignore a user's attempt to transfer funds from its parent account. This censorship incurs a higher cost when it is tied to checkpointing the entire system.
- Checkpointing is used for 2 purposes:
  1. To enact the token transfers between parent-child accounts of the same user. E.g., releasing some of a user's tokens that are locked in $\mathcal{SN}$ to its account in $\texttt{parent}(\mathcal{SN})$. This is what enables the hierarchy to function. Necessary![**gg:** also guarantees an account-grained firewall.][**arp:** same as withdrawals]
  2. There is another use for checkpoints that is unnecessary for the HC functionality but provides an additional benefit to the users. Periodically checkpointing the (complete) state of $\mathcal{SN}$ to its parent for added persistence that relies on the parent's guarantees. [**arp:** long-range attack mitigation]

  Notice that the two purposes of the checkpointing routine are theoretically independent of each other. In practice, however, they will probably be combined so further thought on the matter is warranted. Remark: while each sub-net can be a full SMR, for HCs functionality, only accounting information (balance) needs to be communicated between child and parent sub-nets.[**arp:** actually, only circulating supply suffices] [12]

[12] Simplifies things and should be sufficient. In order to reduce load from the parent, all of the changes that happened in the sub-net since the last checkpoint should be compressed into an efficient state-transfer operation. Hence, general state compression might be problematic. Accounts balance is easy to do and fits most uses. A future question would be extending to other states besides balances.

- The creation and maintenance of a sub-net $\mathcal{SN}$ is governed by an Actor in $\texttt{parent}(\mathcal{SN})$.

11.1 Checkpoint issues

TODO: explain compression issue.

A checkpoint can be seen as a transaction in the parent net.[**gg:** Although crypto-econ lab approach is different, that is, they suggest special treatment for messages/actions that come from an Actor governing a subnet. I believe in a simpler design in which the aforementioned message is treated as any other message. I conjecture that this natural design (no need for special treatment) would be more robust in practice.] However, several issues arise with regard to the cost of proposing a transaction. A typical transaction is proposed by the user who wishes to get it into the blockchain, and hence, carries the associated costs — specifically, the fee offered to the "miners". The proposing user, thus, decides solely on how much fee to offer. Simply putting, *the one who receives the goods also pays for it*. This forms the basis for classic economic theory of supply and demand. On the other hand, in the case of the checkpoint fee, all nodes/users of $\mathcal{SN}$ decide on the fee and pay it (through the Actor), but not necessarily all of them have the same benefit from the checkpoint. For example, a user that did not perform any transactions since the last checkpoint might consider the checkpoint less urgent than a user that wishes to release funds to its parent account. **This issue requires further analysis!**[**gg:** To provide some default as a starting point, we can use a simple algorithm at the Actor that sets the proposed fee according to past information on the parent.]

- If $\texttt{parent}(\mathcal{SN})$ employs EIP1559, the problem is significantly mitigated. Setting the proposed fee based on an algorithm in the Actor is relatively safe since miners revenue is mostly block rewards and fees are expected to be negligible. Hence, miners have negligible incentive to discriminate transactions based on fees. (That is assuming that we trust EIP1559.)

- If $\texttt{parent}(\mathcal{SN})$ employs the classic bidding mechanism to decide on which transactions to include in a block (i.e, highest fees win), the fact that the fee is payed from the "common" account is problematic. It might incentive miners to manipulate the Actor's fee-setting algorithm.

11.2 Parent-Child Interactions

- currently parent-child communication is asymmetric. Parents send transactions down continuously but children send upwards only through checkpoints. This could be problematic (e.g., liveness of the protocol? Potential DDoS/flooding attack from the parent?). Suggestion: Ini-

tially, make transactions symmetric — both directions happen through checkpoints.

- what is the subnet state included in the checkpoint that can be atomically modified with the top-down messages queued in the parent so the subnet knows the state from which to start again?
- how does a sub-net continues after a checkpoint? Is the checkpoint treated as a "genesis"?
- Can the sub-net continue optimistically? (probably, we can determine the delta and merge the "extra operations" included in the parent).

11.3 Inter sub-net Interactions

Conceptually, users are the ones that want to act, they do so by their accounts in the system. If a user wishes to interact with an asset that is in the control of sub-net $\mathcal{SN}$ (e.g., a smart contract), she should open an account in that sub-net. [**gg:** Enabling an account from subnet $\mathcal{SN}_1$ to interact with resources in $\mathcal{SN}_2$ opens the door to long-term troubles.]

Fundamentally, cross-net messages are a chain of (upward) interactions between accounts of the same user until reaching a common ancestor with the target, then an interaction within this ancestor subnet, and finally a chain of (downward) interactions to the target recipient. Since both the upward and downward chains happen within the same user, it is her responsibility to make them happen (when your brain tells your hand to move, I don't need to interfere). The only interaction between distinct users happens in the same subnet and then we are in charge of making sure it happens properly. Having said that, clearly, from a UX point of view, a gadget that helps the user execute the upward/downward chain of interactions will be important to users. A service that streamlines the process of inter sub-net interactions (from a client's perspective) could be build on top of our system. Nevertheless, in my opinion, cross-net messaging (with multiple hops) should not be a part of the core functionality but rather an add-on gadget.

12 Slashing

Slashing is an additional functionality that aims at increasing the user's trust in a subnet. In essence, by financially punishing misbehavior, participants are encouraged to follow the protocol.

Clearly, a general slashing mechanism that fits all possible subnets is impossible to devise. Hence, we suggest here a reference mechanism that fits several known Blockchain/SMR protocols [**?**, **?**, **?**].

12.1 Reference Slashing Mechanism

We start by outlining the key principles:

- The slashing is done at the parent. That means, the collateral of an $\mathcal{SN}$-validator is stored and managed at subnet $\texttt{parent}(\mathcal{SN})$.
- Slashing is done linearly with respect to the validator's voting power. E.g., if an equivocation from validator a with x_a of the voting power results in s_a deduction from a's collateral, then an equivocation from validator b with $x_b = 2x_a$ results in a deduction of $s_b = 2s_a$ from b's collateral.
- Offences that are more of a "malicious nature" should incur higher penalties than "benign" offences.

Example of slashable offences:

1. proposing equivocating blocks.
2. voting on more than one block per epoch.
3. proposing/voting on invalid block.

13 Miscellaneous

13.1 Shortcuts as a Service

Q: Can IBC be implemented independently as a "shortcut" between subnets in the "simple" model?
A general IBC channel between subnets might be problematic since it would damage the subnet isolation guarantees. However, a service providing a "shortcut" can be build as an application over the simple model, in a similar manner to bidirectional payment channels. A service provider opens 2 accounts – one in each subnet – and deposits tokens in each. Denote these subnets by $\mathcal{SN}_A$ and $\mathcal{SN}_B$, and the service accounts as $s_A \in \mathcal{SN}_A$ and $s_B \in \mathcal{SN}_B$. Now say that account a from subnet A wants to transfer m tokens to account $b \in \mathcal{SN}_B$. It can do so by transferring m tokens in $\mathcal{SN}_A$ to s_A and having s_B transfer m tokens in $\mathcal{SN}_B$ to b. Thereby creating a "shortcut" instead of traversing the hierarchy tree. Clearly, for this service a must pay fees to the service provider (to s_A).

[**arp:** this is HTLCs]The trust/risk in this service falls only on the ones who participate in it, namely, the users behind a and b, and the service provider. Smart contracts can govern this service (instead of trusting the service provider). Essentially, a deposits the tokens in the smart contract which releases them to the control of s_A given a proof that b have received the tokens from s_B. The proof is based on $\mathcal{SN}_B$ guarantees, for example,

the transaction appearing in $\mathcal{SN}_B$'s blockchain at a given depth. The tokens would be released back to a in case a timer expired without s_A providing the proof. To reduce costs in the common case, I suggest to include an "cooparation" release mechanism. By adding the possibility for a to approve that the transaction happened to its satisfaction, the need to include a proof from $\mathcal{SN}_B$ in $\mathcal{SN}_A$'s blockchain is mitigated, when everyone is honest. To encourage the use of the cooperation release by a, the service provider can offer reduced fees whenever the release happens due to a'a approval.

While this service entails liquidity costs (for the service provider), having it as an application on top of the HC framework provides several benefits. (1) Service providers are expected to create shortcuts where they are most profitable, which should typically correlate with where they are most needed. (2) Competition may arise among different service providers, driving for better service and reduced costs. (3) All is done without central intervention.

14 Pseudo-code

In algorithms 8 to 10 we omit the fee paid by user u to the governance account of subnet $\mathcal{SN}_C$.

Algorithm 8: Join (very similar to move funds down)

input: user u, child subnet $\mathcal{SN}_C$, parent subnet $\mathcal{PN}_P$, amount amt

verify that u has an account in $\mathcal{SN}_P$ and that it has enough funds in $\mathcal{SN}_P$

remove amt funds from u's account in $\mathcal{SN}_C$

add to the down-tx-Batch the transaction: $tx_{create}(u, amt)$ instantiating an account for u with amt funds in $\mathcal{SN}_C$

upon *Propose down-tx-Batch* **do**

    gov-account proposes down-tx-Batch in $\mathcal{SN}_C$ with fee fee from $\mathcal{SN}_C$ governance account `// this is an option for paying the fee.`

upon *Execute down-tx-Batch (in* $\mathcal{SN}_P$*)* **do**

    for *each* tx_{create} *in down-tx-Batch* **do**

        $(u, amt) \leftarrow tx_{create}$

        add account in $\mathcal{SN}_C.accounts$ for user u add amt funds to u's account in $\mathcal{SN}_C$

15 Notes form meeting at 2AUG22

There are two different models to consider: (1) the "IBC" model, and (2) the "simple" model.

In the IBC model [2] (which might be the initial intention of HC), accounts from different subnets can communicate with each other using bridges. This allows two accounts to perform cross subnet interactions directly using the IBC protocol, which in HCs case will be the cross-net message transfer mechanism.[**gg:** I need to better understand [2] to be sure, but it appears that interaction between two accounts from different subnets requires not only trust between the accounts but also that *the two subnets trust each other*.]

In the simple model (which is described in this document), accounts cannot communicate directly with other subnets, in general. Rather, their cross-net interactions are limited to basic operations between parent and child accounts. In particular, to token transfers between accounts of the same user. Moreover, in order for a user u_i to interact with a user u_j they must both have accounts on a shared subnet.

Besides the need for a more complex mechanism to support cross-net

Algorithm 9: Move funds down

input: user u, parent subnet $\mathcal{SN}_P$, child subnet $\mathcal{SN}_C$, amount *amt*

verify that u has accounts in both $\mathcal{SN}_P$ and $\mathcal{SN}_C$, and that it has enough funds in $\mathcal{SN}_P$

remove *amt* fund from u's account in $\mathcal{SN}_P$

add *tx* increasing u's account in $\mathcal{SN}_C$ by *amt* to the down-*tx*-Batch

upon *Propose down-tx-Batch* **do**

    gov-account proposes down-*tx*-Batch in $\mathcal{SN}_C$ with fee fee from $\mathcal{SN}_C$ governance account `// this is an option for paying the fee.`

upon *Execute down-tx-Batch (in* $\mathcal{SN}_C$*)* **do**

    for *each tx in down-tx-Batch* **do**

        $(u, amt) \leftarrow tx$

        add *amt* funds to u's account in $\mathcal{SN}_C$

`// If execution continuously fails (due to not enough fee) the money is lost.`

messages in the "IBC" approach, the main difference comes from what is the offered tradeoff. Essentially, in "IBC", the scalability comes on the expense of trust, while in the "simple" approach the scalability comes from locality, and is therefore limited by its existence. In particular, IBC requires stronger trust assumptions (a user needs to trust all subnets in the path of a cross-net interaction), but does not need to maintain a hierarchy of accounts, while "simple" only requires that a user trusts the common ancestor subnet, but also requires to have an account in the common ancestor.

Q: How much benefits can be drawn from the "simple" model (in terms of scalability)?

Q: How limiting is the stronger trust requirements on the usage of the IBC model?

Q: Can IBC be implemented independently as a "shortcut" between subnets in the "simple" model?

15.1 Additional points

- It might be that batching is the main benefit of implementing IBC via HC. In that case, we should consider whether this can be treated orthogonally (e.g., via a dedicated smart contract), rather than entangling it with HC. On the other hand, maybe the entanglement provides implementation benefits, e.g., speedup. [**arp:** probs in same smart contract but imo added feature: not necessary to spawn a subnet. 'Simple' -¿ base functionality, IBC -¿ optional added features]

Algorithm 10: Move funds up

input: user u, child subnet $\mathcal{SN}_C$, parent subnet $\mathcal{SN}_P$, amount *amt*

verify that u has accounts in both $\mathcal{SN}_C$ and $\mathcal{SN}_P$, and that it has enough funds in $\mathcal{SN}_C$

remove *amt* funds from u's account in $\mathcal{SN}_C$

add *tx* increasing u's account in $\mathcal{SN}_P$ by *amt* to the up-*tx*-Batch

upon *Propose up-tx-Batch* **do**

    gov-account proposes up-*tx*-Batch in $\mathcal{SN}_P$ with fee fee from $\mathcal{SN}_C$ governance account `// this is an option for paying the fee.`

upon *Execute up-tx-Batch (in* $\mathcal{SN}_P$*)* **do**

    for *each tx in up-tx-Batch* **do**

        $(u, amt) \leftarrow tx$

        add *amt* funds to u's account in $\mathcal{SN}_P$

`// If execution continuously fails (due to not enough fee) the money is lost.`

- It appears that currently the system requires a single "almighty" token to support the IBC implementation (which is sufficient for MVP). Using the simple approach, naturally supports internal token generation. The only requirement is that the balances of $\mathcal{SN}$ in `parent`($\mathcal{SN}$) are in a token that is supported in `parent`($\mathcal{SN}$).

16 Meeting Notes from Vaduz

Money / coins / tokens can only be sent from a concrete account at the parent to a concrete account on the subnet

SA: Subnet actor (smart contract) on the parent, holding all information relevant for a single subnet. G: Guy's account on the parent g: Guy's account on the subnet g': Guy's subnet balance as seen by the parent, represented as an entry in the SA's account balances.

State of the smart contract contains:

- Account balances for all the children accounts in the subnet.
- A governance account and an associated policy how this account is funded and under which conditions money is transferred from it. The purpose of a governance account is to store funds associated with running the subnet (the subnet's "commons"). E.g., the transaction fees incurred by submitting checkpoints might be reimbursed from the governance account (under conditions specified by some policy). It can

Algorithm 11: Create Checkpoint

input: *prevCP*

1 *newCP*← get-snapshot($\mathcal{SN}_C$) `// a recent state of` $\mathcal{SN}_C$ `individual accounts`

2 verify-snapshot(*newCP*) `// E.g., a valid successor of` *prevCP* `and signed by majority of validators.`

3 *diff*← calculate diff from prevCP

4 propose in $\mathcal{SN}_P$ to update *prevCP* with *diff* `// fee should be paid from governance account`

5 **for** *each validator in* $\mathcal{SN}_C$ **do**

6 **upon** *application of newCP in* $\mathcal{SN}_P$ **do**

7 only offspring of *newCP* will now be validated

Algorithm 12: Default fee setting algorithm

1 to do...

be funded, for example, by transaction fees within the subnet. The governance account might be involved in providing incentives for child validators to participate (rewards) and / or to behave correctly (slashing). The governance account has a corresponding account in the child subnet as well. It can be funded, e.g., through withdrawals from the child subnet.

- Validators' set + metadata (e.g. voting rights/collateral/stake/...)
  - Consensus protocol
  - Subnet configuration (probably a pointer to the data)
  - Governance rules
- A function $checkProofOfGlobalFinality(localState, proof) \rightarrow bool$. This function defines what constitutes a proof of global finality of a state of the child. E.g., this could be a set of signatures of subnet validators. For a longest-chain-based subnet, the definition of this function would involve a depth parameter, such that all blocks deeper than depth are considered final. If a reorganization of the child happens deeper than that depth, it can be considered as a failure of the whole subnet.
- Slashing functionality [**arp:** and checkpointing rules]

Interactions between parent and child It is likely that in practice, for efficiency reasons, multiple interactions will be batched together (possibly

Algorithm 13: Example slashing mechanism (Depends on specific consensus mechanism)

input: *validator, proof*

if *proof that validator voted for more than 1 block in epoch* **then**
  slash *validator* completely

if *proof that validator proposed more than 1 block in epoch* **then**
  slash *validator* completely

if *proof that validator proposed more than 1 block in epoch* **then**
  slash *validator* completely

if *proof that validator proposed an invalid block* **then**
  slash *validator* by a factor of 1/10

even combining withdrawals, deposits and checkpoint into a single transaction). Nevertheless, we present here a cleaner version where operations are handled independently as it trivially generalizes to the batched version.

- **DEPOSIT money (e.g. 2 coins) (transfer from G to g).**
  - $\langle TX(G \rightarrow^{+2} SA : g')\rangle_{Guy}$: Transaction sending 2 coins from G to $SA : g'$, signed by Guy.
  - Observer(s) of parent state create a TX for the child: $TX(g + 2, proof)$ that adds 2 to g (when accepted as valid). The validity of this transaction must only depend on the transactions preceding it and itself. Example options for checking the validity of $TX(g + 2, proof)$:
    1. **on-chain voting**: When locally observing the parent state change, each subnet validator submits $\langle TX(g + 2)\rangle_{Validator}$ to the child SMR system. When a quorum (e.g. $f + 1$) such transactions signed by different validators are ordered, the last of them has the effect of increasing g by 2.
    2. **off-chain voting**: Like on-chain voting, but all the transactions (signed by different validators) are gathered off-band and submitted (by any client to the child SMR system) as a single transaction $TX(g+2, proof)$, where the $proof$ consists of a quorum of child subnet validators' signatures.
    3. **local validity check** (simpler, efficient, *weaker guarantees*): $proof$ contains a pointer to the block containing $\langle TX(G \rightarrow^{+2} SA : g')\rangle_{Guy}$ at the parent, together with the height h of that block. To validate the TX 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.

4. **subnet protocol integration**: Using PBFT as an example. Any client of the child subnet submits $TX(g+2)$ to the child (one is enough). On reception of a PBFT Preprepare message, a validator only sends the Prepare message once it observes the corresponding parent state change. This way, the child subnet's validators implicitly vote on the validity of $TX(g+2)$. Once it is committed, it is considered valid by all validators, even those that did not vote for it.

- **WITHDRAW money (e.g. 2 coins) (transfer from g to G)**
  - $\langle TX(g-2)\rangle_{Guy}$ (transaction "burning" 2 coins from g, signed by Guy) on the child.
  - Observer(s) of child state create a TX for the parent: $\langle TX(SA : g' \rightarrow^{+2} G, proof)\rangle_{Guy}$ that decreases SA:g' by 2 and adds 2 to G (when accepted as valid). Validating this transaction must only depend on $proof$ and SAs internal state (which is visible at the parent). Example options for checking the validity of $\langle TX(SA : g' \rightarrow^{+2} G, proof)\rangle_{Guy}$:
    1. **Threshold signature from the child subnet validators**:[13] $proof$ contains a threshold signature from the child validators (which are listed in SA). A partial signature (for the threshold) from a validator v affirms that v considers $\langle TX(g-2)\rangle_{Guy}$ as stable in the child subnet.
    2. **Direct voting of the child validators:**[14] A quorum of the child validators submits a transaction (each validator a separate transaction) $\langle TX(SA : g' \rightarrow^{+2} G)\rangle_{Guy}$ to the parent. Only after enough (desired quorum size based on the validators is SA) such messages are appended to the parent ledger, the transaction TX takes effect (transferring the money from $SA : g'$ to G).

- CHECKPOINT (include a representation of / reference to a particular version of the child's whole replicated state in the parent's replicated state) [**gg:** Think on who pays (and how) for TX at the parent.]
  - Observer(s) of child state identifies a condition for creating a checkpoint. Example for such conditions:
    1. Δ blocks were appended to the child SMR since previous transaction,
    2. Δ money changed hands since previous checkpoint,

[13] An example is an $f+1$ threshold signature.
[14] This is applicable very generally. However, it might be inefficient.

3. enough fees were collected in a checkpoint request smart contract.
4. Validator's will (most likely at its own expense, not that of the governance account). This could be a good approach where above conditions do not justify the overhead of periodic checkpoints (an instead this can be left open to incentives)

– The observer then creates *data* and *proof* for the checkpoint, where *data* is the checkpoint data (or reference thereof) and *proof* is a proof of the validity of the data. Example options for *proof* and *data*:

1. *data* is the diff between the state of the accounts in SA (e.g., $SA : g'$) and the state of the accounts at the child SMR (e.g. g). *proof* is a threshold signature from a quorum of the child subnet validators confirming *data* as well as the condition for creating a checkpoint. In addition, the *diff* might be required to sum to 0.
2. [**gg:** provide additional example. ZK]

– The observer submits a checkpoint TX for the parent: $\langle TX(data, proof)\rangle_{Guy}$ that updates the state of SA according to *data* if TX is valid. Validating this transaction must only depend on *proof* and SAs internal state (which is visible at the parent). Note that the fee for TX is payed by G (the signer on TX), hence it should be reimbursed by SAs governance account upon the execution of TX.[15]

– SA updates its state according to TX and might reimburse Guy for TX's fee from the governance account according to the SA's policy.

- REPORT violation (for slashing purposes): The SA contains a *slashing policy (sp)*. This policy defines

1. What constitutes a proof of misbehavior (PoM) ($sp.validate(PoM) -> bool$)
2. The consequences of submitting a valid PoM ($sp.apply(PoM, metadata)$)

Slashing works as follows:

1. Any user submits $\langle TX(PoM, metadata)\rangle$ to the parent. Metadata contains additional information about the TX, e.g., the identity of the submitting user.

[15] If there is a mechanism to directly charge SA:governance with the fee for a transaction signed by Guy, it would be better.

2. The execution of this transaction triggers a call to $SA.REPORT(PoM, metadata)$ (method of the SA).
3. The implementation of the method invokes $sp.validate(PoM)$. The validity of the PoM must only depend on the state of the SA and the PoM itself.
4. If the PoM is valid, the SA invokes $sp.apply(PoM, metadata)$ that changes the internal state of SA and may produce additional internal transactions (applied directly to the state of the parent chain). For example, this might result in transferring part of the collateral of the offending child validator (identified through the PoM) that is locked in the SA to the governance account (or burning it), and/or transferring part of the validator's collateral to the user that submitted the PoM transaction (identified it the transaction's *metadata*).

- **POST-OFFICE for inter-subnet transactions:** SA contains a functionality that can be used to transfer data from one subnet to another. In particular, consider the following case involving a smart contract (which, therefore, cannot be solved by deposit and withdraw operations).[16] Smart contract *SmCt* emits an event e that contains *data* which is desired to reach the destination *dest*.

  1. *SmCt* calls the functionality POST-OFFICE.*propagate*(*data*, *dest*). According to *dest*, this triggers a call to PropUp, PropDn or PropHere. (We leave out slight optimization for readability.)

  Below we describe only PropUp as PropDn can be deducted from it and PropHere is trivial.

  2. The functionality PropUp handles a call e only if one of the following holds:
     - **A subnet-local call:** PropUp(*data*, *dest*) is called by a smart contract in the same subnet as PropUp. In that case, it emits an event *POST-OFFICE.propagateUp* = (*src* = *SmCt*, *data*, *dest*) that will be propagated up.
     - **A propagate from child call:** PropUp(*POST-OFFICE.propagateUp*, *proof*) is called, where *proof* validates that *POST-OFFICE.propagateUp* was emitted in the child subnet (that is on the path to *propagateUp.src*) by the POST-OFFICE functionality in the child. Examples for *proof* are similar to those of WITHDRAW. In that case, PropUp emits an event *propagateUp* = (*src* = *SmCt*, *data*, *dest*) that will be propagated up.

[16] When the data transfers is between Externally Owned Users, it is typically better to use withdraw and deposit operations.

3. An observer of the child state submits a TX for the parent: $\langle TX(propagateUp, proof)\rangle_{Guy}$. Validating this transaction must only depend on *proof* and SAs internal state (which is visible at the parent).
4. At the parent, once the proposal $\langle TX(propagateUp, proof)\rangle_{Guy}$ is committed, it triggers *POST-OFFICE.propagate* at the parent.

Note that at step 3 the EOA belonging to Guy pays the gas costs for TX. A mechanism for incentivizing Guy to propose TX is warranted. Purposely, we do not entangle the incentive with the propagate functionality, instead we allow for different incentive mechanisms to be used. Example for incentive mechanisms:

- ***SmCt* is part of a smart contracts network:** This network (which has smart contracts covering the path) will reimburse Guy.
- **A third party service:** *SmCt* pays a service that has an account in its subnet. This service has has proposers (or smart contracts to incentivize them) in every subnet on the path that will propose.
- **An interested EOA:** A user that is interested in the success of the transaction and has accounts covering the path proposes the propagating TXs and pays the fees.
- **POST-OFFICE as a third-party service:** Equivalently to a third-party service, the POST-OFFICE can provide proposing reimbursement services for entire network. We suggest to add this as an explicit opt-in option (not a default). Moreover, to reduce user disappointments and to encourage the deployment of advanced services, we suggest that the POST-OFFICE charges very conservatively (expensive prices) for the proposing incentivization service.

We stress that no incentive mechanism can guarantee that TX reaches from *src* to *dest*, since the required fees for the entire path are, in general, not known in advance.

Agreeing on parent state changes (e.g.)

16.1 Interface between software modules

In addition to the smart contract SA, we separate the software needed to run IPC into three abstract modules with defined interfaces:

- **Parent validator:** The software that runs the parent blockchain. Note that in this module is also in charge of interacting with the IPC smart contract SA, which is maintained at the parent subnet. Any

update that the parent validator performs on the SA is notified to the IPC module.

- **Child validator:** 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 validator through the IPC module.
- **IPC module:** 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 this is not the a smart contract (it is not SA). It is a piece of software that runs at a node and mediates the interactions between the child and parent validator software.

These three pieces of software interact through interfaces that consume and produce events.

16.1.1 Deposits and Withdrawals

We show in Figures 6 and 7 the events that each interface consumes and produces during deposits and withdrawals, respectively.
Any other operation that augments withdrawals (e.g. checkpoints) is analogous to withdraw operations. Checkpoints may change in the payload being transferred, but require the same events to be committed/aborted.
We proceed to explain each event being consumed/produced at each module.

Child Validator
Deposits. The child validator consumes an event `Deposited(TX)` produced by the IPC module upon which it updates its state in order to reflect this deposit.

Withdrawals. During operations that withdraw from child to parent subnets, the child validator first consumes a `WITHDRAW(TX)` from a user (or users) of the child subnet and then produces a `ChildWithdrawn(TX, [proof])` event to send to the IPC module. Depending on the type of subnet, the child validator can already provide a proof of global finality. If the child validator does not produce a proof of global finality then it is the job of the IPC module to produce such proof for the parent.

IPC Module
Deposits. The child validator first consumes a `ParentDeposited(TX)` transaction from the parent validator, and then performs the necessary operations to confirm the inclusion of the deposit into the child, depending on the type

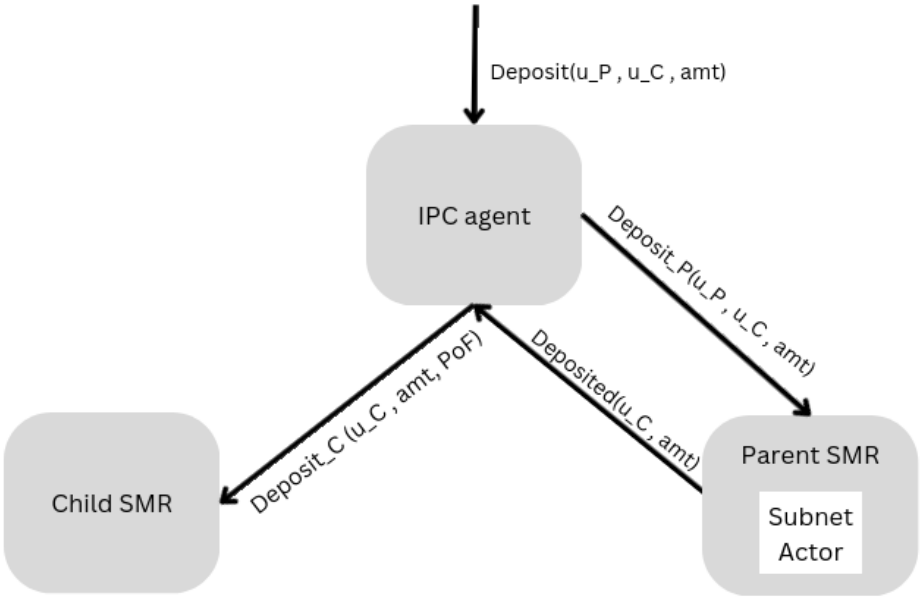

Figure 6: Events produced and consumed during a deposit.

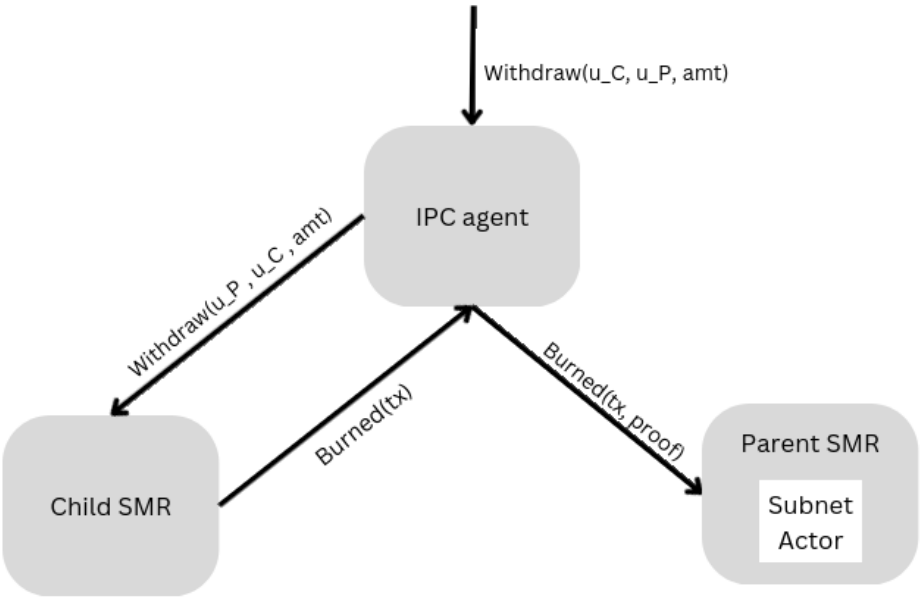

Figure 7: Events produced and consumed during a withdrawal.

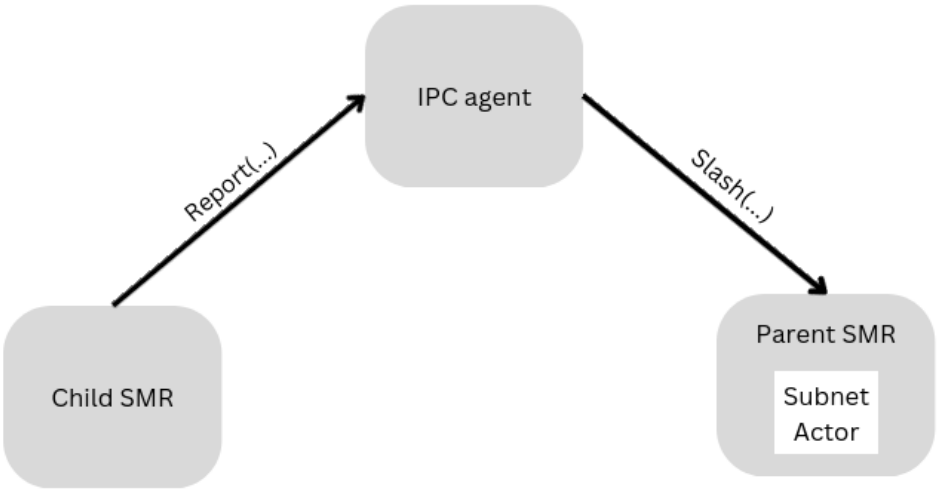

Figure 8: Events produced and consumed during a report.

of deposit defined by the SA for this subnet and listed above in this document: on-chain voting, off-chain voting, local validity, etc. Once this deposit is confirmed by the IPC module, then the IPC module produces a `Deposited` event to be consumed by the child validator. The IPC module may also locally abort and not produce any events during deposits, should the deposit not be valid.

Withdrawals. During withdrawals, the IPC module first consumes a `ChildWithdrawn(TX, [proof])` event from the child validator, after which it produces a proof of the withdrawal that is valid for the SA at the parent (if the child subnet did not produce one already) and produces a `ChildWithdrawn(TX, proof)` to be consumed by the parent validator. **Parent Validator**
Deposits. The parent validator consumes a `Deposit(TX)` from a user (or users) and then produces a `ParentDeposited(TX)` event to be consumed by the IPC module.

Withdrawals. During withdrawals, the Parent Validator consumes a `ChildWithdrawn(TX,proof)` event.

16.1.2 Other operations

We show reports in Figure 8. If the child validator detects provable misbehaviour, it notifies the IPC module through a `Report` event. Then, the IPC module follows the same path to request and notify slashing of the responsible deviants as the one showed for the withdrawal case.
Checkpoints. Checkpoints should follow the same procedure as withdrawals, except for: (i) the initial event that triggers the operation, as this is not

necessarily a a transaction submitted by a user of the child subnet (most likely triggered by the IPC module); (ii) the payload of the events in the operation; and (iii) the local treatment of the events. The rationale for the events that are being triggered is analogous. [**arp:** TODO: Propagate]

16.2 additional points and future topics

Non essential thing to think about in the future.

- Entity observing parent replicated state and submitting corresponding transactions to the child SMR system (could be the user's laptop, a validator node, a third party, ...)
- finality function possibilities. E.g., varying according to amount, potentially slashed amount, network/faulty assumptions, etc. -¿ trade-off performance/security
- atomicity operations (atomic swap/atomic execution), IBC-like bridges
- Alfonso: Fraud proofs and gas fees distribution. Two under-defined.
- Alfonso: if you don't mind including to your backlog of under-defined things: `https://github.com/consensus-shipyard/ipc-actors/issues/22` Also out of scope for M2, but really worth having for M3 and to answer future questions

References

[1] Alfonso de la Rocha, Lefteris Kokoris-Kogias, Jorge M Soares, and Marko Vukolic. Hierarchical consensus: A horizontal scaling framework for blockchains. In *5th International Symposium on Foundations and Applications of Blockchain 2022 (FAB 2022)*, volume 101, page 3. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2022.

[2] Christopher Goes. The interblockchain communication protocol: An overview. *arXiv preprint arXiv:2006.15918*, 2020.