Anoma Research Topics | TECHNICAL REPORT

Cross-Chain Integrity with Controller Labels and Endorsement

Isaac Sheff [a]

[a]Heliax AG

* *E-Mail: isaac@heliax.dev*

Abstract

In distributed systems, mutable digital objects typically require some state machine to decide on their definitive current state. This state machine can be replicated to enhance availability and fault tolerance. We call the authoritative state machine of a digital object its *controller*. Typical examples of controllers defining objects include a database storing a record, or a blockchain storing the current state of a smart contract. Without some kind of controller, different parties may have contradictory notions of what the state is, and no way to reconcile them. In a distributed system, some controllers may be *Byzantine*, and make duplicitous or incoherent statements about state.

Here we design rules and procedures for a multi-state-machine ecosystem, featuring digital objects, or *resources*, with application-defined state-dependent rules for how they can be updated. Each controller can express an authoritative state, including authoritative resource states. Each resource is also labeled with a controller identifier, whose state is definitive for this resource. Resources can transfer between controllers, and updates can depend on multiple resources, so resource labels also express a *dependency graph* detailing which controllers, if they were Byzantine, may have corrupted this resource. In a sense, these labels represent a distributed *taint tracking* or *dynamic information flow control* solution. One challenge is avoiding size explosion in this dependency graph: we enable removing unnecessary parts of history when, say, a resource transfers from $\boldsymbol{A}$ to $\boldsymbol{B}$ and back to $\boldsymbol{A}$ again. In information flow control terms, these operations require *endorsement*. Our resource controller operations generalize a number of techniques used in blockchain settings. We define rules and procedures for creating, updating, transferring, and tracking the state of labeled resources, and prove that our rules maintain safety properties including *causal resource history* and *consistent controller labels*.

(Received: April 16, 2024; Version: Jun 25, 2024)

1. Introduction

Distributed platform interoperability is crucial to improve cost, usability, adoption, and even security. When applications must share a state machine (or, more generally, *controller*) to interact, there is an incentive to push all applications onto one controller trustworthy enough (and with enough throughput) for everyone. Attempts to create such a controller typically require trust in a single authority (e.g., AWS [Ama24]) or an extremely expensive global consensus (e.g., Ethereum [But13]) and remain inadequate for some applications: JP Morgan does not trust Ethereum to control their accounts [JP 24]. In fact, it is unlikely that *all* worthwhile applications will ever agree on a

controller who can manage all of their state. This is why interoperability is so important: the internet works not because we all trust some single authority to manage all of it but because many different applications in different trust domains can interact.

Nevertheless, cross-domain data integrity remains a challenge. While existing systems can track controllers that may have affected each datum [LAG+17, WGDW22], these can easily lead to a state explosion: each object's label features an ever-growing set of controllers that may have affected it. In blockchain ecosystems, some protocols cleverly get around this: objects can reduce their labelling burden when one controller *endorses* another, reviewing (some part of) its history and, crucially, pledging not to endorse a contradictory history. For example, blockchains can create *wrapped* tokens with IBC and ICS20 [Sco23, Goe20, ics24]. A wrapper represents a controller that had previously controlled the token, and nested wrappers represent a list of controllers. A controller can *unwrap* a token with a simple but effective review of the wrapped token history. It checks a crucial invariant: it will unwrap no more tokens of each type from each destination than it wrapped. In this work, we generalize this controller history tracking and endorsement approach, and enable fully general transferable digital objects (not only tokens), with arbitrary transactions. Crucially, this means generalizing resource histories from a list of wrapping controllers to a DAG. This endorsement approach differs somewhat from some Information Flow Control approaches [LAG+17, CMA17], in that we assume controllers can review and endorse entire execution traces of other controllers, and check for contradictions.

Here, we introduce a novel protocol for controllers that enables very general operations across state machines with different trust domains. Our controller protocol will eventually be part of a larger unified cross-domain architecture, with standards for each state machine, as well as transferable objects called *resources* [KG24]. We detail controller operations that allow our architecture to generalize many existing techniques (including many cross-chain and side-chain operations).

1.1. Controllers

Controllers are a key component of any transaction processing state machine, including blockchains [AGMS18, Sch90]. We define the controller as the component that *orders*: it decides on an ever-growing sequence of transactions defining the execution *trace*, and thus the current state, of a state machine. Controllers do not necessarily compute and store this state themselves, although it may be efficient to do so. Committing to an ever-growing sequence of transactions, however, does require that controllers keep *some* state, to ensure they do not *fork*: commit two contradictory traces (neither

is a prefix of the other). Forks are the essence of, for example, double-spend attacks [AM17].

Trusting controllers is fundamental for digital objects: tables in Postgres maintain their invariants iff the database is working properly [Pos24]. Similarly, a smart contract on Ethereum is consistent iff the Ethereum consensus is working properly[But13]. We categorize controllers in terms of safety and liveness:

- *Safe* controllers commit transactions only in a totally ordered sequence (called a *trace*): they do not fork. The *state* of a valid controller is unique and defined as the result serially applying the transactions (atomic transitions defined by the state machine) in the trace committed, and ignoring any *invalid* transactions (as defined by the state machine). *Unsafe* controllers are also called *malicious* or *Byzantine*.

- *Live* controllers eventually respond to valid queries, and append valid transactions to their trace. Controllers that are not live are called *unlive* or *crash-prone*.

In general, we assume all unsafe controllers are unlive: controllers that don't follow the specification could ignore all queries.

1.2. Resources

Our architecture tracks specific types of transferable digital objects, which we call *resources* [KG24]. We can encode extremely general mutable state with resources, but resources themselves are fairly simple. Resources have very limited mutable state: they are *not yet created* by default, can transition to *created*, and then to *consumed*. However, each resource can carry arbitrary immutable data: the identity of the resource specifies this data (the id could be a hash). Resources transition between these states in transactions ordered by controllers. Each resource therefore specifies a single controller that can order transactions for each type of transition, ensuring there is a single authority in charge of deciding whether each transition has or has not occurred. Transactions which perform a state transition but are ordered by the wrong controller are *invalid*.

Each controller's state carries cryptographic accumulators (e.g., Merkle roots [Mer88]) representing the set of resources created by transactions ordered by this controller and the set of resources consumed. If a resource is neither created nor consumed, it is *not yet created*. As part of their immutable data, resources can have complex proof obligations (which we call *resource logics*) determining when they can be created or consumed, and these may depend on the state of other resources. A resource logic can for example, specify exactly what programs can consume this resource: it would require

a proof that the resources created are precisely the outputs of running a particular program with the consumed resources as inputs. Such a proof might be as simple as a full execution trace of the program, or as complex as a zero-knowledge proof [KST22]. Through these logics, resources can encode fairly arbitrary state, not limited to scalar registers or tokens, while still allowing ZKP-style confidential transactions [KG24].

1.3. Transactions

Transactions are atomic state transitions [AGMS18, Sch90]. For our purposes, transactions designate a set of resources (which must be *created*) as *inputs*, *consume* some subset of their inputs, and *create* some *output* resources. In general, we assume these transactions are deterministic, so each new state is uniquely defined. Transactions can only update state controlled by one controller, and must include checkable proofs that the relevant resource logics of each resource created or consumed are satisfied. However, input resources may have been *created* in a transaction on another controller. Therefore, controllers can sync with one another, allowing transactions to check if resources on other controllers have been created. These updates can be asynchronous, so it is possible a transaction will not immediately be able to prove that a resource has been created.

1.4. Labels

Resources themselves carry *labels* concerning controllers who can or have affected the history (or ancestry) of that resource. We will detail exactly what these labels will be later, but they include, among other things, a *creating* controller, whose state defines whether this resource has transitioned from *not yet created* to *created*, and a *terminal* controller, whose state defines whether this resource is *consumed*. Any transaction with this resource as an input must be ordered by its terminal controller. This ensures there are never two controllers trying to consume the same resource with different transactions. For example, imagine a resource representing a token, created on some controller A. If controller B orders a transaction that consumes this token in order to create a new token called *Alice*, and controller C orders a transaction that consumes this token in order to create a new token called *Bob*, then we have a "double-consume:" together, *Alice* and *Bob*'s histories represent the token being consumed twice, which should not happen. This is clearly a problem if the tokens have, say, monetary value: it's a double-spend [Nak08]. Terminal controllers solve the problem: if the resource specifies it can only be used by transactions on B, then the token can only be double-spent if B forks.

We might imagine labels which include a set of *affecting* controllers, who have influenced this resource's history (or provenance). In general, we can

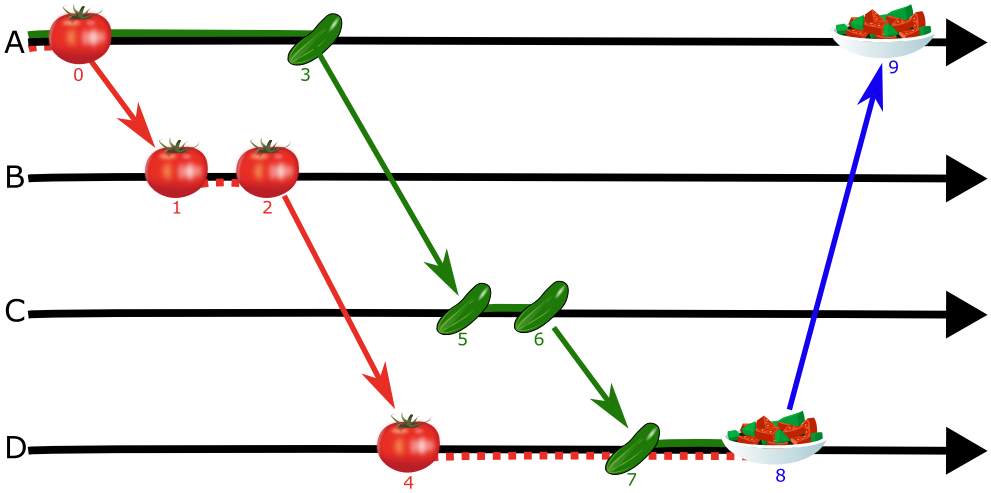


Figure 1. Timelines for 4 controllers, and resources for a virtual cooking application. Controllers are labeled with letters, and resources are labeled with numbers.

"transfer" a resource from one controller to another: we consume a resource with one terminal controller, and produce a similar resource with a different terminal controller, and the old resource's terminal controller in its affecting controllers (encoded in its label).

1.5. Salad Example

Suppose four controllers (A, B, C, and D) order transactions for a virtual cooking application (shown in Fig. 1). In the beginning, a tomato resource (0) and a cucumber resource (3) are on controller A. The tomato resource transfers to B (resource 0 is consumed, requiring whatever proofs are required to move a tomato, and resource 1 is created), and then, after some B-only transactions, it transfers to D (resource 4). Likewise, the cucumber resource transfers to C and then to D (resource 7). On D, a transaction consumes both the tomato and cucumber resources to create a salad resource (resource 8). At this point, the salad resource's history depends on A, B, C, and D. These are the affecting controllers of the salad's label. If any of these controllers have been unsafe (as in Fig. 2), there may be other resources elsewhere, claiming to represent the same tomato or cucumber that are supposedly part of this salad. The salad then transfers from D to A, where the tomato and cucumber began. Eventually, we want to allow A to *endorse* the salad's history, pledging not to endorse any alternative histories in which the cucumber or tomato did anything else, and removing the need to remember that B, C, or D could have "tainted" the salad's history [WGDW22]. They can be removed from the affecting controllers of the salad's label.

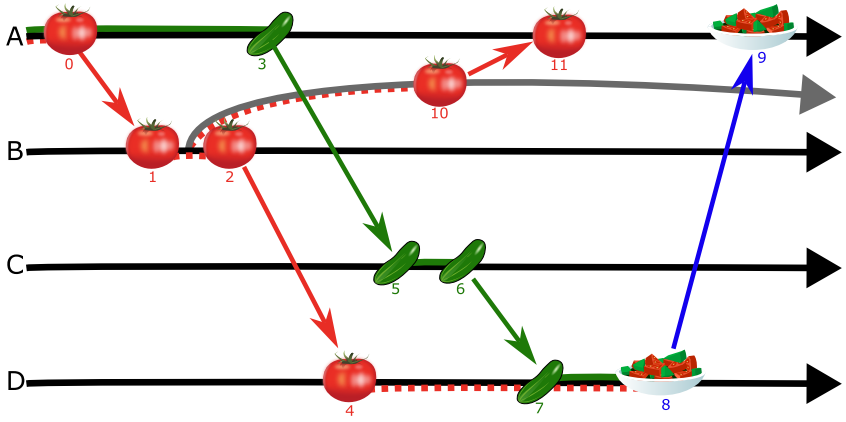


Figure 2. Controller *B* has forked into two timelines (gray and black). Resources 10 and 11 *conflict* with resources 2, 4, 8, and 9.

2. Desiderata

There are several properties we want our controllers and resource labels to have. As a running example, consider Fig. 2, a version of our salad example in which controller *B* has forked into two traces: gray and black. Resource 10 (and therefore 11) has a *conflicting* history with resource 2 (and therefore, 4, 8, and 9).

2.1. Causal Resource History (CRH)

We begin with a weak but relatively simple to maintain property: causally consistent resource history, or *Causal Resource History (CRH)* [LFKA11]. A resource with CRH was created in a valid[1] transaction, whose inputs were resources with CRH. This property is relatively easy to prove with recursive zero knowledge proofs (compared with some stricter properties we'll get to later) [KST22]. In general, in order to maintain CRH, a resource should carry a (recursive) proof that it was created in a valid transaction, the inputs of which had CRH proofs. In fact, CRH can be implemented in resource logics.

This property is not perfect: it still permits histories we might want to call *inconsistent*, including double-consumes. For example, in Fig. 2, any future resource that depends on resources 9 and 11 (so, something constructed by consuming the salad and the tomato), would have a history in which the same tomato (which started as resource 0), is consumed twice, but would still have CRH.

2.2. Serializable Resource History (SRH)

Here we define what we consider *consistent*: each resource (individually) reflects some fully serializable history featuring only correct atomic transactions (and no forks) [Pap79]. Equivalently, in the resource's history, all

[1] [KG24] defines a "valid" transaction as not using an input that's already been consumed. Here, as we lack global time, we require only that a transaction's inputs have not been consumed by a transaction in its *causal* past.

transactions are valid, and no resource is used as an input for a transaction after it has been consumed. In our Fig. 2 example, resources 11 and 9 individually have SRH, but they cannot both appear in the history of any future resource with SRH (no one can consume both as an input in some transaction). Alternatively, you could not combine money from both sides of the double-spend for one big purchase. SRH can be easily generalized to a group of resources by imagining a new resource that depends on all the resources in the group, and checking if it has SRH.

Maintaining SRH requires some kind of mechanism for checking if two resources can both be part of some future resource's history. Effectively, any such mechanism checks for forks, and can be used to filter for resources that have SRH.

In Appendix A, we detail a technique for maintaining SRH for all resources, but it can require very expensive computation and requires each resource to carry an ever-growing set of controllers in its label. Instead, we allow users to introduce a little trust into the system, and dramatically improve day-to-day operations. We will detail techniques for checking SRH while using our technique, but each check can require a lot of information.

2.3. Consistent Controller Labels (CCL)

The *Consistent Controller Labels* property (CCL) requires that if some resource r does not have SRH, then it has an unsafe controller in the *affecting controllers* set in its label. With CRH, if a user *trusts* that a resource's affecting controllers are all safe, they can be sure the resource has SRH.

One easy way to maintain CCL would be to start with a system that maintains CRH, and label every resource with every controller involved in its history (affecting controllers). If all the controllers are safe, then the resource has SRH. For example, in Fig. 2, resource 8's affecting controllers include A, B, C, and D. However, we explore optimizations that allow removing unnecessary controllers from the affecting controllers set. For instance, in Fig. 2, A can *endorse* the history of resource 9, and remove B, C, and D from its affecting controllers. The key in this case is to ensure that any future resource that depends on both resource 9 and resource 11 *will* include B in its affecting controllers. To accomplish this, we prevent A from endorsing (and thus removing B from) the history of both resource 9 and resource 11.

In Appendix B, we introduce Consistent Controller State (CCS), an even stronger property than SRH, along with a technique to maintain it, but we believe the inherent "forks split the world" drawbacks make CCS too strong to be worth using.

Field	Type	Description
halted	boolean	Is this a halting state? There are no valid transitions from halting states.
endorsement	map: controller id → state root	state roots of other controllers this controller has (non-recursively) endorsed.
sends	set of *send*	represents outgoing resource transfers
receives	set of *receives*	represents incoming resource transfers

Table 1. Fields in a controller's state.

3. Controller DAGs with Endorsement: a Technique for CCL

In this approach, we maintain CRH and CCL, with relatively little overhead. Resource label size can be kept proportional to the number of cross-controller resource transfers in its history (in the worst case), but in the best case is much smaller. Moving a resource between controllers is relatively cheap, requiring only few simple checks and a single endorsement (which can be done as two recursive ZKP checks). Endorsements are not resource-specific, so they can be done opportunistically and their costs amortized.

3.1. Controller State

We assume each controller has a unique *controller id*. We also assume that each controller's entire state (including accumulators for which resources it has created and consumed, and everything else) can be uniquely identified with a digest or hash called a *state root*. We have a notion of one state (or state root) being provably *after* another state (or state root) if the "later" one is the result of a (possibly empty) sequence of valid state transitions (transactions) ordered by the state root's controller, starting with the "earlier" one. We can define provably *before* similarly.

Each controller's state includes (but is not limited to) the data in Table 1.

The *endorsement* map starts with every controller id mapping to genesis (whatever state root it must start with; this may be defined in the controller id itself). There is one exception: the element for this controller's id is the state root for this state. We also introduce a new type of transaction. A controller id's endorsement element can be updated, given:

- a proof that the new state root is provably after the old state root, and
- a proof that if the endorsed state root contains an endorsement for this controller, it endorsed a state root provably before this state (you can't endorse someone who has endorsed a fork of yourself).

Note that this update does *not* require any kind of recursive update of controllers the endorsed controller has endorsed.

Next we discuss *send* and *receive* records.

Field	Type	Description
id	send id	uniquely identifies this send record.
resource	resource id	the resource sent.
removable	boolean	defaults to false, true if this has no *incoming*, or as a result of some activity that can only occur on this controller (e.g. minting a currency).
updatable	boolean	defaults to true, set to false as part of endorsement-reduction.
incoming	set of receive ids	the receive records on which this send depends.
outgoing	set of controller ids	the controller(s) to which a resource was sent.

Table 2. Fields of a *send* record.

Field	Type	Description
id	receive id	uniquely identifies this receive record.
resource	resource id	the resource sent.
incoming	set of send ids	non-empty set of send records on other controllers.
live	set of resource ids	the created (and not consumed) resources on this controller which depend on this receive.
outgoing	set of send ids	the sends on this controller created when recources dependent on this receive were sent elsewhere.

Table 3. Fields of a *receive* record.

3.1.1. Send Records

Each controller maintains a set of *send* records, which are created each time a resource transfer from one controller to another begins. To initiate a transfer, a controller consumes the sent resource, and creates a similar resource with a different *terminal* controller, as well as a corresponding *send* record. Each *send* record has an (immutable) unique *send id*, as well as fields described in Table 2. If the transferred resource has ancestors from another controller, the *incoming* field reflects the events when ancestors came to this controller. Similarly, the *outgoing* field represents the destination of the transfer. This can be updated as part of the reducing process (Section 3.4). If a send record's *updatable* field is false, it becomes immutable: it can never change again. This is used in controller DAG reduction (Section 3.4).

A *send* record represents a *promise*: when one controller transfers a resource to another, it promises to accept those resources, or new resources created using those, in transfers back. Pending a review of the returned resources' history, these can be treated just like resources created on the original sending controller. We describe this process in more detail in Section 3.4.

3.1.2. Receive Records

Each controller also maintains a set of *receive* records, which are created each time a transfer from one controller to another completes. Note that a resource cannot be received twice: no state can contain two receives with the same *id* or the same *resource*. To receive a transfer, a controller creates a receive record, as well as a copy of the sent resource, whose terminal controller matches the destination controller. Each *receive* record has an (immutable)

Field	Type	Description
creating	controller id	which controller determines if this resource has been created?
terminal	controller id	which controller determines if this resource has been consumed?
backup	list of controller ids	If the terminal controller halts, who will be the new terminal controller? If that controller has halted, who will be the new terminal controller? and so on…
receives	set of receive ids	the receive records (on the terminal controller) on which this resource depends.

Table 4. Fields in a resource label.

receive id, as well as fields described in Table 3. A receive's *incoming* field must feature at least one send, and all on other controllers: these are the sends which this receive received.

In a safe controller's state, each send in a receive's outgoing field should feature the receive in its incoming field.

Note that send and receive records can be encoded as resources (with sufficiently precise resource logics). For our purposes, it is useful to discuss them as mutable records, even if they are implemented as resources "under the hood."

3.2. Resource Controller Labels

Each resource features a label with information about controllers that can or have affected it. The label includes a DAG representing controllers who have affected the history of this resource. Each *node* of this DAG includes:

- a *controller id*
- a *send id* representing when an ancestor of this resource was transferred out of this controller.
- a *state root* for this controller, after (or equal to) the state when the *send* was created.

For each edge in the DAG $s \longrightarrow s'$, there must be a receive id $r \in s.incoming$ such there is or was a corresponding receive record r with $s' \in r.incoming$. (here we abuse notation and use r for both the receive id and the corresponding receive record.) Without reductions as in Section 3.4, the DAG has an edge whenever there is such an r. In this way, the controller DAG reflects (some of the) history of the resource.

The set of all controllers from all the nodes of this DAG are the resource's *affecting controllers*.

Each resource label also includes the fields in Table 4.

3.3. Creating a resource

A transaction must use as inputs resources with the same terminal controller: the controller that orders the transaction. The terminal controller's state determines if any of the input resources have been consumed (which would make the transaction invalid). To determine if all the input resources have in fact been created, one must check whether the creating controller of the resource has created it. To ensure CRH, any resource with a creating controller other than the terminal controller requires a corresponding receive record to prove it has been created (see Transfers below).

A created resource can *depend* on input resources, meaning the new resource's history includes those input resources, and their histories. In general, it is safe for a created resource to depend on all the input resources used in proofs of the resource's logic; this would work for very general applications. However, we might allow resource logics to specify a subset of the input resources on which they depend. We might imagine a UTXO-style currency application [Kha22] which tracks only input resources of the same currency as dependencies: anything which was necessary to authorize a "spend" isn't considered part of the history of the currency resource.

The resource's creating id is the id of the controller ordering the transaction that created it: this matches the terminal id of all the input resources. If the resource's terminal id is another controller, this is a *transfer*: the transaction must also produce a corresponding *send* record. The resource has "transferred" to a different controller from its ancestors.

Each resource's controller DAG includes the union of the controller DAGs of the input resources on which it depends. If this is a transfer, then the DAG also includes a node for the new send record, and an edge from each sink of the input resource's DAGs to the new node.

Each resource's *receives* field is the union of the *receives* field of all of its dependencies, or for dependencies created on another controller, the corresponding receive record. Each time a resource is created, it must be added to the *live* set of each receive in its *receives* field. Likewise, each time a resource is consumed, it must be removed from the *live* set of each receive in its *receives* field.

3.3.1. Transfers

In order to create a resource on one controller and consume it on another, there are several steps. First, in a transaction on the sending controller:

- Create a new send id
- Create the "sent" resource: it must have a new unique controller DAG sink: this send id. This resource's *receives* field is empty. Do not add this resource to any receive record's *live* field.

- add this send id to the *outgoing* field of all the receives in the resource's dependencies' *receives* fields.
- Create a new send record with this new send id, referencing this resource, with *incoming* set to the union of all of the receives of the dependencies.

Second, on the receiving controller:

- Endorse a state root from the sending controller whose corresponding state features the send record and the sent resource.
- Create a receive record with a unique id, empty outgoing, a live set containing only the sent resource, and an *incoming* field containing only the send id. Future transactions can now consume the resource, and new resources which depend on it must put this receive record in their *receives* field.

For example, in Fig. 1, in order the first transfer (resource 0 on A to resource 1 on B), would require creating a send record s_0 on A, and a receive record r_1 on B, with $r_1.incoming = \{s_0\}$. Any subsequent resources representing the tomato would appear in $r_1.live$, until they are consumed. Eventually, transferring resource 2 to B creates a new send record, s_2 on B, and $r_1.outgoing$ must include s_2. We illustrate the resulting state of affairs in Fig. 3a. Resource 4, then, with a history that depends on 2 sends, would have a Controller DAG: $A : s_0 \longrightarrow B : s_2$, and terminal controller C.

As a more in depth example, we draw the controller DAG for resource 9, with s_n as the send record for sending resource n:

$$\begin{matrix} A : s_0 \to B : s_2 \\ A : s_3 \to C : s_6 \end{matrix} \overset{\to}{\to} D : s_8$$

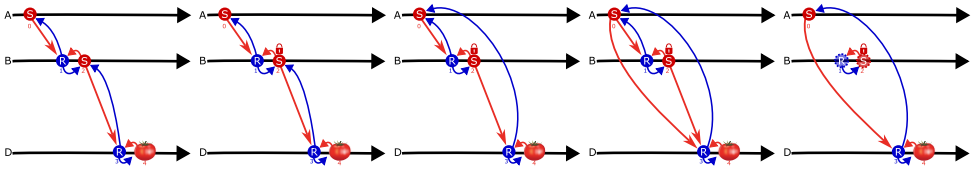
Note that, since they have no *incoming*, s_0 and s_3 have *removable*=false.

3.4. Reducing a Controller DAG

3.4.1. Endorsement

We can use the controller's endorsement map to remove unnecessary elements from a resource's controller DAG. The idea is that if a resource's history involves moving to some intermediate controller, doing some transactions, and then moving on, the intermediate controller is *arbitrary*: as long as other controllers are willing to check the part of the history done on the intermediate controller, they can claim they "may as well have" done those state changes themselves, and remove the intermediate controller. This kind

Figure 3. Steps to generate a *reduce proof* for B in resource 4's DAG (Fig. 1). The proof itself has a state root for A, B, and D, and shows they have the *send* and *receive* objects in **(d)**.



(a) start **(b)** updatable=false **(c)** redirect receives **(d)** update parents **(e)** garbage collect

of endorsement-based DAG reduction generalizes IBC and ICS20's *unwrapping* [Sco23, Goe20, ics24]. In our Fig. 2 example, A could endorse C, D, and (one fork of) B, and then reduce resource 9's controller DAG to just nodes with controller A. The challenge is to do this in a way that provably maintains CCL. We would not want to allow resource 11 to end up with a DAG that doesn't contain B, or someone could consume resources 9 and 11 to create a new resource that doesn't have SRH (it contains a double-consume of a tomato), and also doesn't have B in its controller DAG: that would violate CCL. There are a number of subtleties here, so we start with some definitions.

Definition 1 (Endorsement)**.** *A controller A endorses a state root R if A's endorsement map element for R's controller id is provably after (or equal to) R.*

Definition 2 (Remove)**.** *We remove a node from a resource's controller DAG if we consume the resource, and create an identical one whose controller DAG is missing the node. Instead, the new resource's controller DAG has edges from all of the removed node's parents to all of the removed node's children.*

3.4.2. Reduce Proofs

Before we can remove a node n, we require a *reduce proof* for the send record in n. Conceptually, a reduce proof shows that n's parents and children have endorsed the history that happened on n's controller, and agreed not to endorse any contradictory history. Note that we can only create reduce proofs for sends with removable=true. In "promise" terminology, wherein each send represents a promise to review and accept descendants of the sent resource, if A promises to accept resources from B, and B promises to accept resources from C, then with enough endorsement, we can collapse this to A promising to accept resources directly from C. In taint tracking terminology, the controllers from n's neighbors accept any responsibility for taint n's controller might have caused, by reviewing the history of n's send record.

For a send record s, let $P(s) = \{s' | r' \in s.incoming \wedge s' \in r'.incoming\}$ be the *parents* of s. These sends mark the boundary of s's history on its own controller.

For various reasons, it is convenient to define reduce proofs for sets of

send records. Consider a set of sends Y such that each send record has *removable* =true (which in turn implies that each has parents). We now detail the process of creating a reduce proof for Y.

Let X be the set of parents of sends in Y which are not themselves in Y: $X = \cup_{y \in Y} P(y)/Y$.

Suppose a receive exists for every send in Y. Let Z contain a receive for each send in Y: $\forall y \in Y : \exists z \in Z : z.resource = y.resource$.

Let the ancestors $a : Z \rightarrow 2^X$ of a receive record $z \in Z$ be the elements $x \in X$ such that there is a path (defined by *incoming* field elements) from z to x in which all but the final send element (x) are in Y. These represent the boundary events that "transitively affect" z.

Let C : (set of send or receive $\rightarrow$ set of controller) be the function that maps a set of send or receive records to the controllers of those records, so $C(Y)$ is the set of controllers of the send records in Y.

The steps to create a reduce proof for all the sends in Y are below. In Fig. 3, we illustrate this process with $Y = \{s_2\}$, $X = \{s_0\}$, and $Z = \{r_3\}$.

1. For each $y \in Y$, set $y.updatable$ to false (Fig. 3b). This effectively prevents us from making a reduce proof for any of y's "neighbors" (sends connected to the same receives as y) without making a reduce proof for y. Otherwise, we can get some very peculiar behavior where 2 resources with identical DAGs $A : s_0 \rightarrow B : s_2 \rightarrow C : s_3 \rightarrow D : s_4$ can end up with DAGs $A : s_0 \rightarrow B : s_2 \rightarrow D : s_4$ and $A : s_0 \rightarrow C : s_3 \rightarrow D : s_4$, which can cause problems for maintaining CCL.

2. Let R_Y be the set of state roots of $C(Y)$ just after completing step 1.

3. For each controller in $C(Z)$:

   - endorse all roots in R_Y.
   - for each controller in $C(X)$, endorse a state root after all elements of X on that controller were created. This essentially just recognizes that everything in X has indeed occurred.
   - Update all receives $z \in Z$ by removing elements of Y in $z.incoming$ and adding all elements of $a(z)$ to $z.incoming$ (Fig. 3c). Effectively, controller in $C(Z)$, having reviewed the history of $C(Y)$, accepts responsibility for the history (from X onward) of the resources received.

4. let R_Z be the set of state roots of C_Z just after completing step 3.

5. For each controller in $c \in C_X$:

- endorse R_Y and R_Z.
- update all elements $x \in X$ to include outgoing references to the receives Z: $C(z) \in x.outgoing$ if $\exists z \in Z : c \in z.incoming$ (Fig. 3d).

6. The *reduce proof* for each $y \in Y$ is a collection of state roots from all the controllers in $C(X), C(Y)$, and $C(Z)$, with proofs that the above steps have happened. These proofs show that $\forall y \in Y : y.updatable$=false, all $z \in Z$ now have *incoming* sends from X, and all $x \in X$ have been updated to refer to $C(Z)$ where appropriate.

Once enough reduce proofs have been generated, we can even garbage collect some send and receive records (Fig. 3e). Specifically, when a send record s_0 on controller A has a controller B in its *outgoing* field, but all receive records r_1 on B receiving s_0 have an empty *live* field, and there are reduce proofs for all the sends $r_1.outgoing$, (and A and B has endorsed state roots proving this is the case) we can remove B from the $s_0.outgoing$.

Once all a receive record's incoming sends have removed that receive record's controller from their outgoing fields (and the receive record's controller has endorsed state roots that prove this), we can garbage-collect the receive record: it will no longer be used. Once all of a send record's *incoming* receive records are garbage collected (if the send record has *removable*=true), we know that send will no longer be used, and can garbage collect it.

For liveness, reduce proofs should remain available indefinitely unless there is a way to prove that every resource with the send record in its controller DAG has been consumed. However, they do not need to be stored on any particular controller. We leave reduce proof distribution to future work.

3.4.3. Removing a Node from a Controller DAG

For any transaction that creates a resource r with a node n in its controller DAG, removing n requires a reduce proof for $n.send$. First, r's terminal controller must endorse all the state roots in the proof. Then each node in r's controller DAG must update its state root to whichever is provably after the other: the state root it currently has or the state root with the same controller in the proof. (If there is a state root with the same controller in the proof, but neither is provably after the other, this remove operation is not valid.) Then it can remove n from the controller DAG, and add edges from all of n's parents to all of n's children. For example, given a reduce proof for s_2, we could reduce the controller DAG of resource 9 (Fig. 2):

$$\text{from: } \begin{matrix} A : s_0 \to B : s_2 \\ A : s_3 \to C : s_6 \end{matrix} \overset{\to}{\underset{\to}{}} D : s_8 \qquad \text{to: } \begin{matrix} A : s_0 \\ A : s_3 \to C : s_6 \end{matrix} \overset{\to}{\underset{\to}{}} D : s_8$$

With enough reduce proofs, we could eventually reduce it to just $A : s_0$ and $A : s_3$ (and no edges). Note that, since they have no *incoming*, s_0 and s_3 have *removable*=false, so they will never have a reduce proof.

Theorem 3 (Reduction preserves CCL)**.** *If a safe controller creates a resource r, and valid transactions must follow the "Creating a Resource", endorsement, and "Removing a Node from a Controller DAG" procedures above, then r has CCL.*

Proof. First, we note that all resources created on safe controllers have CRH: safe controllers only create resources through valid transactions with inputs that have CRH, and only accept transferred resources after they have endorsed their history, proving the transferred resources have CRH.

Consider S, the graph of send objects in the most recent state of all safe controllers, and all the most recent states of all forks of unsafe controllers. We define directed edges of S using P: each send references any send which has *ever been* its parent (an *incoming* send of an *incoming* receive).

All paths through r's controllers DAG correspond to a path through S. Some sends in that path may have reduce proofs, so there may be shorter paths.

Either r has an unsafe controller in its DAG or it does not. If r has an unsafe controller in its DAG, then r has CCL. Hereafter we consider the case where r's dag is entirely safe.

Either r has SRH or it does not. If r has SRH, then r has CCL. Hereafter we consider the case where r's dag is entirely safe but r does not have SRH.

Given that r has CRH, lack of SRH means there is a fork in r's history. Some unsafe controller, which we will call C, created 2 contradictory *send*s s_1 and s_2, which share an *incoming* element, but neither of which is provably after the other. If we consider all the transactions in r's history in some serialized order, and consider the set of all sends in all (unconsumed) resources' controller DAGs, we show that no transaction can remove the last element of the last pair of contradictory sends.

Specifically, suppose that r's ancestor a wishes to remove the last element s_2 of a contradictory pair of sends from unsafe controller C. These sends share an *incoming* element and therefore an ancestor send s_0 on another controller, which we'll call B. Without loss of generality, s_0 has $updatable$=false, since without an updatable ancestor, B cannot be removed. Therefore, $B : s_0$ has not yet been removed from a's controller DAG. B cannot endorse both contradictory sends and add them to s_0 without forking and creating two contradictory versions of s_0. Thus it is impossible to remove the last element of the last contradictory pair. Therefore, if r does not have SRH, it has an unsafe controller in its DAG (and thus its affecting controllers). Thus CCL is guaranteed.

□

Moving and Reducing Together. It is possible to move a resource to another controller and remove the sender together. Specifically, the send is marked *updatable*=true as soon as its created, and the receive updates its *incoming* field immediately. With one transaction from the parents of the send, the received resource could remove the send the first time it's used.

This is useful when, for example, some state has been temporarily moved from a base chain to a side chain, and the side chain wants to move it back to the base chain. The side chain would condense all its nodes on the controller graph to a single node, and set that node's reducing field. The base chain could then endorse the side chain, and remove the side chain from the resource's controller DAG.

3.4.4. Removing a Halted Controller

Controller states carry a boolean *halted* flag, which defaults to false. At any time, changing this flag to true is a valid transition. All transitions must be ordered / decided by the controller, so this transition would represent the controller deciding to halt. There are no valid state transitions starting with a state featuring a halted flag set to true. In effect, a *halt* transaction creates a send for every unconsumed resource with *outgoing* set to the resource's backup controller, and *updatable* set to false. A halted controller can also be said to have a receive record for every resource sent to it, followed immediately by a send.

A connected sub-DAG (of a controller DAG) consisting entirely of halted controllers can be removed if all of the sub-DAG's children endorse every node in the sub-DAG, and all of the sub-DAG's parents endorse every element of the sub-DAG, and all of the sub-DAG's children. We can be certain that the halted controllers will not endorse some contradictory send (unless the halted controllers have forked), since they are halted, and cannot endorse anything.

3.4.5. Backup Controllers

If the terminal controller has halted, a backup controller (who has endorsed a halted state for the halted controller) can assume control of the resource with a receive record. In order to show the corresponding send has in effect happened, the backup controller must also show that any earlier controllers in the resource label's *backup* field have halted *without* receiving this resource. This allows resources to survive a halted controller.

A good default backup controller list might be derived some kind of path "up" the controller DAG, representing a history of controllers who have recently affected this resource.

3.4.6. Emergency Override Condition (EOC)

Controllers decide on an order of transactions. Usually this decision is defined by some kind of signature or record of consensus proving that some computer or computers decided on some ordering. In principle, not all decisions (not all transactions) require the same procedure. In particular, we can add to a controller a special case that is only allowed to do the halt transition. We call this the Emergency Override Condition (EOC). For example, this could be some kind of high-integrity (but slow) "supervisor" who is trusted to declare when a controller's consensus mechanism (e.g. a blockchain) is dead. If an EOC incorrectly halts a controller, but that controller continues ordering, it has forked: mistaken EOCs make controllers *unsafe*.

This does mean there could be multiple controllers who differ only in their EOC, but are otherwise maintained by the same consensus or machine. These are still distinct controllers. These controllers would be trivially able to endorse each other frequently, making transferring resources between them very easy. We might even be able to do atomic transactions consuming resources from both, although we leave that to future work.

3.5. Checking SRH

The protocol described does not guarantee SRH. A violation can occur when one of the controllers in the DAG has forked. In our Fig. 2 example, a transaction could create a new resource which depends on resources 9 and 11. Its controller DAG would include the forked controller B (preserving CCL), but its history would not be serializable, and in fact contains a double-consumed tomato.

This can happen even if the controller appears only once in the DAG: the "other" fork may have been endorsed and removed by some parent in the DAG. The easiest way to verify SRH of a resource or set of resources is to get a recent state root from each of the controller ids in their controller DAGs (and creating controllers), as well as all of their corresponding endorsed state roots for all the controller ids in the DAGs (and creating controllers), and then prove that for each controller: the "recent" state root is provably after all of the endorsed state roots or state roots in the DAG. This would prove that no one has endorsed any fork contrary to the state roots in the DAG, and so the history is consistent.

To understand why this holds, consider that a node with no parents cannot be removed (its send must have *removable*=false), and if a node with an honest parent has forked, both sides cannot be removed without the honest parent endorsing one. The honest parent then cannot be removed from the other side, as it cannot endorse the contradictory send.

To ensure this information is always available would require each resource to carry an awful lot of information. This is another case where a little trust

goes a long way: if you trust that all the controllers in a resource's controller DAG haven't forked, then CCL (with CRH) implies SRH. If you want to check, you can, but it requires acquiring more data.

4. Future Work

There are several directions for future work. For instance, although all the techniques we have found for preserving SRH are expensive (Appendix A), we have not proven that preserving SRH is inherently expensive. There may yet be some technique that is satisfactory.

Likewise, we have not proven that it is crucial for controllers to retain resource-specific records to allow endorsement-based reduction (e.g. sends and receives). There are almost certainly ways to batch and amortize these costs. Perhaps there is even a system that only requires endorsement maps, like Appendix C, while preserving the local "only neighbors in the controller DAG need to participate in a reduce operation" property.

Furthermore, we make no claims about data availability, which is crucial for liveness. Despite ordering transactions, controllers do not necessarily need to be able to calculate their state: zero-knowledge proofs can show that a transaction is valid without revealing much of what happened in it [KST22, KG24]. We leave the difficult task of tracking what data needs to be available (e.g. current state of send and receive records), and to whom, to future work.

5. Concluding remarks

We detail a technique for maintaining efficient controller labels with endorsement-based reduction. Our technique keeps each transaction simple and inexpensive, and allows resources to carry relatively small labels, while simultaneously ensuring Causal Resource Histories and Consistent Controller Labels. Fully Serializable Resource History remains check-able, although we show how a trust can improve performance, allowing CCL to be sufficient. As part of a larger cross-domain architecture, our technique will allow more secure, more flexible cross-domain applications.

References

AGMS18. Karolos Antoniadis, Rachid Guerraoui, Dahlia Malkhi, and Dragos-Adrian Seredinschi. State Machine Replication Is More Expensive Than Consensus. In Ulrich Schmid and Josef Widder, editors, *32nd International Symposium on Distributed Computing (DISC 2018)*, volume 121 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 7:1–7:18, Dagstuhl, Germany, 2018. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. (cit. on pp. 2 and 4.)

AM17. Ittai Abraham and Dahlia Malkhi. The blockchain consensus layer and bft. *Bull. EATCS*, 123, 2017. (cit. on p. 3.)

Ama24. Amazon. AWS AppSync, 2024. (cit. on p. 1.)

But13. Vitalik Buterin. Ethereum white paper: A next generation smart contract & decentralized application platform. 2013. (cit. on pp. 1 and 3.)
CMA17. Ethan Cecchetti, Andrew C. Myers, and Owen Arden. Nonmalleable information flow control. *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017. (cit. on p. 2.)
Goe20. Christopher Goes. The interblockchain communication protocol: An overview. 2020. (cit. on pp. 2 and 13.)
ics24. Draft spec for multi-denom packets ics20 v2, march 2024. (cit. on pp. 2 and 13.)
JP 24. JP Morgan. Onyx: Transforming the future of banking., 2024. (cit. on p. 1.)
KG24. Yulia Khalniyazova and Christopher Goes. Anoma Resource Machine Specification. *Anoma Research Topics*, Jan 2024. (cit. on pp. 2, 3, 4, 6, and 19.)
Kha22. Khadija Khartit. UTXO Model: Definition, How It Works, and Goals, april 2022. (cit. on p. 11.)
KST22. Abhiram Kothapalli, Srinath Setty, and Ioanna Tzialla. Nova: Recursive zero-knowledge arguments from folding schemes. In Yevgeniy Dodis and Thomas Shrimpton, editors, *Advances in Cryptology – CRYPTO 2022*, pages 359–388, Cham, 2022. Springer Nature Switzerland. (cit. on pp. 4, 6, and 19.)
LAG+17. Jed Liu, Owen Arden, Michael D. George, Andrew C. Myers, Toby Murray, Andrei Sabelfeld, and Lujo Bauer. Fabric: Building open distributed systems securely by construction. *J. Comput. Secur.*, 25(4–5):367–426, jan 2017. (cit. on p. 2.)
LFKA11. Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with cops. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, page 401–416, New York, NY, USA, 2011. Association for Computing Machinery. (cit. on p. 6.)
Mer88. Ralph C. Merkle. A digital signature based on a conventional encryption function. In Carl Pomerance, editor, *Advances in Cryptology — CRYPTO '87*, pages 369–378, Berlin, Heidelberg, 1988. Springer Berlin Heidelberg. (cit. on p. 3.)
Nak08. Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, Dec 2008. Accessed: 2024. (cit. on p. 4.)
Pap79. Christos H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, oct 1979. (cit. on p. 6.)
Pos24. PostgreSQL Global Development Group. Constraints. *PostgreSQL 16 Documentation*, 2024. (cit. on p. 3.)
Sch90. Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22(4):299–319, dec 1990. (cit. on pp. 2 and 4.)
Sco23. AJ Scolaro. What is ibc? 2023. (cit. on pp. 2 and 13.)
WGDW22. Dong Wang, Yu Gao, Wensheng Dou, and Jun Wei. Dista: Generic dynamic taint tracking for java-based distributed systems. In *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 547–558, 2022. (cit. on pp. 2 and 5.)

A. Resource Vector Clocks: a Technique for SRH

One approach to maintain SRH (no "double-consumes" in the history of one resource) would be for each resource label to carry a *vector-clock*: a map from controller ids to state roots. Resource labels also designate a creating controller (which must be in the vector-clock) and a terminal controller. The "affecting controllers" in this technique are the keys of the vector-clock.

When a transaction creates a new resource, the new resource's vector-clock must feature all the controllers from all of the input resources' vector clocks, each mapped to a state root that is provably after (or equal to) the corresponding state roots from each of the input resources. This ensures the history of each resource cannot include a fork from any controller. The state root for the controller creating the resource must be that controller's current state root.

A.1. Problem: Cost

Every transaction now has to do "state root is after" proofs for an unlimited number of controllers.

A.2. Problem: Can't Remove Controllers

There is no "endorse" mechanism that would allow a controller that appears in an ancestor to be absent from a descendant's vector-clock. This could lead to very large vector clocks in every resource, which result in lots of proofs with each transaction.

A fairly common pattern in the blockchain industry is to transfer resources from a more trustworthy "base chain" to a "side-chain," or "L2" chain, do some transactions, and then transfer them back to the base chain, which somehow "endorses" the side-chain changes, so *it doesn't matter where they happened.* Fundamentally, such an endorsement technique requires that the "base chain" remembers what it has endorsed, and doesn't endorse any conflicting histories. We have not encoded this in our vector clock model. Furthermore, it is difficult to add: it is not clear which controllers should be empowered to endorse and remove which other controllers.

B. Consistent Controller State: Even Stronger than SRH

One property we might want would be for each correct controller's state (the set of resources it can use as input for valid transactions) to reflect some fully serializable history featuring only correct transactions (and no forks). In Fig. 2, for instance, controllers *C* and *D* have Consistent Controller State. This would mean that, for example, if another controller has forked and produced resources representing a double-spend, no correct controller's state will contain resources affected by "both sides" of this double-spend. In Fig. 2, controller *A* does not maintain consistent controller state, because its state contains resources 11 and 9, which descend from conflicting sides of a fork (*B*).

Consistent controller state is equivalent to requiring that all the resources on a controller together have SRH. Likewise, SRH would be equivalent to consistent controller state if each controller had only one resource.

The techniques we have found to maintain consistent controller state are incredibly strict, which is why we generally use weaker properties.

B.1. Recursive Endorsement: a Technique for Consistent Controller State

The idea with this technique is to have each controller fully *endorse*, or check, the entire history of other controllers, including the history of controllers they've endorsed. This ensures that the state each controller recognizes is fully consistent.

B.1.1. Controller State

Suppose that each controller has a unique *controller id*. Suppose also that each controller's entire state (including representations of resources created, consumed, and everything else) can be uniquely identified with a digest or hash called a *state root*. Each controller's state also includes a *recursive endorsement map*, which is a map from *controller id*s to *state root*s. Its entry for itself is, in effect, its own state root. At any time, (as a valid transaction), it can update the entry for any set of controllers, provided:

- the new state root for all the controllers is proven after the old state root for all the controllers (or genesis, if there isn't one)
- The new recursive endorsement map includes state roots for all the controllers in all the recursive endorsement maps of the controllers updated.
- For pairs of elements $(\langle C_a, R_a\rangle, \langle C_b, R_b\rangle)$ in the recursive endorsement map, if R_a's recursive endorsement map includes an entry $\langle C_b, R'_b\rangle$, then R_b is provably after R'_b.

Basically, each update *includes* a recursive history check for all the controllers on which it depends. This makes recursive endorsement map updates *much* more expensive than regular endorsement map updates (Section 3).

B.1.2. Resource Labels

Each resource label only specifies a *terminal controller* id, a *creating controller* id, and a *creating controller state root*. The creating controller state root is the state of the controller just after creating the resource. One possible type of transaction consumes a resource, and replaces it with a similar resource featuring a different terminal controller id. This represents "transferring" the resource to a different controller. This can be constrained by the resource logic.

In general (not only for transfers), a resource cannot be used in a transaction unless:

- the transaction is ordered and executed on the resource's terminal controller.
- the terminal controller's recursive endorsement map state root for the resource's creating controller is proven after (or equal to) the resource's creating controller state root.
- all resources created by this transaction have a creating controller equal to the terminal controller of the inputs.
- all resources created by this transaction have a creating controller state root representing the state of the creating controller after this transaction.

In other words, a resource can't be used on a controller until its history is fully endorsed.

The nice thing is that transactions using resources created on the terminal controller are cheap, and resources each carry a constant amount of information about their controllers: they do not have to carry a set of "affecting controllers." A resource is "as trustworthy" as its creating controller, and any controller that has endorsed that creating controller's state root (equal to or after this resource).

B.1.3. Problem: Forks Split the World

Suppose controller B forks, and A updates its recursive endorsement vector with one side of the fork, and D updates its recursive endorsement vector with the other side. (This is what would happen in our Fig. 2 example.) This means that hereafter, any resources A creates cannot be used on D, and vice-versa. (The Fig. 2 transfer of resource 8 on D to resource 9 on A would be impossible.) In fact, all controllers (if they ever use any resource from B), divide into 2 groups (one for each side of the fork), which can never interact again.

This problem doesn't happen (or is less bad) if we only have to preserve SRH, because transactions only depends on input resources' history, so, in our Fig. 2 example, the transfer of resource 8 on D to resource 9 on A would be ok. Furthermore, resources on A that weren't actually affected by anything after the B fork could still transfer to D (which they can't in consistent controller state).

C. Ancestral Endorsement Reduction

In Section 3.4, we discuss a technique for removing nodes from the controller DAG that requires controllers to keep send and receive records, in addition to

an endorsement map. This technique does not require such records. Unfortunately, this technique is not *local*: it involves participation from controllers beyond just the neighbors of the nodes to be removed, or the terminal controller.

Each resource's label contains a DAG of state roots. The resource's affecting controllers are the controllers of all the state roots in its DAG. Each created resource's DAG contains the DAGs from all of their dependencies, with a new state root as a sink, representing the state of the creating controller upon completion of this transaction. We also allow transactions to update any state root to a state root provably after the old one.

We allow a transaction to remove a node from a resource DAG if:

- the node has parents (it is not a source in the DAG)
- all of the node's ancestors endorse the node

This preserves CCL, because we cannot remove sources, and if a controller forks, its parent cannot endorse both sides of the fork (without forking themselves). Intuitively, if there is a double-spend, at least one side of the spend will never be able to remove all the forked controllers from its controller DAG.

The problem is that this is very expensive. It involves activity from an arbitrarily large set of controllers for every removal. What's more, sources in the controller DAG essentially have to endorse every removal, so they can end up doing a lot of work (although this can be batched and amortized), with no way to offload that responsibility to anyone else. If, for example, a cryptocurrency issuer moves tokens to other controllers and then becomes inactive for a long time (perhaps because it doesn't want to issue new currency for a while), then none of those tokens can do any ancestral endorsement reduction during that time.

Unfortunately, resources cannot allow both Ancestral Endorsement Reduction and reduction as we've outlined above: we have not found a CCL-preserving system that does both.