


Anoma State Architecture

Isaac Sheff ^a

^aHeliix AG

* E-Mail: isaac@heliix.dev

Abstract

Each anoma instance maintains a state machine, which ultimately implements the Resource Machine. This is the state of that state machine. This state limits what can be done in post-ordering execution.

Keywords: State machine ; Storage ; Execution ; Resource Machine

(Received: 2 December 2024; Version: December 3, 2024)

Contents

1	Introduction	2
2	Fitting in the Execution Engine	2
3	Current Time	3
4	State Roots	4
4.1	Using State Roots Outside of Post-Ordering Execution	4
4.2	Using State Roots During Post-Ordering Execution	4
5	Commitments	5
5.1	Commitment Keys	5
5.1.1	Using Commitment Keys During Post-Ordering Execution	6
5.1.2	Using Commitment Keys Outside of Post-Ordering Execution	6
5.2	Commitment Inclusion Root Keys	6
5.2.1	Using Commitment Inclusion Root Keys During Post-Ordering Execution	6
5.2.2	Using Commitment Inclusion Roots Outside Post-Ordering Execution	7
6	Unstructured Nullifiers	7
6.1	Using Nullifiers in Post-Ordering Execution	8
6.2	Using Nullifiers Outside of Post-Ordering Execution	8

7	Blob Storage	8
7.1	Blob Labels	8
7.2	Blob Structure	9
7.2.1	Logic Hash	9
7.2.2	Blob Label	9
7.2.3	Deletion Criterion	9
7.2.4	Commitments	10
7.2.5	Structured Nullifiers	10
7.2.6	Data	10
7.3	Using Blobs in Post-Ordering Execution	11
7.4	Using Blobs Outside Post-Ordering Execution	11
8	Endorsement Map	11
8.1	Using the Endorsement Map in Post-Ordering Execution	12
8.2	Using the Endorsement Map Outside Post-Ordering Execution	12
9	Send and Receive Records	12
10	Concluding remarks	12
	References	12

1. Introduction

Each Anoma instance maintains a replicated state machine [Sch86], which updates using serializable transactions [Pap79], as defined in the Anoma Resource Machine [KG24], acting as a controller [Isa24]. Here we define the state of this replicated state machine. This state can be read, written, and updated atomically in post-ordering execution of Transaction Candidates [KG24]. Other state that validators store, but do not necessarily update in sync, (such as network-level key-value stores) is not considered here.

2. Fitting in the Execution Engine

The Anoma state machine should be able to run in Typhon’s execution engine, which updates state using serializable transaction candidates, and assumes state can be expressed as key-value pairs. It tries to run transaction candidates that don’t touch the same keys in parallel, when possible. Transaction candidates therefore specify in advance sets of keys they may read and write. We say that a transaction candidate has a **lock** on keys its label specifies: it can have a **read lock**, a **write lock**, or both. To facilitate these read and write locks over complex data structures, we allow transaction candidates to lock *ranges*. For example, blob storage by logic hash and label *prefix*, meaning

they can *in principle* lock infinite keys. There should be fees for read and write locks, but I don't know how to set them yet.

Note that it is difficult to store the “same” piece of state under multiple keys: two transaction that try to access the same state would not appear to conflict if you only looked at the keys they touch, when they do in fact conflict. Similarly, it is difficult lookup the “same” pieces of state using ranges in different dimensions (e.g. “all cars made between 1960 and 1970” and “all cars made north of Cairo and south of Berlin”). This is because it is difficult to tell if two transactions could conflict.¹ For this reason, we do not allow multiple keys to look up the same datum, and we specify at most one dimension in which range queries may be used for each sub-type of key.

Sharding. The execution engine's assignment of keys to storage shards, referred to as **sharding strategy**, should take these ranges into account. If a transaction might query “all cars made between 1960 and 1970,” then it's useful for that data to be on the same shard (or some small number of shards), rather than spread across all of them (spreading them out means more messaging to more processes). The easiest way to accomplish this is to store key-value pairs sequentially (according to whatever ordering is used for range queries), leaving buffer space on each shard, and then re-shard (re-distribute which key ranges go on which shard) before any one shard gets too full.

Although Typhon's state machine execution engine does not distinguish between different “types” of lookup keys, we designate sub-types useful for each part of state. The “lookup key type” for the state machine as a whole is thus a sum type (a.k.a. a coproduct or *tagged union*) of all of these.

Hereafter, we specify the types of key, value pairs that make up the state. Major components include:

1. **Key Type:** the type of the lookup key in the key, value pair
2. **Value Type:** the type stored under the lookup key in the key, value pair
3. **Range Queries:** how any kind of “range of keys” is specified (there can only be one dimension)

Other details including useful invariants or examples may be provided.

3. Current Time

We can make “current time” part of the state. This is optional: it is not necessary for the anoma resource machine.

¹It is not impossible to design an execution engine which can handle this [EWS12], but it is much more complicated.

Key Type. Unit. There is only one key.

Value Type. Timestamp. The “current time” in the state machine.

Range Queries. None.

This would only be updated by specific transactions, for example, once every consensus. Invariants would include correspondence to the timestamps used in state root and commitment root storage.

4. State Roots

A **state root** is a deterministic digest of the entire state of the state machine (including everything below). Ideally, this would facilitate zk-friendly inclusion proofs for each of the other elements of state (i.e. this commitment is found under this state root, or this nullifier accumulator is under this state root, or this blob is under this state root). State roots can be, for example, a Merkle root. However, we do not specify here exactly which digest to use. We generally assume that state roots are small, like a hash.

Some transactions may want to query for state roots at various times in the past, possibly to prove that a specific state root on which they base some proof is valid.

Key Type. timestamp. We’ll have to choose what kind of timestamp we used internally, and how, if at all, it’s related to wall-clock time.

Value Type. state root. The exact nature of this depends on what kind of state commitment structure we’re using.

Range Queries. Time range. specified with start and end time.

These state roots may be part of fancy merkle structures, but queries using this kind of key should be allowed regardless of the fancy structure.

4.1. Using State Roots Outside of Post-Ordering Execution

Users, solvers, and other controllers may want to retrieve a *recent* state root for an Anoma instance. This does not necessarily need to be the *current* state root, as instances may update fast. These state roots can be used in proofs about state at that time (e.g. this blob was indeed stored at that time).

4.2. Using State Roots During Post-Ordering Execution

Any transaction can read recent state roots, and even identify the *most recent* state root, during post-ordering execution. This might be used in specific proofs, perhaps in order to satisfy a resource logic.

Only very specific transaction candidates are allowed to store new state roots or delete old ones. Specifically, these transaction candidates must generate the state root correctly. They are not “normal” transaction candidates

submitted by users. They might be scheduled regularly (say, with every consensus).

Aside. An alternative would be to use the state root itself as a key (and store only a unit), so the only query would be “was this state root valid at some time in the past.” This does not allow any interesting kind of range query, and it’s hard to allow both kinds of queries in a fully-safe way. Unfortunately, doing both requires storing the “same state root” under 2 keys: both the state root itself and the timestamp.

5. Commitments

A **Commitment** is a hash representing a resource that has been created. Since users may want to be able to prove that a given resource has been created without specifying which, we will want a cryptographic accumulator for commitments with zk-friendly inclusion proofs. Possible cryptographic accumulators include Merkle Mountain Belts, which have some nifty properties including constant computational complexity to add a new commitment to the accumulator, and shorter proofs for more recently added commitments.

Since these inclusion proofs relate to larger Merkle structures, we may want some additional storage for that, or perhaps it will fit into blob storage.

Regardless of specific cryptographic accumulator, we need 2 sub-types of keys: Commitment Keys, and Commitment Inclusion Root Keys.

5.1. Commitment Keys

Since rapid transactions may want to use resources from recent (even the previous) transactions, we want to be able to check for / somehow “use” recent commitments. This would require the ability to create (not necessarily zero knowledge) proofs for recent commitments in post-ordering execution. If the proving environment can simply look up recent commitments, this becomes relatively easy.

Key Type. Commitment, as defined in the resource machine report.

Value Type. For each commitment key, we store either *unit* (representing that this commitment was made recently), or an inclusion proof for a recent inclusion proof root, and the timestamp for that root.

Range Queries. None.

Aside. We could assume that all inclusion proofs are grounded in a state root, and so looking up a separate inclusion proof structure is unnecessary. For now, we do not assume this.

5.1.1. Using Commitment Keys During Post-Ordering Execution

Some notable things a Transaction Candidate can do using these keys:

1. Add a commitment (after verifying all the resource logic proofs for an ARM Transaction).
2. Prove the existence of a recent commitment (as part of an ARM Logic Proof)

Transactions that don't know what commitment they want to look up in advance cannot get a read lock on a commitment key. These may still be able to prove the existence of a recent commitment if it's included in a blob (Section 7).

5.1.2. Using Commitment Keys Outside of Post-Ordering Execution

Users will want to be able to read an inclusion proof for a recent commitment. They can issue read-only transactions to validators to read this, or that information can be disseminated by other means.

We generally assume that users and solvers, without communicating with controllers, can generate a resource and its commitment and appropriate proofs (ZK or otherwise), given whatever the resource logic requires.

5.2. Commitment Inclusion Root Keys

Key Type. timestamp

Value Type. inclusion proof root. The exact type of this depends on the cryptographic accumulator used for commitments. These may be part of some larger Merkle structure, but we still need to be able to look them up using these keys.

Range Queries. timestamp range

5.2.1. Using Commitment Inclusion Root Keys During Post-Ordering Execution

The most important use of Commitment Inclusion Root Keys is as part of verifying resource logic proofs: looking up a commitment inclusion root proves the commitment inclusion root is valid (for some time in the past). Transaction Candidates Can also look up specifically the *most recent* commitment inclusion root (by querying an open time interval), if that is useful.

Only very specific transaction candidates should be able to update commitment inclusion roots, or write commitment inclusion proofs. Periodically (perhaps every consensus), the anoma state machine should have a transaction candidate that creates a new inclusion root, and updates all commitment

key-value pairs to use the new inclusion root. This allows users to read recent inclusion proofs for their commitments at any time.

To save on storage, we may want to delete some of the data necessary to generate an inclusion proof after a certain period, while retaining the data required to check such a proof. This also entails that we store whatever is necessary to check arbitrarily-old inclusion proofs (e.g. old merkle roots). This means that there will be multiple “phases” after generating a commitment: for a while, some kind of proof is available from the validators, and after that time has elapsed, it is the responsibility of whomever wants to use this resource in the future to remember the relevant proof. Some commitment schemes require the anyone storing generated proofs to occasionally update their proofs with on-chain information.

5.2.2. Using Commitment Inclusion Roots Outside Post-Ordering Execution

Users and solvers may will want to query recent (although not necessarily the *most* recent) commitment inclusion roots from controllers. They can do this with read-only queries. Other mechanisms can disseminate this information further.

We generally assume users and solvers can verify a commitment inclusion proof against a commitment Inclusion Root.

6. Unstructured Nullifiers

Each time a resource is consumed, we produce a nullifier. Most of the time, we assume this nullifier has no structure: queries for two nullifiers only disclose if the two are exactly equal, and nothing more. A nullifier query needs to check if a nullifier has been issued or not. There are no range queries. As a consequence, even a transparent resource specifying an unstructured nullifier cannot be nullified unless the transaction candidate knows the nullifier it’s going to use in advance. We discuss structured nullifiers for rapidly-mutating transparent resources in [Section 7.2.5](#).

Key Type. nullifier, as defined in the resource machine report.

Value Type. storing a *unit* represents that this nullifier has officially been produced.

Range Queries. None.

Aside. to avoid storing all nullifiers forever, someday we may do something more complicated. Since transactions will want to prove that a given resource has *not* yet been consumed, we would want a cryptographic accumulator for nullifiers with *non-inclusion* proofs. We have a few different ideas on how to save on storage for these.

6.1. Using Nullifiers in Post-Ordering Execution

As part of checking an ARM Transaction's validity, we query for all unstructured nullifiers emitted, in order to prove they are not yet present. If the ARM Transaction is valid, we store *unit* at keys corresponding to all nullifiers emitted.

6.2. Using Nullifiers Outside of Post-Ordering Execution

Users and Solvers may want to query controllers to see if specific nullifiers have yet been issued. This represents checking if some resource has been consumed. Note that such queries can leak correlation information. Read-only transactions work for this purpose, especially if the querier wants maximally up-to-date information. Note that such a query doesn't guarantee the nullifier will still be unissued by the time any particular transaction candidate is executed.

We generally assume that users and solvers can generate a nullifier and appropriate proofs (ZK or otherwise) from a resource (given whatever the resource requires).

7. Blob Storage

Blob storage in the state machine is used for structured queries of data. Blobs can include, but do not have to include, resources. Queries and in-state-machine storage are expensive, so data that is fully determined before post-ordering execution (which would include anything that can be fetched by hash) is probably better included in the Transaction Candidate itself than fetched in post-execution runtime.

Key Type. tuple of:

1. Resource logic hash
2. List of sub-labels (hashes)
3. Hash of the blob itself

Range Query. logic hash with label *prefix*: represents all elements with labels that have that prefix.

Value Type. Blob (see [Section 7.2](#))

7.1. Blob Labels

Blob keys have a forest structure: the root of each key is a resource logic hash that governs what blobs may be created in this tree. In order to write a blob, it must be part of the *appdata* field of the proof for a resource logic matching

this hash. This means that the resources used in a transaction govern what blobs can be written.

The branches of the forest structure are arbitrary lists of hashes, which can in turn be governed by the blob logic. Each blob features a **label**: list of such hashes, or **sub-labels**. For example, we could envision an application that stores records about animals, where each blob's label is a full taxonomic classification (e.g. "Eukaryota / Animalia / Chordata / Mammalia / Carnifora / Feliformia / Felidae / Felinae / Felis / Catus").

Finally, leaves are hashes of specific blobs, ensuring that blobs are uniquely identified with their keys.

This allows blobs to represent mutable state, each mutable object is identified by a label prefix, and each state update appends a timestamp to that prefix to store the new state. The "current state" of label prefix P would be the blob with the label of the form $P : t$ with the greatest timestamp t .

We do not specify here how the execution engine shards or indexes this forest. It will need to maintain an index structure, but this is not part of the anoma state machine itself.

7.2. Blob Structure

Blobs contain several elements, and can be used for several different purposes.

7.2.1. Logic Hash

This resource logic governs what blobs can be written with this hash. Only transactions involving a resource with this logic can write this blob. In particular, the blob must be part of the appdata for such a resource's logic proof. In this way, resource logics govern blob permissions and structure. Different resources may have completely different ways of using the sub-labels structure, allowing for different kinds of queries.

7.2.2. Blob Label

Each blob features a complete label. A blob label is a list of sub-labels, each of which is a hash. These are used in organizing blobs into larger (tree) data structures, from which transaction candidates can query sub-trees.

7.2.3. Deletion Criterion

Each blob features a deletion criterion (which is a predicate). A transaction candidate must prove this criterion is satisfied in order to delete a blob. For now, we probably want to restrict the allowed criteria to a few cases, such as:

1. current "block height" is at least X
2. current "time" is at least X
3. a signature for some specific message X verified using public key Y

4. nullifier X has been committed

7.2.4. Commitments

Blobs contain a (possibly empty) set of commitments. Such storage is only allowed if the commitments are also stored (either in the same or some prior transaction) in the commitment state. Thus, blob reads *can be used as proof of a commitment* in resource logics. This allows post-ordering execution to construct ARM transactions that rely on prior commits without knowing exactly which prior commits they need in advance.

7.2.5. Structured Nullifiers

Structured Nullifiers are a new concept, extending the nullifiers as defined in the Anoma Resource Machine Report [KG24].

Blobs contain a (possibly empty) set of nullifiers. Each resource can specify whether it can be nullified with an *unstructured nullifier*, as outlined in ??, or a structured nullifier with a specific blob label prefix. An ARM transaction nullifying any structured nullifier resource must store a blob (with its own resource logic, and a label with the specified prefix) with the nullifier in this field. An ARM transaction requiring a proof that a structured nullifier resource is not yet nullified must query all blobs with the specified label prefix and check.

The reason we might want structured nullifiers is kind of subtle. Transaction candidates must list everything they could read in their labels before post-ordering execution begins. This means that they can't check if a nullifier has been committed unless they know what that nullifier is in advance. For some post-ordering execution applications (using transparent resources), this may be unnecessarily restrictive. For example, a "king of the hill" application might have a resource representing current ownership of the "king" title, but it's impossible to consume this resource (and therefore impossible to transfer ownership) unless you know who the king is before post-ordering execution starts. If this kingship is transferred frequently, this may be a problem.

Structured nullifiers allow transaction candidates to lock a label prefix known in advance, without knowing the precise *nullifier* in advance. This would allow a "king of the hill" label that would force all "king of the hill" transaction candidates to be executed serially (which is desirable) while allowing other transaction candidates to execute concurrently. Such nullifier labels are terrible for unlinkability: they're really only useful for unshielded resources.

7.2.6. Data

Blobs contain an arbitrary (possibly empty) bytestring called *data*. This can contain, for example, a resource.

7.3. Using Blobs in Post-Ordering Execution

A transaction candidate should be able to look up a blob (given a Key), or look up a set of blobs, given a resource logic hash and a blob label prefix.

Given a resource logic hash and a blob label prefix p , a transaction candidate can also query “for which sub-labels s are there any blobs with the prefix $p : s$?” This has requires exactly the same transaction candidate label as a query for all the blobs with the resource logic hash and blob label prefix, but might let transaction candidates use trees of data without having to fetch the whole tree from storage.

Transaction candidates can also delete a blob, given a proof that its deletion criterion has been met.

When verifying resource logics, a commitment from a blob in state can be used as proof a commitment was correctly created. Likewise, the non-existence of a structured nullifier in any blob with a given prefix can be used as a proof that a structured-nullifier resource with that prefix was not nullified.

An ARM Transaction can store a blob, provided:

1. There is a resource with the logic hash of this blob in the ARM Transaction, and this blob is in the *appdata* for that resource logic proof.
2. All commitments in the blob are also stored (or have been previously stored) in commitment state.
3. All structured nullifiers in the blob correspond to resources nullified in this ARM transaction, with this blob’s resource logic and specifying structured nullifiers with a prefix of this blob’s label.

Additionally, if an ARM Transaction nullifies a resource that uses structured nullifiers, it must write a blob containing the nullifier for that resource.

7.4. Using Blobs Outside Post-Ordering Execution

Solvers and users may want to query for blobs in storage, which they can do using read-only transactions.

It should also be possible to prove that a blob was in storage as of a certain state root in some compact way, such as a Merkle proof.

8. Endorsement Map

For cross-instance communication, Anoma chains can maintain light clients of other chains [Isa24]. This might, in principle, be encoded as resources in blobs, but I will talk about it here like it’s a separate thing, in case that doesn’t work out.

The Endorsement Map is a map from ChainIDs to StateRoots.

Key Type. ChainID

Value Type. State Root (technically, not every controller has to have the same type of state root)

Range Queries. None.

8.1. Using the Endorsement Map in Post-Ordering Execution

A specific type of transaction candidate should be able to update the state root for a chain: this requires a proof that the new StateRoot represents a state that is the result of a valid sequence of transactions starting with the state represented by the old StateRoot.

8.2. Using the Endorsement Map Outside Post-Ordering Execution

Given 2 state roots from a correct (not-forked) chain, validators of that chain should be able to generate a proof that one is the result of a sequence of valid transactions starting with the other. Other controllers (and users and solvers) should be able to read and validate this proof.

9. Send and Receive Records

For sending resources between instances, anoma maintains records [Isa24]. These may be representable as resources, but we'll talk about them here as if they're not.

Here we require a collection of mutable send records and receive records, each with a unique (immutable) ID, assigned at creation.

The exact structure of these has not yet been worked out, and remains future work.

10. Concluding remarks

We have outlined the key/value structure of the Anoma State Machine's state. It is designed to be compliant with the Typhon execution engine's requirements, allowing concurrent transaction processing. Of particular note are our use of Blob storage for structured lookups that can be used in Resource Logic proofs: storing commitments and structured nullifiers (Section 7.2.5).

References

- EWS12. Robert Escriva, Bernard Wong, and Emin Gün Sirer. Hyperdex: a distributed, searchable key-value store. 42(4), 2012. URL: <https://www.cs.cornell.edu/people/egs/papers/hyperdex-sigcomm.pdf>, doi:10.1145/2377677.2377681. (cit. on p. 3.)
- Isa24. Sheff Isaac. Cross-Chain Integrity with Controller Labels and Endorsement. *Anoma Research Topics*, Jun 2024. URL: <https://doi.org/10.5281/zenodo.10498996>, doi:10.5281/zenodo.10498997. (cit. on pp. 2, 11, and 12.)

- KG24. Yulia Khalniyazova and Christopher Goes. Anoma Resource Machine Specification. *Anoma Research Topics*, Jun 2024. URL: <https://doi.org/10.5281/zenodo.10498990>, doi:10.5281/zenodo.10689620. (cit. on pp. 2 and 10.)
- Pap79. Christos H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, October 1979. doi:10.1145/322154.322158. (cit. on p. 2.)
- Sch86. Fred B. Schneider. The state machine approach: A tutorial. Technical report, USA, 1986. URL: <https://www.cs.cornell.edu/fbs/publications/ibmFault.sm.pdf>. (cit. on p. 2.)