IPC with Tendermint Core

Background

This document is an elaboration of Option E in the IPC Pathways. I wanted to present the case for using Tendermint as an off-the-shelf SMR in subnets, and use the opportunity to answer some of the questions and comments the proposal received over there.

What is Tendermint Core?

The introductory documentation describes it very well.

Tendermint Core is a *generic* blockchain SMR system; it's generic in the sense that it allows us to **plug in** our own *application logic* **regardless of what language it's written in**: it can be Go, Rust, Java, Haskell, Scheme, etc. It allows us to concentrate on our domain, and takes care of *consensus* and the general blockchain machinery, such as chain sync, snapshots, mempool, query interfaces, light clients, etc. Transactions are *completely opaque* to Tendermint Core, and it does *not provide state storage*, that is an application concern.

The consensus is the bit which is not generic, it's the Tendermint BFT consensus and that's that. It has a few characteristics that make writing applications for it simpler than it is for longest chain consensus: Tendermint blocks are finalized *before* they are executed, therefore there is never a need to roll back the state, there are no reorgs. All the application has to be able to do is apply a block of transactions, and commit at the end when Tendermint says so.

The best way to get a feel for this is to take a quick look at the tutorial, which is about building a key value store in Go, Java or Kotlin. In the tutorial, transactions (what we call Messages in Filecoin) are just strings in the form of *key=value* and they set values in a database. Queries are just the keys, to which we respond with the value.

When using Go, we have the option of a built-in (ie. running in the same process as Tendermint Core) or an external app - for the other languages we have to run externally and handle calls from Tendermint via gRPC.

Tendermint Core is not Cosmos

It bears emphasizing that Tendermint Core is a separate, independent application from the Cosmos SDK. The Cosmos SDK is a Tendermint Core Application. It is written in Go and provides a whole host of *ledger* functionalities: transaction format, storage (ie. authenticated data structures), optional modules for accounts, banking, staking, slashing, evidence, etc.

Blockchain developers can pick and choose the modules they need and roll *just* what is different for their ledger. They can also use any library that makes sense for their application, because each Cosmos Zone is a separate blockchain, just running a single application.

In this proposal, we would not be using the Cosmos SDK, only Tendermint Core.

Why is this cool?

This is cool because it allows us to write an application layer any way we can conceive it: we can reuse some parts that were developed in Eudico in Go, and mix it with Rust if if it makes sense; we can use the FVM and the built-in actors, and we can use an IPLD store. Tendermint doesn't know and doesn't care what we are doing.

This would allow us to cut straight to the chase and focus on IPC, rather than a risky refactoring exercise.

Why is this different from using Lotus?

It's different because Tendermint Core was *made to be generic*, it was made to assume nothing about what application we are trying to run. This is not the case with Lotus (and Forest), which was made to be a Filecoin client, and assumes in a lot of places that we are using Expected Consensus, with certain limits on what the blockchain can look like.

It is very difficult to refactor something like Lotus/Forest into a generic blockchain *with a different consensus algorithm* and be confident that there is no hidden assumption left *somewhere* in the code that will at some point cause a consensus failure. I tried.

Application Interface

Tendermint Core expects the application to implement the ABCI (Application Blockchain Interface). Here's a good overview. The TD;DR version of it is that validators take turn proposing blocks, they vote for it, a (weighted) quorum of signatures finalizes the block, *then* it's delivered to the application in the form of *BeginBlock -> DeliverTx* -> EndBlock -> Commit* RPC calls.

This implies a malicious validator can include any transaction it wants, because during the voting process Tendermint doesn't consult the application. The application has to be prepared for invalid transactions in the block, mark them as such, and move on. A newer interface changes this; we'll see it in a bit.

Proof-of-Stake

The response to *EndBlock* can contain an update to the power table of Tendermint validators, which is how the application can adjust the weights in the Proof-of-Stake consensus. How and

why those changes take place are up to the application, which in our case would be the IPC spec.

ABCI++

The next version of Tendermint Core will update to this formula by introducing the new ABCI++ interface. Here's an overview of the basic concepts. It contains some changes which are crucial for us:

- It allows the application to inspect and modify the transactions in the block proposal *before they are sent out* to the other validators.
- It allows the application to inspect the proposed block *before casting a vote*.

This means we can take control of which blocks can become part of the chain before they are finalized, which we can use in a few different ways to achieve IPC:

- We can reject blocks that propose bottom-up messages we cannot resolve from the child subnet.
- We can reject blocks that propose top-down messages that aren't final in the parent subnet.

Here's the tracking issue of these two features being ported to the main Tendermint Core branch.

IPLD

One of the more under-appreciated aspects of what we're building is that it has to use content addressing and IPLD. It's not immediately obvious why this is a good choice or how it matters.

One of the most prominent places this appears in IPC are the cross-chain messages: checkpoints contain a list of CIDs of checkpointed messages (or a CID of a data structure that contains a list of CIDs). These messages need to be resolved before they can be executed. The resolution can happen via Graphsync or bitswap, which can follow embedded CID links.

A message such as this would be problematic if it was used with vanilla Tendermint, (that is, without using ABCI++), where blocks are final as soon as they are proposed, because a malicious validator or subnet could include CIDs that cannot be resolved, but no individual validator could be sure that it's just them, just temporarily, or is the data unavailable for everyone else.

Using the ABCI++ machinery we can just *not vote* on things that aren't available. This way, a quorum over a block implicitly becomes an *availability certificate* for the CIDs contained in it.

We just have to make sure that we don't stall consensus by not voting for any block, because something is always missing or takes too long to resolve.

Block Structure with CIDs

Here's one option to support CIDs as messages in blocks, while mitigating the chance of stalling consensus, and maximizing throughput. It is inspired by NC-Max, used by Nervos.

We can divide the transactions in the blocks into two types (by wrapping them into some meta-data that is meaningful to our application, not Tendermint):

- Transactions we propose for *execution* in the block
- Transactions we propose for *resolution*, to be executed in a later block

The voting process would be simple:

- Do not vote on a block that proposes for *execution* a CID we don't already have.

Execution would be different for the two kinds:

- CIDs proposed for resolution would be handed to a separate component that uses Graphsync or Bitswap to procure the content, asynchronously.
- CIDs proposed for execution would be available on the majority of nodes (because a quorum of them signed the block), but some validators might have to resolve the CID *synchronously* from their more hipster peers.

Transactions would enter the system in two ways:

- When Tendermint receives a transaction via API, it passes it to *CheckTx*, which is where our application can kick off the asynchronous resolution. Tendermint will use the mempool to broadcast these transactions too, but I think the application doesn't get notified about that.
- When Tendermint proposes a new block, it gathers all transactions from the mempool, then passes them to the application via ABCI++; here the application could 1) remove transactions it still hasn't resolved 2) resolve anything new and 3) add ones that have been resolved as proposals for execution.

Using blocks like the above, and allowing side-effecting transaction handling in the application layer (that is, talking to the network instead of just the ledger), we can use CIDs with Tendermint Core without having to change the code to add something like the cross-chain message pool mentioned in the IPC spec. It also removes the content resolution from the critical path of block execution and consensus, increasing throughput.

Once again, this broadcast mechanism is redundant with the mempool, but it's a way that enforces the communication of CIDs to the application before they become an immediate concern. *CheckTx* I think only kicks in on the node that receives the transaction from a user.

Message Enveloping

IPLD being a graph suggests that we could rely on it for storage efficiency as well.

IPC arranges subnets in a tree, and we assume that a validator in a subnet runs full nodes all the ancestor subnets all the way to the root node. A full node means they run consensus and execute transactions, but don't necessarily propose blocks, ie. they don't run validator nodes in each subnet.

By this arrangement, we claim that a validator can immediately see what the state of consensus is in the parent subnet, which is why top-down messages can be sent towards the child subnet *as soon as they are final*. These top-down messages, and their nested CIDs, are immediately available to all full nodes in the parent subnet, which includes all validators in the child subnet.

It would be a shame if we had to necessarily duplicate the entire message for each level in the hierarchy. We should be able to reclaim some efficiency by reusing the CIDs inside messages, only wrapping them with a thin layer of meta-data necessary for routing.

The same is true for bottom-up messages, except that they are not immediately available: a bottom up message has to be proposed by someone in the parent subnet, and then resolved by the other validators, before it can be executed. The benefit of enveloping is the same, though: full nodes participating in multiple heights in the hierarchy would only resolve the "meat" of a message once.

On the other hand, having separate IPLD stores and going through content resolution between each subnet facilitates easier garbage collection: if a validator doesn't want to be part of a subnet any more, they can just drop the entire storage that served that subnet.

Architecture Proposal

One benefit of trying to use Tendermint Core for IPC is that it *forces* us to think modularly, because we **don't want to fork** Tendermint. Forking would mean any changes we made would have to be audited. By delivering modular components we can maximize our chances that we can reuse them in subnets where we don't want to use Tendermint.

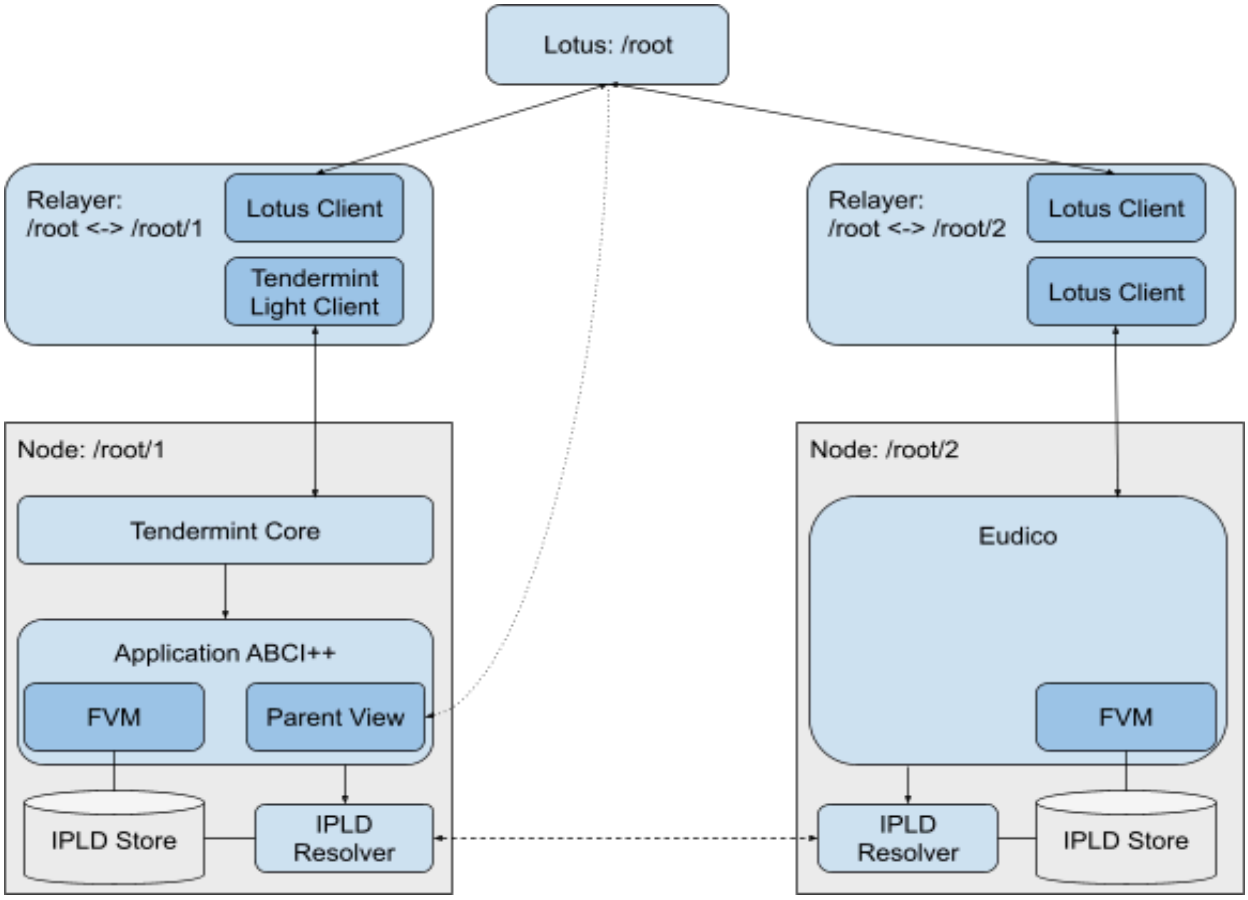
The following diagram depicts the root subnet and two different subnets: one running with Tendermint Core and another running with Eudico.

Notable components:

- **Lotus rootnet**: Lotus has to support IPC to act as the rootnet for its child subnets. There are competing proposals with regards to this being in the form of mostly user-deployed smart contracts, or built-in (privileged) capabilities.
- **Tendermint Core**: acts as the generic SMR in one of the subnets, talking to all other Tendermint instances in that subnet. It runs separate from the Application, which is completely in our control, and talks to it via ABCI++.
- **Application ABCI++**: This is where we implement the IPC ledger logic, the transaction handling, using the FVM/FEVM. We implement the ABCI++ interface to be compatible with Tendermint. Other than that we delegate to reusable smart contracts, for example to

produce checkpoints. We rely on ABCI to pass us the headers to be signed, and we can use the ledger to gather signatures for checkpoints. Alternatively we can potentially use the ABCI++ vote extensions to enforce checkpoints.

- **Parent View**: The Application *might* observe the parent subnet consensus state directly for the purpose of *voting*. The goal here is to deal with the fact that Lotus might roll back a message, so we can't execute a top-down message as soon as it appears, we have to wait for it to be embedded. We can make sure of this either by a) including a light client of the parent in the child ledger or b) using the voting process and let validators take a peek at the other chain.
- **IPLD Resolver**: A separate process (or a library) that the Application can contact to resolve CIDs and store them in the IPLD Store. It maintains connection with other nodes in the parent/child subnets, or perhaps even beyond.
- **IPLD Store**: A common store available for read/write for both the FVM and the IPLD Resolver. There might be a separate instance for each subnet, or one larger serving the needs of all subnets of an operator.
- **FVM**: This is our execution layer. Subnets can use FEVM if they want.
- **Relayer**: The role of relayers is to shovel messages between parent and child subnets. They have to follow both the parent and the child consensus, subscribe to events, re-package the messages in the appropriate formats and resend them. How they are incentivized to do so is an open-ended question. They should be trustless.
  Note that both subnets can have an entirely different block structure and consensus, and it's only the relayers that understand both, by being purposefully constructed to act between certain combinations.
  By contrast, if their functions were built into a common IPC orchestrator, then adding support for Tendermint would require that orchestrator to contain a Tendermint light client as well as any other potential subnet client it's supposed to work with.

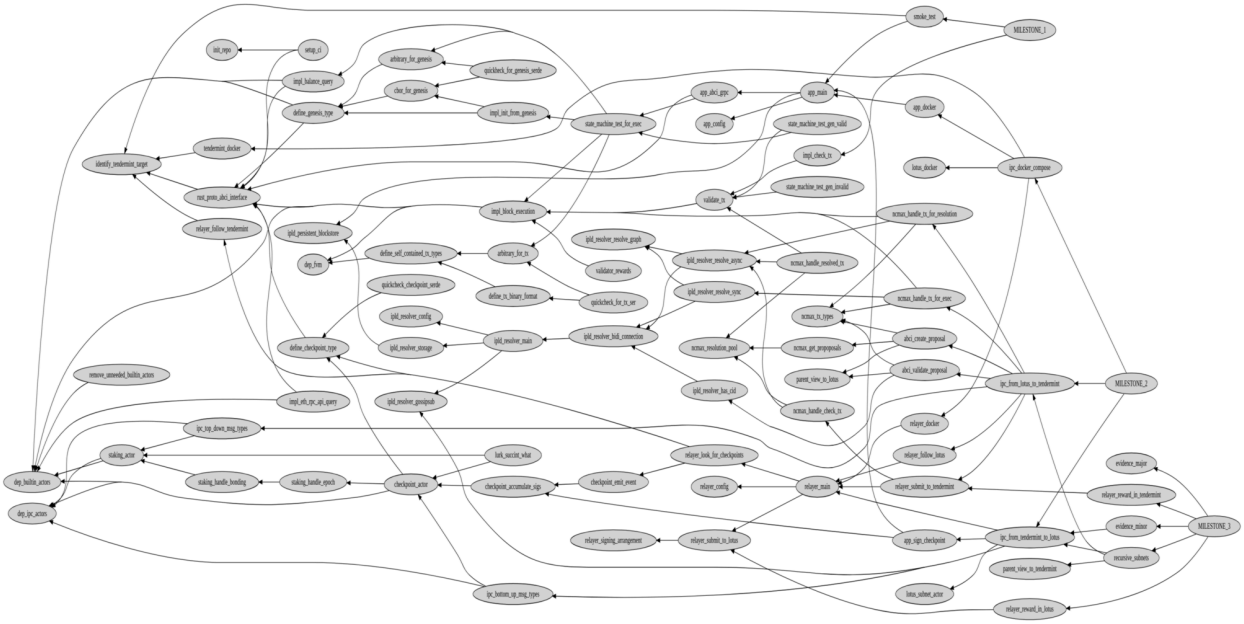


Blue rectangles are processes; darker blues are in-process libraries; gray rectangles are pods of multiple processes.

Roadmap

A tentative roadmap laying out the tasks and dependencies to implement the proposals in this document can be found here. A snapshot of the task dependency graph is included below.

(NOTE: It's better to open the roadmap spreadsheet and visualize from there; select PNG in the formats when the graph is shown, then click open in a new tab in the image context menu, and click again to zoom in. An Google Docs image zoom Chrome extension like this also helps.)


smoke_test
MILESTONE_1
init_repo
setup_ci
arbitrary_for_genesis
quickcheck_for_genesis_serde
impl_balance_query
app_abci_grpc
app_main
app_docker
define_genesis_type
clear_for_genesis
impl_init_from_genesis
state_machine_test_for_exec
app_config
state_machine_test_gen_valid
tendermint_docker
impl_check_tx
identify_tendermint_target
lotus_docker
ipc_docker_compose
state_machine_test_gen_invalid
root_errno_abci_interface
validate_tx
impl_block_execution
ncmax_handle_tx_for_resolution
relayer_follow_tendermint
ipld_persistent_blockstore
ipld_resolver_resolve_graph
dep_fvm
define_self_contained_tx_types
arbitrary_for_tx
ipld_resolver_resolve_async
ncmax_handle_resolved_tx
validate_rewards
quickcheck_checkpoint_serde
define_tx_binary_format
quickcheck_for_tx_ser
ipld_resolver_resolve_sync
ncmax_handle_tx_for_exec
ncmax_tx_types
ipld_resolver_config
ipld_resolver_main
ipld_resolver_bidi_connection
ncmax_resolution_pool
ncmax_get_proposals
abci_create_proposal
define_checkpoint_type
ipld_resolver_storage
parent_view_to_lotus
abci_validate_proposal
ipc_from_lotus_to_tendermint
MILESTONE_2
remove_unneeded_builtin_actors
ipld_resolver_lan_cid
ncmax_handle_check_tx
impl_eth_rpc_api_query
ipld_resolver_gossipsub
relayer_docker
ipc_top_down_msg_types
relayer_follow_lotus
staking_actor
lark_succint_what
relayer_look_for_checkpoints
evidence_major
dep_builtin_actors
staking_handle_bonding
staking_handle_epoch
checkpoint_actor
checkpoint_accumulate_sigs
checkpoint_emit_event
relayer_config
relayer_main
relayer_submit_to_tendermint
relayer_reward_in_tendermint
dep_ipc_actors
evidence_minor
MILESTONE_3
relayer_signing_arrangement
relayer_submit_to_lotus
app_sign_checkpoint
ipc_from_tendermint_to_lotus
recursive_subnets
parent_view_to_tendermint
ipc_bottom_up_msg_types
lotus_subnet_actor
relayer_reward_in_lotus