## Background

(feel free to skip if you're familiar with the context)

Recall the CAP theorem, which states that any distributed database system can only pick two of:

- Consistency (reads return latest write)
- Availability (reads return)
- Partition tolerance (progress continues even when arbitrary messages are dropped)

In a Venn diagram of possible choices:

[

CAP_Theorem_Venn_Diagram

1048×1044 85.4 KB

](https://europe1.discourse-cdn.com/standard20/uploads/anoma1/original/1X/1daf498e1c8c24f357fb545813c45882d1b414f2.png)

Consider the interchain ecosystem (different L1s, rollups, Cosmos zones, etc.). Let's refer to each of these components as domains

, where different domains can have both heterogeneous security and heterogeneous clocks (so shards would also count, if they run in parallel).

Which class do these domains fall into?

- Most domains are CP (consistent and partition tolerant). BFT consensus (e.g. Tendermint, so all Cosmos chains), Near, decentralised sequencer proposals based on BFT systems, etc. are all CP; under network partitions no progress is made.
- Nakamoto consensus chains such as Bitcoin are AP (available and partition tolerant); under network partitions they lose consistency.
- Hybrid chains with Nakamoto-esque (longest chain) block production and a separate finality gadget - e.g. Ethereum, Polkadot - are AP from the perspective of block production, but CP from the perspective of the finality gadget, and from the perspective of the rest of the system they are typically treated as CP (otherwise consistency would be lost if blocks were accepted before being finalized).

So far, nothing new. However, these domains are interconnected, with message-passing provided by bridges, IBC, ZK light clients, etc. - so we can also consider the whole interconnected set as one database system, and ask the same question of the interchain as such - which class does it fall into?

If we understand the chains as processes, I think the interchain is basically an AP system - individual chains will return reads even when they haven't heard from others in awhile, parts of the system will continue making progress under partitions, etc. Current bridges send messages authenticated in some fashion, but they typically don't do much in the way of detecting or preventing consistency violations (at most, equivocation can be detected, e.g. with Tendermint light client misbehaviour evidence). Operating this way, we give up consistency - chains might disagree with each other on what another chain said - and there's no easy way to even check whether a subset of the system is consistent at any point in logical time.

Suppose that we had some way of merging history from one chain into another. Then, the interchain would be an AP system where individual reads can choose to be consistent with respect to certain chains, by giving up availability with respect to those chains

(so, instead, CP "on demand"). For example, as a client I could wait for a confirmation from the Cosmos Hub, Ethereum, and Solana that some state transition on some other chain had been witnessed and included in such-and-such an order before considering it valid - for all users who do this, their reads are guaranteed to be consistent w.r.t. any other users who also do so (even if the other chain's sequencer/consensus etc. equivocates

), as long as the Cosmos Hub, Ethereum, and Solana chains refuse to accept any linearity violations in their locally verified history (to which they re-attest by signing blocks).

One difficulty of implementing this safely is dealing with recovery from a linearity violation elsewhere. Suppose that chain A signs two conflicting blocks and sends one to chain B, the other to chain C. Now, if chains B and C have each naively accepted the first block sent to them by A, they can no longer merge history from each other - in a permissionless context where anyone can create and merge history, this becomes a huge DoS vector and renders this design unworkable.

In order to safely merge history, we need to do a few things:

- Craft proofs in some fashion that makes merging history a constant-cost operation from the perspective of the recipient, regardless of the length of the history
- Separate out individual dependencies (instead of dealing only on a block level) so that Byzantine users or chains cannot block others' progress by manufacturing equivocations
- Specify how conflicts should be resolved

, such that B and C (perhaps with some other signature) can make progress after the equivocation event and remain consistent with each other

To satisfy these requirements, as a proposal for part of a standardised protocol architecture, Anoma introduces the concept of resource controllers

.

## Introduction

The main idea of resource controllers

is to allow for resources (including data and logic) to move across trust and concurrency domains, preserving appropriate integrity guarantees, with no additional work required by the application developer. Controllers establish an order of resolution between consensus providers involved in the history of a particular resource, such that conflicts later in the controller chain can be resolved by appeal to earlier controllers, all the way back to the original resource issuer if necessary. In the context of an execution logic capable of privacy-preserving verification (such as Taiga), controllers allow for the privacy-preserving properties to be retained across the domain boundary (i.e., private bridging). In the context of an execution logic capable of succinct verification (such as Taiga - both privacy-preserving and succinct), merging history is a constant-cost operation independent of the length of the history.

Controllers generalize the intuition behind cross-chain token denomination tracking in the IBC protocol (details), which is greatly simplified with a standardised resource model and identity type. Unlike chains in IBC, controllers in this standardised resource logic can provide global double spend detection (with garbage collection under a tunable synchrony assumption), and the controller model allow for recovery by the original issuer under exceptional circumstances (i.e. manually deciding to gain availability at the cost of consistency).

## Intuition

In this model, each resource has a DAG of controllers

, ending in a terminal node (last controller), where the signature of this last controller is required to execute a partial transaction. A controller is not necessarily the owner of a resource, but they are the current custodian, of whom a liveness assumption is made for interactions involving the resource in question.

The controller DAG can be appended to and reduced (from the end) at any time. Intuitively, appending controller A is a cross-domain transfer to A, and removing a controller at the end of the DAG is unwinding by one step the previous cross-domain transfer path. A DAG is used instead of a list in order to allow resources to have multiple ancestors from different controllers.

## Transaction execution logic changes

Controllers take the basic resource structure and extend it with a list of external identities:

data Resource = Resource { logic :: Set Resource -> Set Resource -> Extradata -> Bool, data_static :: ByteString, data_dynamic :: ByteString, quantity :: Integer, controllers :: TerminalDAG ExternalIdentity }

The controllers

field is also included in the static data (for the purposes of denomination derivation).

Note: Possibly, controllers

could just be put

in the static data, and this logic standardised at a slightly higher level.

The partial transaction validity function is also changed such that the following extra condition must hold:

- In order for the partial transaction to be considered valid, a valid signature from the last controller in the list over the partial transaction (or a block containing it) must be provided in extradata.

The resource logic should also include the following rule:

- A resource with controller DAG cs

may be spent to a copy of the same resource, but with any number of (arbitrary) controllers appended to the DAG. These controllers do not need to sign.

- A resource with controller DAG cs <> cs'

, where both cs

and cs'

are sub-DAGs of arbitrary length, may be spent to a resource with controller DAG cs

, iff. signatures over the partial transaction are provided by all controllers in cs'

. (i.e., any later sub-DAG which is strictly after the earlier counterpart sub-DAG may be pruned if all controllers in the later sub-DAG sign).

## Integrity assumptions

In order to provide integrity in this system, a controller c

must commit to not signing any two partial transactions which double-spend (reveal the same nullifiers). This does not require a total order, but it does require that controllers track the nullifiers in their signature history.

## Cross-domain usage flow

Let’s say that a user Alice wants to issue a resource, move it to a controller C

, and send it to Bob. Bob then transfers it to controller D

, and sends it to Charlie.

1. Alice creates the resource (ex nihilo), with herself as the initial controller.
2. Alice crafts a transaction spending this resource to a copy with C

as an additional controller and Bob as the owner. She signs it.

1. Alice relays this transaction to controller C

. C

performs the following steps (probably with the assistance of proofs provided by Alice):

a. Checks partial transaction validity (as above)

b. Checks valid resource history all the way back to the time of creation by Alice

c. Checks no-conflicts (that C hasn’t signed over any other partial transaction which revealed any nullifiers in this history)

d. Then, C

executes the transaction and appends the nullifiers to its known nullifier set

1. Bob crafts a transaction spending this resource to a copy with D

as an additional controller and Charlie as the owner. Bob relays this transaction first to controller C

, for a signature from C

. C

performs the checks as above, then signs it. Bob then relays this transaction, along with C

’s signature, to controller D

, which performs the same steps as C

above.

1. Charlie can now spend the resource (on controller D

).

Note: it may be prudent to explicitly separate the step of C

learning about Alice’s valid history of which C

was previously unaware, and perform this “eagerly” as opposed to “lazily”.

Note: if chains do not have compatible history formats, they can instead accept resources under a promise of computational integrity. Concern: application-dependent invariants.

At first glance this may seem expensive, since each recipient of a resource must verify the history since inception. This can be cut to a constant cost, however, with recursive ZKPs and incremental proof composition. In order to adhere to the integrity assumption, potential controllers must store a commitment to all nullifiers over which they have ever signed, but data availability (storage) for the actual nullifiers can be sharded (in which case the transaction author must fetch the nullifiers in question, and prove to the consensus provider into whose history they wish to merge that there are no conflicts).

Note: Partitioning the nullifier space is also possible (with some slight privacy loss).

## Partition recovery

A resource may allow that a signature from a controller earlier in the DAG, without

a signature from the last controller, is sufficient to execute a transaction (perhaps under certain additional circumstances, e.g. an attestation from other parties that the last controller is not live). This will mean that the resource in question will not be able to be sent back from the last controller to any parties who learn about the nullifier reveal of the initial (revocation) spend first, and that further history built on top of a potential partition spend will not be mergeable. Effectively, this constitutes an (intentional) choice of the controller to break their redemption promise for controllers later in the chain of a particular resource. Consistency is lost (we’re opting to give up consistency for availability here), so controllers should be extremely careful about doing this.

## Private bridging

Assuming succinct recursive ZKPs, private bridging emerges naturally here merely through the use of a privacy-preserving transaction execution logic and nullifier-based double spend prevention. Note that - unlike IBC - in a fully privacy-preserving world, what balances of which assets were held with which controllers at any given point in logical time would be not in general public information. Some further research to determine security implications of this would be prudent. Auditing options such as keeping track of token movements with homomorphically encrypted counters and periodically threshold-decrypting should be reasonably straightforward (this may be useful, for example, to check that the amount of stake securing a particular domain is sufficient).